

---

# Formal Framework for Proof Generating Optimizers

Vom Fachbereich Informatik  
der Universität Kaiserslautern  
zur Verleihung des akademischen Grades  
Doktor der Naturwissenschaften (Dr. rer. nat.)  
genehmigte Dissertation

von

Marek Jerzy Gawkowski

November 25, 2008

Termin der wissenschaftlichen Aussprache:	20.04.2009
Dekan:	Prof. Dr. Karsten Berns
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Karsten Berns
1. Berichterstatter:	Prof. Dr. Arnd Poetzsch-Heffter
2. Berichterstatter:	Prof. Dr. Klaus Schneider

D 386



**Zusammenfassung.** Ein Softwaresystem wird meistens entweder in einer Spezifikations- oder einer höheren Programmiersprache geschrieben. Dessen Laufzeitverhalten wird jedoch durch den Maschinencode, der das Ergebnis der Übersetzung seiner Beschreibung in einer höheren Quellsprache ist, bestimmt. Für ein unkritisches Softwaresystem mag es ausreichen, dieses auf die Abwesenheit der Fehler durch Überprüfung von Testläufen zu testen. Für ein sicherheitskritisches Softwaresystem kommen jedoch erschwerend zwei Faktoren hinzu. Erstens, der *Software-Producer*, der dieses Softwaresystem implementiert hat, muss formal nachweisen, dass sein Maschinencode die formale Spezifikation, die vom *Software-Consumer* ausgestellt wird und seine *Safety-Policy* beschreibt, erfüllt. Zweitens, die Quellsprachenbeschreibung des Softwaresystems ist das Ergebnis der Anwendung der statischen Analyse und formaler Methoden, welche diese Beschreibung in Richtung Effizienz und Korrektheit optimieren. Demnach ist das Ziel der Erstellung der formal nachweisbaren Übersetzungskorrektheit ausschlaggebend für den Erfolg bei der Implementierung eines sicherheitskritischen Softwaresystems.

In dieser Dissertation schlage ich einen Ansatz zu zertifizierenden Übersetzern vor, der die Formalismen und Techniken aus den Bereichen der mathematischen Logik und Programmiersprachen anwendet, um das letztgenannte Ziel zu erreichen. Ich nenne den Ansatz *foundational translation validation* (FTV). Ein Compilerhersteller, der diesen Ansatz verfolgt, implementiert ein FTV-System, das neben dem Übersetzerprogramm aus einem *Spezifikations- und Verifikationsframework* (SVF) besteht. Das SVF wird in der Logik höherer Stufe (HOL) implementiert und seine wichtigste Komponente ist ein *Übersetzungsvertrag*, der die Formalisierungen der Quell- und Zielsprache sowie eines zweistelligen *Übersetzungskorrektheitsprädikates*  $\text{corrTrans}(S, T)$  über die Quellprogramme  $S$  und die Zielprogramme  $T$  enthält. Die Formalisierungen der Sprachen werden als sog. tiefe Einbettungen in HOL realisiert. Durch Verwendung von tiefen Einbettungen erreicht man, dass ganze Programme in diesen Sprachen als HOL-Konstanten deklariert werden können. Die Definition von  $\text{corrTrans}$  drückt formal aus, was es heißt, dass  $T$  die korrekte Übersetzung von  $S$  ist, und verwendet explizit die Definitionen der Programmsemantiken der Quell- und Zielsprachen, die vom Übersetzungsvertrag festgelegt werden. Im Anschluss an die Übersetzung des Quellprogramms übersetzt der Übersetzerprogramm das Quell- und das Zielprogramm in ihre textuellen Repräsentationen als HOL-Konstanten  $S$  und  $T$  und generiert einen *Korrektheitsbeweis* für die Gültigkeit der Aussage  $\text{corrTrans}(S, T)$ . Ich nenne ein Übersetzerprogramm, das den FTV-Ansatz verfolgt, ein *beweisgenerierender Übersetzer* und ein Beweisskript, das den Korrektheitsbeweis enthält, ein *Übersetzungszertifikat*.

Die Idee, dass Programme in Korrektheitsbeweisen als logische Konstante repräsentiert werden können, ist nicht neu und wurde von dem *foundational proof-carrying code*-Ansatz übernommen. Andere Merkmale des Ansatzes, die neuartig sind, und ihn von anderen Ansätzen zu zertifizierenden Übersetzern, wie *proof-carrying code* (PCC) und *translation validation* (TV), unterscheiden, sind die folgenden:

*Die Anwesenheit eines expliziten und formalen Übersetzungsvertrags:* Die Ansätze PCC und TV sehen weder die Formalisierungen des Übersetzungsvertrags noch die Verwendung von Theorembeweisern zum Zwecke der Verifikation der Übersetzungen vor. Stattdessen werden zum Zwecke der Übersetzungsverifikation dedizierte Übersetzungsverifizierungswerkzeuge verwendet. Dabei werden die operationalen Semantiken der Quell- und Zielsprache sowie ein sogenanntes Übersetzungskorrektheitskriterium mit Mitteln der Implementierungssprachen dieser Werkzeuge kodiert und in diese eingebettet.

*Die Repräsentation der Programme in Korrektheitsbeweisen als logische Konstanten:* Im Rahmen der Ansätze PCC and TV werden die Programme in ihre semantische Abstraktionen übersetzt. Diese Abstraktionen werden den Übersetzungsverifizierungswerkzeugen als Eingaben übergeben.

*Zertifizierung von Programmtransformationsketten:* Im Gegensatz zu dem TV-Ansatz, der auf die Zertifizierung von einzelnen Programmtransformationen spezialisiert ist, ermöglicht der FTV-Ansatz die Zertifizierung von ganzen Ketten von Programmtransformationen. Dies rührt von der Tatsache her, dass die Definitionen der Programmsemantikfunktionen im Übersetzungsvertrag die Programme in Elemente der Menge, die durch eine transitive Relation " $\leq$ " partiell geordnet sind, abbilden.

Diese Dissertation erläutert den FTV-Ansatz anhand eines FTV-Systems, das im Rahmen der Doktorarbeit implementiert wurde. Das System besteht aus einem Übersetzer-Frontend, das seine Optimierungsphase zertifiziert und einem SVF, das mit dem Theorembeweiser Isabelle/HOL implementiert wurde. Der Schwerpunkt dieser Dissertation liegt auf der Beschreibung des SVF und seiner Techniken der Übersetzungsverifikation.

**Summary.** Most software systems are described in high-level model or programming languages. Their runtime behavior, however, is determined by the compiled code. For uncritical software, it may be sufficient to test the runtime behavior of the code. For safety-critical software, there is an additional aggravating factor resulting from the fact that the code must satisfy the formal specification which reflects the *safety policy* of the *software consumer* and that the *software producer* is obliged to demonstrate that the code is correct with respect to the specification using formal verification techniques. In this scenario, it is of great importance that static analyses and formal methods can be applied on the source code level, because this level is more abstract and better suited for such techniques. However, the results of the analyses and the verification can only be carried over to the machine code level, if we can establish the correctness of the translation. Thus, compilation is a crucial step in the development of software systems and formally verified *translation correctness* is essential to close the formalization chain from high-level formal methods to the machine-code level.

In this thesis, I propose an approach to certifying compilers which achieves the aim of closing the formalization chain from high-level formal methods to the machine-code level by applying techniques from mathematical logic and programming language semantics. I propose an approach called *foundational translation validation* (FTV) in which the software producer implements an FTV system comprising a compiler and a *specification and verification framework* (SVF) which is implemented in higher-order logic (HOL). The most important part of the SVF is an explicit *translation contract* which comprises the formalizations of the source and the target languages of the compiler and the formalization of a binary *translation correctness predicate*  $\text{corrTrans}(S, T)$  for source programs  $S$  and target programs  $T$ . The formalizations of the languages are realized as deep embeddings in HOL. This enables one to declare the whole program in a formalized language as a HOL constant. The predicate formally specifies when  $T$  is considered to be a correct translation of  $S$ . Its definition is explicitly based on the program semantics definitions provided by the translation contract. Subsequent to the translation, the compiler translates the source and the target programs into their syntactic representations as HOL constants,  $S$  and  $T$ , and generates a proof of  $\text{corrTrans}(S, T)$ . We call a compiler which follows the FTV approach a *proof generating compiler*.

Our approach borrows the idea of representing programs in correctness proofs as logic constants from the foundational proof-carrying code (FPCC) approach. Novel features that distinguish our approach from further approaches to certifying compilers, such as proof-carrying code (PCC) and translation validation (TV) are the following:

*The presence of an explicit translation contract formalized in HOL:* The approaches PCC and TV do not formalize a translation contract explicitly. Instead of this, they incorporate operational semantics and translation correctness criterion in translation validation tools on the programming language level.

*Representation of programs in correctness proofs as logic constants:* The approaches PCC and the TV translate programs into their representations as semantic abstractions that serve as inputs for translation validation tools.

*Certification of program transformation chains:* Unlike the TV approach, which certifies single program transformations, the FTV approach achieves the aim of certifying whole chains of program transformations. This is possible due to the fact that the translation contract provides, for all programming languages involved in the program transformation chain, definitions of program semantics functions which map programs to mathematical objects that are elements of a set with an (at least) partial order " $\leq$ ". Then, the proof makes use of the fact that the relation " $\leq$ " is transitive.

In this thesis, the feasibility of the FTV approach is exemplified by the implementation of an FTV system. The system comprises a compiler front-end that certifies its optimization phase and an accompanying SVF that is implemented in the theorem prover Isabelle/HOL. The compiler front-end translates programs in a small C-like programming language, performs three optimizations: constant folding, dead assignment elimination, and loop invariant hoisting, and generates translation certificates in the form of Isabelle/HOL theories. The main focus of the thesis is on the description of the SVF and its translation verification techniques.

**Acknowledgement.** First and foremost, I would like to thank my supervisor Arnd Poetsch-Heffter for his expert guidance, constant encouragement and enduring patience during my doctoral research.

I thank Klaus Schneider for being my referee, and am I indebted to him to his numerous comments on this dissertation, which increased its technical and scientific quality significantly.

I thank the - former and current - members of our working group, namely (in alphabetical order): Anca Bealu, Jan Olaf Blech, Giorgio Dal Molin, Christoph Feller, Jean-Marie Gaillourdet, Kathrin Geilmann, Patrick Michel, Nicole Rauch, Markus Reitz, Ina Schaefer, Jan Schäfer, Christian Stenzel, and Yannick Welsch, as well as staff members of our chair, for the excellent working atmosphere and the technical and organizational support.

Last but not the least, special thanks go to my family. I thank my mother Teresa and my father Eugeniusz for their love and support. Finally, I thank my wife Małgorzata for her support and patience and our daughter Marianna who's birth right at the beginning of writing this dissertation gave me zest for life and inspired me with tremendous power to find the golden mean required to write this dissertation as concise as possible but not too concise.



*To Matgorzata and Marianna*



---

# Contents

<b>1</b>	<b>Introduction</b> .....	1
1.1	Approaches to certifying compiler .....	3
1.2	Our approach .....	7
1.3	Objectives of this thesis .....	9
1.4	The PROGENCO project .....	10
1.5	Overview of our implementation .....	13
1.5.1	Compiler front-end .....	13
1.5.2	Specification and verification framework .....	18
1.6	Related work .....	23
1.6.1	Work on certified compilers .....	24
1.6.2	Work on certifying compilers .....	26
1.7	Overview of the thesis .....	29
<b>2</b>	<b>Preliminaries</b> .....	33
2.1	Modelling and proving in Isabelle/HOL .....	33
2.1.1	Terms, types, formulae and theories .....	33
2.1.2	Recursive datatypes and functions .....	34
2.1.3	Isabelle/HOL library .....	34
2.2	Programming language formalization style .....	36
2.3	Conventions .....	36
<b>3</b>	<b>Overview of the SVF</b> .....	39
3.1	Layer 0: Higher-order logic .....	39
3.2	Layer 1: Logic extension .....	39
3.3	Layer 2: Translation contract .....	41
3.4	Layer 3: Type safety proofs .....	43
3.5	Layer 4: Compiler phase independent translation correctness criterion .....	44
3.6	Layer 5: Translation correctness criteria for particular compiler phases .....	48
3.7	Layer 6: Proof environments specific to particular proof tasks .....	49
3.8	Proof checking: an example .....	51
<b>4</b>	<b>Translation contract</b> .....	55
4.1	Abstract syntax of IL .....	55
4.2	Semantics of IL .....	57
4.3	Translation correctness predicate .....	65

<b>5</b>	<b>Type safety of the language IL</b> .....	69
5.1	Well-formedness .....	69
5.2	Well-typedness .....	72
5.3	Conform configuration .....	77
5.4	Type safety theorem .....	78
<b>6</b>	<b>Optimization independent translation correctness criterion</b> .....	81
6.1	Overview .....	82
6.2	Block position environments .....	85
6.3	Well-formedness of block position environments .....	97
6.4	Formalization of the language $IL'$ .....	103
6.4.1	Abstract syntax of $IL'$ .....	103
6.4.2	Semantics of the $IL'$ language .....	103
6.5	Equality of the semantics of the languages $IL$ and $IL'$ .....	110
6.6	Formalization of the language $IL''$ .....	113
6.6.1	Abstract syntax .....	113
6.6.2	Semantics of the language $IL''$ .....	114
6.7	Bisimulation predicate on pairs of $IL''$ program executions .....	116
6.8	Optimization independent translation correctness criterion .....	118
6.9	Optimization independent translation correctness theorem .....	118
<b>7</b>	<b>Translation correctness criteria for particular optimizations</b> .....	121
7.1	SVF for CF optimizations .....	122
7.1.1	Abstract syntax of CPA results .....	123
7.1.2	Bisimulation relation for the CF optimization .....	124
7.1.3	Optimization correctness criterion for the CF optimization .....	127
7.1.4	Verification of the optimization correctness criterion $TCC_{CF}$ .....	142
7.2	SVF for DAE optimizations .....	143
7.2.1	Abstract syntax of LA results .....	145
7.2.2	Bisimulation relation for the DAE optimization .....	145
7.2.3	Optimization correctness criterion for the DAE optimization .....	150
7.2.4	Verification of the optimization correctness criterion $TCC_{DAE}$ .....	163
7.3	SVF for NI optimizations .....	164
7.3.1	Corresponding block edge pairs .....	168
7.3.2	Bisimulation relation for the NI optimization .....	168
7.3.3	Optimization correctness criterion for NI transformations .....	174
7.3.4	Verification of the optimization correctness criterion $TCC_{NI}$ .....	184
7.4	SVF for RAI optimizations .....	185
7.5	SVF for RAE .....	187
7.5.1	Abstract syntax of AEA results .....	188
7.5.2	Bisimulation relation for the RAE optimization .....	189
7.5.3	Optimization correctness criterion for the RAE optimization .....	194
7.5.4	Verification of the optimization correctness criterion $TCC_{RAE}$ .....	206
<b>8</b>	<b>Evaluation</b> .....	209
8.1	Proof script size .....	209
8.2	Performance .....	213
8.3	Framework evaluation .....	215

<b>9</b>	<b>Conclusions and future work</b> .....	217
9.1	Contributions .....	217
9.2	Future work .....	219
<b>A</b>	<b>Intermediate programs illustrating work-flow of our front-end</b> .....	223
<b>B</b>	<b>Specification of optimization relation for constant folding</b> .....	229
B.1	Optimization patterns for expressions .....	229
B.2	Optimization patterns for l-value .....	240
<b>C</b>	<b>Specification of optimization relation for nop insertion</b> .....	251
C.1	Assignment .....	252
C.2	Printi .....	255
C.3	Branch .....	258
C.4	Goto .....	261
C.5	Exit .....	264
<b>D</b>	<b>The companion CD</b> .....	267
D.1	The content .....	267
D.2	Compiling the frontend .....	268
D.3	Using the frontend .....	269
D.4	Using the SVF .....	269
	<b>References</b> .....	273



# Chapter 1

---

## Introduction

The compiler is a crucial part in the development of software systems. Most software systems are described in high-level model or programming languages. Their runtime behavior, however, is determined by the compiled code. For uncritical software, it may be sufficient to test the runtime behavior of the code. If an error is detected, it can be caused by the programmer, by the compiler, or by a semantical ambiguity, e.g. the programmer might assume a particular evaluation order of expressions that is not realized by the used compiler. For safety-critical software, there is an additional aggravating factor resulting from the fact that the code must satisfy the formal specification which reflects the *safety policy* of the *software consumer* and that the *software producer* is obliged to demonstrate that the code is correct with respect to the specification using formal verification techniques. In this scenario, it is of great importance that static analyses and formal methods can be applied on the source code level, because this level is more abstract and better suited for such techniques. However, the results of the analyses and the verification can only be carried over to the machine code level, if we can establish the correctness of the translation. Thus, translation correctness is essential to close the formalization chain from high-level formal methods to the machine-code level.

Following the classification given in [67] and [93], we distinguish between two general approaches to how one establishes that a translation performed by a compiler is correct:

- *Certified compiler approach*: The software producer provides two proofs that both the compiler algorithm defines a correct translation for all given well-formed input programs (*compiler algorithm correctness*) and that the algorithm is correctly implemented on a given machine (*compiler implementation correctness*). The software consumer's confidence in the correctness of a translation is due to the fact that it was performed by a certified compiler.
- *Certifying compiler approach*: The software producer equips the compiler with a certifying unit which, for each translation run, takes the source and the target programs as input and returns a formal proof, a *translation certificate*, that the target program is a correct translation of the source program. Whenever a translation is performed by the compiler, an accompanying certificate is generated by the certifying unit and it is attached to the translation result. The certificate can serve for two different purposes: Firstly, the certificate can be verified directly by the compiler user. Secondly, the certificate can be verified by the software consumer in a scenario with an untrusted software producer delivering software to the software consumer. The software consumer's confidence in the correctness of a translation is due to the fact that there exists a formally verifiable proof that the translation is correct.

Both approaches refer to the notions of *translation correctness criterion/predicate* and *translation correctness proofs* to provide confidence in the correctness of translations. The translation

correctness proof shows that the source and the target programs of a translation fulfill the translation correctness criterion.

The definition of the translation correctness criterion/predicate is based on the notion of *translation relation* over program pairs and it says that a target program is a correct translation of a source program iff the pair consisting of these programs is in the relation. The translation relation is specified by a predicate function which is based on the definitions of the source and the target *programming language semantics* which map respective programs to their meanings. Further, both approaches aim at providing high confidence in the correctness by focusing on machine-checkable proofs. This requires the existence of a formal *specification and verification framework (SVF)* within which the above notions and correctness proofs can be formalized and conducted, respectively. In the case of certified compilers, the SVF is a proof assistant within which a compiler correctness proof is conducted. In the case of certifying compilers, the SVF can be either a proof assistant or a program, a *translation checker*. As of proof assistants, proofs have the form of proof scripts in the language provided by the proof assistant. The translation checker can be either a general purpose program, such as a model checker or verification condition verifier or a dedicated program, a *translation validator*, which is a part of checking infrastructure implemented together with the certifying compiler program. In both cases, the translation checker provides a language for encoding of program abstractions and checks whether source and target program abstractions generated by a certifying compiler fulfill a predefined translation correctness criterion.

In general, the certifying compiler approach has four advantages compared to the certified compiler approach. Firstly, the issue of implementation correctness can be completely avoided, the implementation of the compiler algorithms on a hardware system does not have to be trusted or to be proved correct (cf. [94]). This is due to the fact that a translation certificate makes no statements about the algorithm or the implementation details of the compiler, which generated it, but only about two programs associated to a corresponding translation, i.e. the source and the target programs. Secondly, the technique provides, similar to the proof carrying code approach ([76, 75, 4]), a clear interface between compiler producer and user. In the certified compiler approach, the software producer has to provide access to the compiler correctness proof if he seeks the compiler user's confidence in the translation correctness. This results in revealing the internal details of the compiler. In contrast, the translation correctness proofs generated by certifying compilers can be independent of compiler implementation details. Thirdly, the effort needed to implement certifying infrastructure for a certifying compiler is less than the effort needed for conducting a compiler correctness proof. Realistic optimizing compilers are large software systems implementing sophisticated optimization algorithms. Certifying them by conducting machine-checkable compiler correctness proofs, which is highly desirable, is both challenging and time consuming. Therefore, in practise there exist only a small number of machine-checkable correctness proofs for realistic compilers. In the majority of cases, the compiler correctness proofs are conducted either to verify a selected aspect of the compiler algorithm, such as a compiler phase, or to present a proof of concept for a representative subset of the source language. In contrast, it was demonstrated in the literature that implementing a realistic certifying compiler is feasible [78, 75, 103, 104, 105, 29, 123, 122, 92, 7, 125, 124, 126, 77, 109, 110, 43, 41, 39, 20, 18]. Fourthly, the effort needed to maintain certifying infrastructure is less than the effort needed for for a certifying compiler than for a certified compiler as every change to the algorithm of a compiler requires redoing its correctness proof. On the contrary, the translation correctness proof generated by the certifying compiler are not sensitive to changes to the translation algorithm. This is due to the aforementioned fact that translation certificates make only statements about whether two programs are in a predefined translation relation which is defined independently of the compiler algorithm.

The certifying compiler approach has two major disadvantages: Firstly, users have to check the certificates for each (critical) compilation. This check may fail, if the compiler has a bug. Secondly,

verifying a realistic translation certificate by a third party program requires large amounts of time and memory resources and there is still a lot of research to be done in this realm to improve this. There are two reasons for this unsatisfying situation: the inefficiency of software tools used to implement the SVF's and the inefficiency of present proof techniques used in certifying compilers.

## 1.1 Approaches to certifying compiler

This section focuses on the certifying compiler approach and discusses existing approaches to certifying compilers. In the discussion, we consider the following aspects of each approach:

1. kind of property certified by the compiler,
2. certificate generation technique,
3. the size of trusting computing base (TCB) of the approach, i.e. which parts of the certifying infrastructure of the compiler has to be trusted,
4. the flexibility and modularity of the techniques used by the approach.

A more detailed discussion of the related work on both certified and certifying compilers will be given in Section 1.6.

Proof-carrying code (PCC) [78, 75] is a framework for guaranteeing that compiled programs meet certain safety requirements such as type safety or the absence of stack overflows. The framework works in the following scenario: The *code consumer* wants to assure himself that it is safe to execute a program supplied by an untrusted source. Hence, he defines a *safety policy*, i.e. a set of the safety requirements which have to be met by each program in order to be permitted for execution on the code consumer's machine, makes it public, and requires that the code supplied by the untrusted code producer must be accompanied by a machine checkable *safety proof* that attests to the adherence of the code to his safety policy. Before executing the code, the code consumer validates the proof (*proof validation*) without consulting any external agents.

A typical PCC system comprises four basic software components: a compiler, a verification condition generator (VCG), a proof assistant, and a proof checker. In a typical scenario, they are used as follows. Firstly, a source program is given as input to the compiler which translates it into a machine code, a target program. Secondly, the VCG incorporates a semantic framework for the representation of the machine code as a vector of verification conditions (VC vector). The target program is given as input to the VCG which automatically generates a representation of the target program as a VC vector (*semantic mapping step*). The generated VC vector serves as a *program abstraction* of the target program. Thirdly, the code producer uses a proof assistant to interactively conduct a safety proof that the VC vector is correct according to the safety policy of the consumer (*abstraction validation step* on the part of the code producer). Fourthly, the code producer delivers the target program together with the corresponding VC vector and the safety proof to the code consumer. Fifthly, as the semantic mapping step results in reducing the information about the original target program to a VC vector, which is, basically, a vector of first-order logic formulas, the code consumer does not trust that the delivered VC vector is indeed the output of the VCG started with the target program as input. Therefore, the consumer regenerates the VC vector using his own VCG, checks if the delivered VC vector is the same as the delivered one. Sixthly, if the delivered VC vector is the same as the delivered one, then it is passed together with the safety proof to a proof checker (*abstraction validation* and *proof validation* steps on the part of the code consumer). If the proof checker validates the proof, then the code consumer permits the target program to be executed on his machine.

The reliance of the code consumer that a delivered machine code can be safely executed on his machine is drawn from the following facts: Firstly, he trusts in the implementation correctness of

the VCG and in the correctness of semantical framework incorporated in the VCG, i.e. he trusts in the correctness of the semantic mapping performed by the VCG. Secondly, there exists a machine checked safety proof of a statement which is a function of a program abstraction delivered with the machine code, i.e. a VC vector, and says that the program abstraction meets the code consumer's safety policy.

Foundational proof-carrying code (FPCC) [5, 4] is a generalized version of PCC. In a conventional approach to proof-carrying code, the machine checkable proofs are written in a logic with built-in understanding of a particular type system, i.e. type constructors appear as primitives of the logic and the typing judgement is defined by a set of type inference rules which are built into the VCG component of the PCC system and the designers of the PCC verification system assume that these typing rules are valid. Due to this restriction, Appel and Felty, the authors of the FPCC, call the conventional PCC a *type-specialized PCC* as the type system has to be constructed for each PCC verification system separately. Unlike type-specialized PCC, the FPCC avoids using a VCG by formalizing the type system, the operational semantics of the machine code, and safety policies within a proof assistant using the higher order logic (HOL). In comparison with the PCC approach, the FPCC approach can be characterized as follows. Firstly, there is no semantic mapping step and no program abstraction in the FPCC in the sense that the FPCC system applies a *syntax-directed translation algorithm* to translate the target program into a *HOL constant* in a proof script. Secondly, the safety requirements for a machine code are formalized explicitly in the HOL as a safety predicate on the target program. Thirdly, certification is realized as generation of a proof of a lemma stating that this HOL constant fulfills the safety predicate.

As stated in [4], the FPCC approach has two advantages over the PCC approach: Firstly, it is more flexible as novel type systems or safety policies can be introduced to the code consumer without reimplementing of the type system or the safety policy incorporated in the VCG. Secondly, the size of the TCB associated with the FPCC is less the one associated with the PCC approach as the code consumer does not have to rely on the soundness of the typing rules encoded in the VCG and the correctness the VCG itself as the type safety proof is the part of the safety proof delivered by the code producer.

It is still possible that a FPCC system computes semantic mappings of some auxiliary data structures, possibly byproducts of data flow analysis, which are useful in the safety proof. Nevertheless, this does not change the fact that the original target program is not represented as program abstraction in the safety proof and that the code consumer can always check if the constant definition in the proof script matches the delivered target program by translating the program into a constant definition and comparing the result of translation with the constant definition in the proof script. Thus, the reliance of the code consumer that a delivered target program can be safely executed on his machine is drawn from the fact that there exists a machine checked proof script with the definition of a constant which corresponds to the program and a proof of a safety property with that constant as parameter.

The PCC and FPCC approaches developed are techniques for certifying various safety properties of *one* program associated with a compiler run, the target program. Now, we turn to approaches that deal with *both* the source *and* the target programs of a compiler run and certify that they are in a translation relationship.

The translation validation (TV) approach [103, 123, 122, 92] regards the compiler as a black box with at most minor instrumentation. In a typical implementation of this approach, each run of the compiler results in a source program, which served as input for the compiler, a target program, and in some implementations an auxiliary data, possibly byproduct of data flow analysis, revealing some details on how the translation was performed. The TV system considers these programs as state transition systems (STS) and defines them as semantically equivalent iff their STS representations are in a predefined relation. The common relations between the sets of STS's used in the literature

are the refinement and the bisimulation relations. Below, we use the refinement relation for the purpose of explanation. The programs that are resulting from a compiler run are passed to a separate program, a *translation validator*, which establishes that the STS representations of these programs are in refinement relation. If the validator succeeds, it generates a proof script with a proof of the statement saying that the STS representations of the source and the target programs are in refinement relation. Otherwise, the validator returns a counter example. In general, checking program equivalence is an undecidable problem. For this reason, we cannot hope to have a complete translation validator. However, equivalence checking is possible if the validator uses an auxiliary data delivered by the compiler. In the following, we explain a typical implementation of the validator:

Firstly, the code producer formalizes the notion of a refinement relation  $\sqsubseteq$  between two sets of state transition systems (STS). Secondly, the code producer formalizes a proof rule of the form  $TCC(STS_T, STS_S) \implies STS_T \sqsubseteq STS_S$  where  $TCC$  is a *translation correctness criterion* on two STS's which formulates necessary conditions which have to be discharged in order to prove that a target state transition system  $STS_T$  refines a source state transition system  $STS_S$ . Thirdly, he proves a *metatheorem* that this proof rule is valid. Fourthly, the code producer implements the components of the validator: a *semantic mapping unit* and an *abstraction validator*. The semantic mapping unit incorporates a semantic framework for the representation of the source and the target program as STS's in the programming language level. The abstraction validator is basically an analysing procedure  $TCC$  which is a programming language level representation of the translation correctness criterion  $TCC$ .

The validator takes the source and the target programs with auxiliary data as input and passes them to the semantic mapping unit which translates them into their programming language level representations as the source and the target STS's,  $STS_S$  and  $STS_T$ , respectively (*semantic mapping step*). The common auxiliary informations used in this step are data structures encoding a relation over corresponding program point pairs, a VC vector, and a refinement mapping between corresponding STS representations of states of execution. The generated STS's serve as abstractions of the source and the target programs (*program abstractions*). Then, the validator passes  $STS_S$  and  $STS_T$  to the abstraction validator which checks if the translation correctness criterion  $TCC(STS_T, STS_S)$  holds in the programming language level by calling  $TCC(STS_T, STS_S)$  (*abstraction validation step*). In principle, checking the predicate  $TCC(STS_T, STS_S)$  is based on performing symbolic executions on corresponding STS representations of states of executions and checking the successor states for the correspondency relation between the states which is defined by the auxiliary data provided by the compiler. If the procedure  $TCC$  establishes that  $STS_S$  and  $STS_T$  fulfill the  $TCC$  criterion, then it generates a proof script with a proof of  $TCC(STS_T, STS_S)$ . Otherwise, it generates a counter-example.

In a typical application scenario of the TV approach, the compiler user is also the code consumer. In this case, the certificate is the proof script generated by the translation validator. To gain confidence in the output of the validator, the compiler user can use a proof checker to validate the proof script (*proof checking step*). In a scenario with the compiler user not being the code consumer, the certificate gets similar to certificates issued by the PCC: The semantic mapping step results in a loss of the information about how the original program, which served as input for the semantic mapping, look like and the code consumer does not trust that the proof script delivered with the source and the target program is indeed the output of the translation validator program started with those programs as input. Therefore, the translation producer has to deliver the source and the target programs  $S$  and  $T$ , the auxiliary data, which was used during the semantic mapping step, the STS representations  $STS_S$  and  $STS_T$ , which were generated by the semantic mapping step, and the proof script with a proof of  $TCC(STS_T, STS_S)$ . Then, the code consumer regenerates the STS representations from the source and the target programs and the proof script using his own

translation validator and checks if the STS representations and the proof script generated by the translation validator are equal to the delivered ones and validates the proof using the proof checker (*proof checking step* on the part of the code consumer).

In the TV approach, the reliance of the code consumer that the delivered source and the target programs which are semantically equivalent is drawn from the following facts: Firstly, he trusts in the correctness of semantical framework incorporated in the semantic mapping unit, i.e. that semantic mapping from the source and the target programs to their STS was correct. Secondly, he trusts in the implementation correctness of the abstraction validator, in particular, in the correctness of the semantical framework incorporated in the abstraction validator's unit which is responsible for performing symbolic executions. Thus, the code consumer trusts that the existence of a machine checked proof script, which is the result of the successful call  $TCC(STS_T, STS_S)$ , implies  $STST$  refines  $STS_S$  in the programming language level.

The TV approach has the following disadvantages:

*Type-specialization issue:* In the TV, the proofs are written, as in the PCC, in a logic with built-in understanding of particular source and target type systems. For this reason the TV approach can also be called type-specialized in the same sense as the PCC approach. The type-specialization of the TV leads to a situation in which the translation validator functions like a black box with internal details, like incorporated type systems and their soundness or the incorporated semantics, which can not be examined by the code consumer and thus they are part of the TCB.

*Phase-specialization issue:* Let us consider again the source and the target programs  $S$  and  $T$ ; and the refinement relation  $\sqsubseteq$  between the sets of STS's, which we used above to explain the principle of the TV approach. As aforementioned, the TV approach regards these programs as STS's and translates them into their STS representations  $STS_S$  and  $STS_T$ , respectively, and shows that  $STS_T$  refines  $STS_S$ . Thus, the TV is suitable for certifying the results of a single compiler phase, provided that the code consumer accepts the size of TCB of the TV approach. However, the question arises whether the TV approach is suitable for developing a certification technique for whole chains of program transformations, which are successively performed by the compiler phases, which can be applied to generate a translation certificate attesting to the correctness of translation of the first program in the chain into the last program in the chain. As aforementioned in the beginning of this chapter, this would allow us to achieve the goal of carrying the results of the analysis and the verification of the source code over to the machine code level. To answer this question, consider a chain of three programs  $S, IL, T$ . The chain consists of a source program  $S$ , an intermediate program  $IL$ , and a target program  $T$ . Assume that the chain is the result of a compiler run with two phases and that all auxiliary informations, which are needed for generating the STS representations from  $S$ ,  $IL$ , and  $T$ , are available. Application of the TV technique to program pairs  $(S, IL)$  and  $(IL, T)$  results in four STS representations  $STS_S$ ,  $STS_{IL}$ ,  $STS'_{IL}$ , and  $STS_T$ ; and two proof scripts showing that  $STS_S$  refines  $STS_{IL}$  and that  $STS'_{IL}$  refines  $STS_T$ . Does it automatically holds that  $STS_S$  refines  $STS_{IL}$  and  $STS'_{IL}$  refines  $STS_T$  implies  $STS_S$  refines  $STS_T$ ? The answer is 'not in general'. For the case  $STS_{IL} = STS'_{IL}$ , we are able to prove an additional proof rule

$$STS_A \sqsubseteq STS_B \wedge STS_C \sqsubseteq STS_D \wedge STS_B = STS_C \implies STS_A \sqsubseteq STS_D$$

and to incorporate application of this rule in the translation validator. But, it is not possible to prove such a rule for the case  $STS_B \neq STS_C$ , which is common in the optimizer phase of a compiler. The proof rule

$$STS_A \sqsubseteq STS_B \wedge STS_C \sqsubseteq STS_D \wedge STS_B \neq STS_C \implies STS_A \sqsubseteq STS_D$$

can only be proved, if we show  $STS_B \sqsubseteq STS_C$  first. To show this, we need an additional general assumption that  $STS_B$  and  $STS_C$  fulfill the translation correctness condition  $TCC(STS_B, STS_C)$ .

But this can be shown automatically only in special cases of  $STS_B$  and  $STS_C$ . For this reason, we say that the conventional TV approach is *phase-specialized* as it requires introducing additional proof rules of the form

$$STS_A \sqsubseteq STS_B \wedge STS_C \sqsubseteq STS_D \wedge \mathcal{P}(STS_B, STS_C) \implies STS_A \sqsubseteq STS_D$$

where the predicate  $\mathcal{P}$  denotes additional assumptions about the pair  $(STS_B, STS_C)$ , if one wants to apply the transitivity argument in order to generate a translation certificate for the fact that  $STS_S$  refines  $STS_T$ . Introducing new proof rules to the abstraction validator is disadvantageous as it results in increasing the TCB size of a TV system.

## 1.2 Our approach

In this section, we describe our approach to certifying compilers, a *foundational translation validation* (FTV).

The FTV is a generalized version of the TV approach and it avoids the shortcomings of the TV in the same way as the FPCC approach does to avoid the shortcomings of the PCC. The FTV formalizes the source and the target languages within a proof assistant using the HOL logic. The formalizations of the languages include soundness proofs of their type systems and formalizations of their corresponding program semantics that are defined as mappings from programs to partially ordered sets of meanings of programs. The FTV formalizes an *explicit translation correctness predicate* which says that two programs, a source and a target programs, are semantically equivalent iff their program semantics are equal. We call the part of the SVF which includes the formalizations of the source and the target languages and the translation correctness predicate a *translation contract* between the code producer and the code consumer. The proof assistant, which was used to formalize the translation contract within the SVF, can also be used as proof checker. The certifying compiler comprises the instrumentation which is needed to generate proof scripts (which are actually HOL theory scripts) which serve as the translation certificates. After each compiler run translating a source program into a target program, the compiler generates a HOL theory which contains two definitions of HOL constants representing those programs and a proof that these constants fulfill the translation correctness predicate. We call such compiler a *proof generating compiler*.

The overall characterization of the FTV approach is as follows (cf. [94]):

Translation contract: We require an explicit *translation contract* which comprises

- formalization of source language  $SL$ ,
- formalization of target language  $TL$ ,
- formal definition of the translation correctness predicate  $\text{corrTrans}(S, T)$  specifying when a target program  $T$  is a correct translation of a source program  $S$ .

Each language formalization in the translation contract includes the formalizations of the underlying type system, the operational semantics, the program semantics, and the type safety proof. The definition of the translation correctness predicate  $\text{corrTrans}(S, T)$  says that a target program  $T$  is a correct translation of a source program  $S$  iff their program semantics are equal. The introduction of the program semantics for both languages is essential in our approach as it enables the compiler to apply the argument of transitivity in the correctness proofs. A translation contract serves as the specification of the proof task and plays the role of the contract between producer and client of the compiler and should be available to and comprehensible for the client. In particular, it can and should be independent of the structure and algorithms of the compilers satisfying the contract.

**Automatic certification and machine-checkability:** We require that the compiler generates proof scripts automatically and that all specifications and proofs are machine-checkable. The machine-checkability requirement is essential because of the complexity and size of the proof tasks.

**Independence of logic:** We require that all specifications and proofs are based on a formal general logic, that is, a logic that is independent of languages and techniques used in the translation. The logical independency is important in order to reduce the size of the TCB of our approach. Using a logical framework that is not specifically developed for the translation task and used in many other areas, increases the confidence in the framework. Of course, as argued in [94], a framework in which only a very small core has to be trusted is desirable.

**Program representations in proofs:** There is no semantic mapping step in the FTV in the sense that the compiler uses a *syntax-directed translation algorithm* to translate the source and the target programs into respective HOL constant definitions in a proof script.

**Certification technique:** Certification is realized as generation of a HOL theory proof script containing

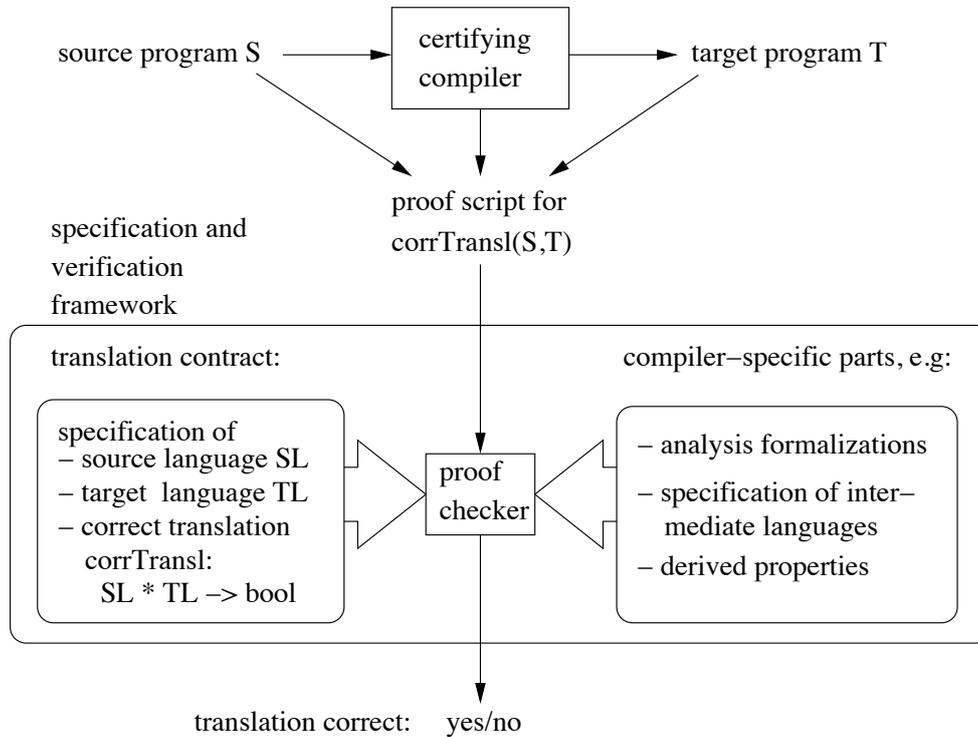
- definitions of constants representing the source and the target programs of a compiler run,
- a lemma stating that these constants fulfill the translation correctness predicate, and
- a proof of this lemma.

The FTV approach has the following advantages over the TV approach:

- It is more flexible as novel type systems or programming language semantics or translation correctness definitions can be introduced to the code consumer without reimplementing the semantic framework incorporated in the semantic mapping unit in the translation validator. This means that each translation contract defines a clear interface between the code producer and the code consumer.
- The size of the TCB associated with the FTV approach is smaller than the size of the TCB associated with the TV approach because the code consumer does not have to rely on the correctness of the translation validator itself and the correctness the semantic framework and the soundness of typing rules incorporated in the semantic mapping unit of a TV system. The programming language semantics is formalized directly in the SVF and the type safety proof is the part of the translation correctness proof delivered by the code producer. Similar to the FPCC approach, a FTV system computes semantic mappings of some auxiliary data structures, possibly byproducts of data flow analysis, which are useful in the translation correctness proof. Nevertheless, this does not change the fact that the representation of the original programs involved in the compiler run contain the whole information about those programs and that the code consumer can always check if the constant definitions in the proof script define delivered source and target programs by translating those programs into constant definitions and comparing the result of translation with the constant definitions in the proof script. In summary, the size of the TCB associated with the FTV is smaller than the size of the TCB associated with the TV approach because in the TV approach programs are represented in proofs by *semantic abstractions* and in the FTV approach programs are represented in proofs by their another *syntactic forms*.
- The FTV enables certification of transformation chains. As the program semantics maps programs to mathematical objects which are partially ordered, it is possible to check program semantics pairs for equality and to derive the correctness of a chain of program transformations by applying the transitivity rule.
- The FTV is more flexible when it comes to introducing new verification techniques. As described above, a TV system applies one verification technique which is implemented in its components. The abstraction validator applies symbolic execution technique to check if two

STS representations are in the refinement relation. Switching to other verification techniques would require to implement new abstraction validators or to employ other verification tools, which would have dramatic impact on the size of TCB associated with the TV system in question. In contrast to the TV, the FTV enables the code producer to freely extend the SVF by new formalizations which can be applied in proof scripts as long as he proves that they are correct by proving corresponding program independent lemmas.

Figure 1.1 shows an overview of the architecture of the FTV approach. The most important components of the architecture are a compiler itself and its accompanying SVF. The SVF comprises a translation contract, a proof checker and compiler-specific parts which belong to the verification infrastructure. The translation contract comprises formal specifications of a source language  $SL$ , a target language  $TL$ , and a binary predicate  $\text{corrTransl} : SL \times TL \rightarrow \mathbf{Bool}$  for source programs  $S$  and target programs  $T$ . The definition of  $\text{corrTransl}$  is based on formal semantics definitions of  $SL$  and  $TL$  and it precisely specifies when  $T$  is considered to be a correct translation of  $S$ . The compiler generates –in addition to the target  $T$ – a proof for  $\text{corrTransl}(S, T)$ . The proof is subject to validation by the proof checker.



**Fig. 1.1.** The architecture of the foundational translation validation approach

### 1.3 Objectives of this thesis

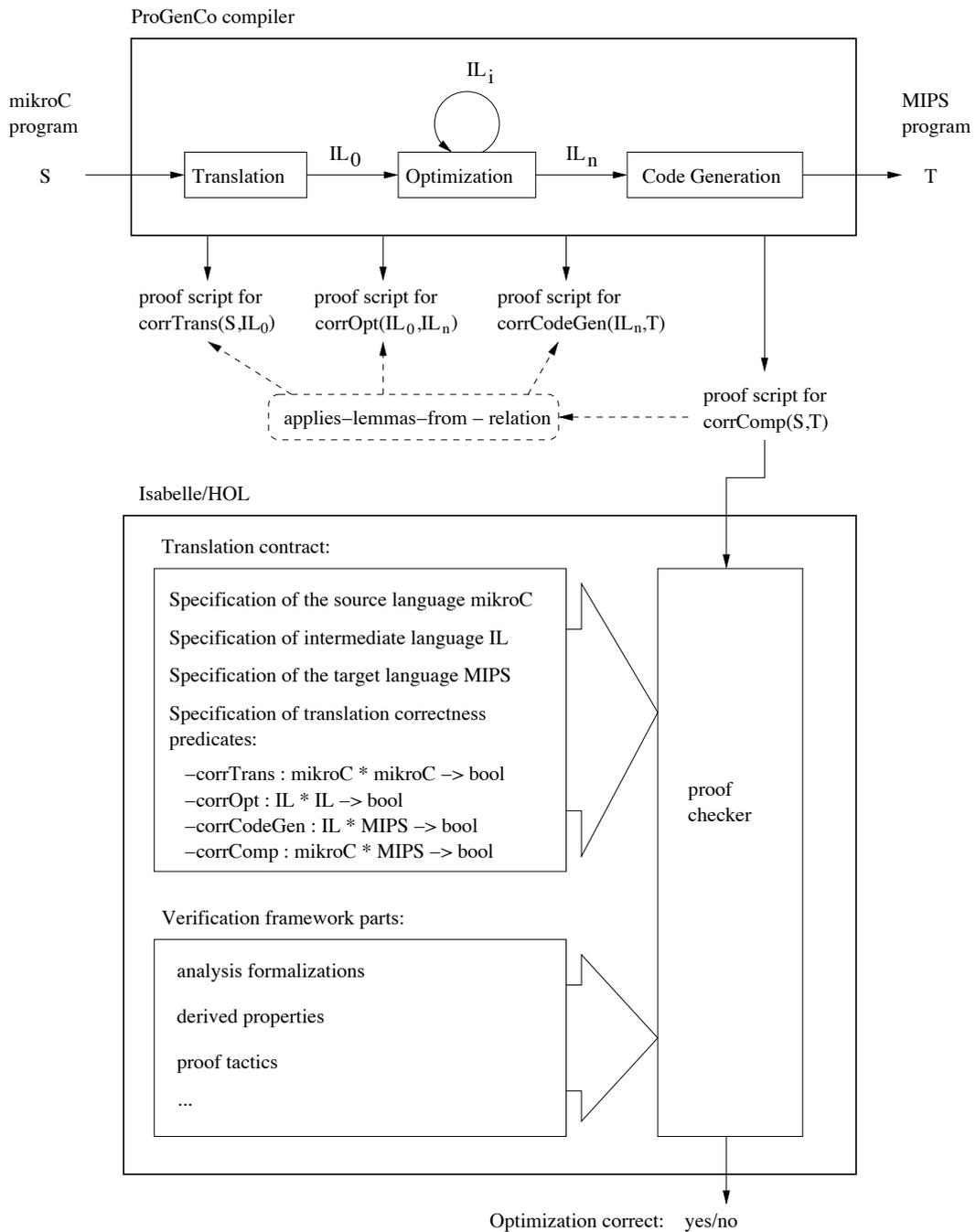
This thesis describes an implementation of a prototype FTV system for a small imperative language. The FTV system comprises an optimizing compiler front-end, which generates Isabelle/HOL theory files serving as proof scripts, and an accompanying SVF implemented in the Isabelle/HOL. As the compiler front-end applies standard translation and optimization algorithms

[1, 3, 119], we only provide a brief overview of its architecture and its work-flow and our demonstration focuses on the SVF in our FTV system. The objectives of our demonstration are as follows.

- Our main objective is to demonstrate that the FTV approach is a viable formal method for certification of whole chains of program transformations which are typically performed by the compiler and that implementation of an FTV system for an imperative programming language is feasible.
- We explain formalizations of concepts in our SVF which make the FTV approach distinguishable from the TV approach:
  - explicit formalization of the programming languages involved in the translation,
  - explicit formalization of translation correctness predicates,
  - type safety proofs,
  - explicit formalizations of the notions of translation relations which are independent of program transformations performed by the compiler,
  - explicit formalizations of the notions of translation relations which are specific to program transformations performed by the compiler (translation patterns),
  - proofs of translation correctness theorems, which can be seen as explicit versions of proof rules in the TV approach.
- The demonstration of our framework is meant to be a guideline for future developments of SVF's and research on certifying compilers following the FTV approach. In particular, the objective is to demonstrate:
  - how to formalize a translation contract for optimizations of programs written in a small imperative programming language,
  - how to formalize translation correctness criteria for individual program transformations performed by the compiler and how to verify their correctness by proving corresponding translation correctness theorems. A translation correctness theorem for a particular program transformation says that if the source and the target programs of this transformation fulfill a translation correctness criterion which is specific to this transformation, then these programs fulfill the translation correctness predicate specified in the translation contract. We exemplify this by describing the translation correctness criteria which we formalized for the optimizations performed by our compiler front-end,
  - how to implement a SVF as a stack of the translation correctness criteria and the respective translation correctness theorems such that
    - the formalization of the translation correctness predicate relating the source and the target programs of the compiler is placed at the bottom of the stack,
    - the formalizations of the translation correctness criteria relating the source and the target programs of the individual compiler phases are placed in the middle of the stack, and
    - the formalizations of the translation correctness criteria relation the source and the target programs of the individual program transformations performed by the compiler are placed atop of the stack.
  - how to specify correctness predicates for single optimization phases in a way that makes them composable and reusable in translation certificates.

## 1.4 The PROGENCo project

As mentioned in the previous section, this thesis reports on the implementation of a compiler front-end that follows the FTV approach to certifying compilers. The implementation of the compiler



**Fig. 1.2.** The architecture of the ProGenCo compiler

is a part of an ongoing work on the proof generating compiler project (PROGENCO project). This section describes the objectives of the project and the architecture of the PROGENCO compiler.

The aim of the project is to implement a prototype compiler that can be seen as an implementation of the FTV approach in the sense that it certifies its complete translation runs. Here, the certification of a complete translation run means that after each translation run the compiler generates a certificate that the translation of its input source program into a target program was correct regardless what compilation phases and optimizations are performed. As a result, the

PROGENCO compiler allows the compiler user to close the formalization chain from high-level methods to the machine-code level mentioned in the beginning of this chapter. In the following, we describe the architecture of the compiler.

Figure 1.2 shows the architecture of PROGENCO compiler. The PROGENCO compiler consists of two parts: the compiler itself and a corresponding SVF implemented within the proof assistant Isabelle/HOL. The front-end of the compiler uses a small C-like language as input, which is called  $\mu C$  in this thesis. The back-end comprises an optimization phase and code generation phase. The optimization phase uses an intermediate language IL as input and performs a number of optimizations. The code generation phase uses the IL language as input and generates MIPS code.

The SVF comprises two parts: a translation contract and a verification framework. The translation contract comprises formalizations of all three languages involved during translation runs of the compiler,  $\mu C$ , IL, and MIPS; and specifications of translation correctness predicates for all intermediate translations performed by the compiler and overall translations from  $\mu C$  into MIPS. Each language formalization comprises definition of program semantics and the definitions of the translation correctness predicates are based on the definitions of corresponding program semantics. The definitions of the translation correctness predicates  $\text{corrTrans}(S, IL_0)$ ,  $\text{corrOpt}(IL_0, IL_n)$ ,  $\text{corrCodeGen}(IL_n, T)$ , and  $\text{corrComp}(S, T)$ , are straightforward and formulated uniformly in the following sense: Each predicate says that a translation of a program in a source language into a program in a target language is correct iff their program semantics are equal.

The verification framework of the SVF is a proof environment for conducting translation correctness proofs for all translations performed by the compiler. It comprises auxiliary specifications, such as specifications of program analysis results or specifications of optimization patterns, program independent lemmas, and implementations of Isabelle/HOL proof tactics.

The aim of the proof environment's design is twofold: Firstly, the proof environment provide proofs of lemmas which allow for conducting proofs of translation correctness statements of the forms  $\text{corrTrans}(S, IL_0)$ ,  $\text{corrOpt}(IL_0, IL_n)$ ,  $\text{corrCodeGen}(IL_n, T)$ , and  $\text{corrComp}(S, T)$ . Secondly, the existence of proof tactics in the environment enables the compiler to generate theories with translation correctness proofs that are machine-checkable in batch mode. As the compiler knows the interface of the SVF, it incorporates appropriate proof tactic calls in the generated proof scripts which automatically prove all lemmas in the proof scripts. As a result, the compiler user can use the Isabelle/HOL proof assistant as a proof checker in order to verify the theories generated by the compiler.

The work-flow of the proof checking process is as follows. The translation certificate generated by the compiler is structured according to compiler phases. For each compiler phase, the compiler generates an Isabelle/HOL theory with the proof of the corresponding translation correctness predicate,  $\text{corrTrans}(S, IL_0)$  or  $\text{corrOpt}(IL_0, IL_n)$  or  $\text{corrCodeGen}(IL_n, T)$ , respectively. Further, the compiler generates a root theory that imports these theories and additionally contains a proof of the predicate  $\text{corrComp}(S, T)$  which specifies correctness of the complete translation run. All theories are given as input to the Isabelle/HOL proof assistant and the verification of the proof scripts proceeds as follows. First, the Isabelle/HOL reads import clauses in the root theory. The import clauses comprise the names of the theories that were generated for the individual compiler phases. The Isabelle/HOL reads those theories and verifies proofs of the translation correctness predicates in those theories. Second, the proof assistant verifies the proof of the predicate  $\text{corrComp}(S, T)$ . The proof of  $\text{corrComp}(S, T)$  is straightforward as it merely requires to show that program semantics of a source program  $S$  and a target program  $T$  are equal. The first step of the proof introduces three new assumptions which state that program transformations performed by the compiler phases were correct:  $\text{corrTrans}(S, IL_0)$ ,  $\text{corrOpt}(IL_0, IL_n)$ , and  $\text{corrCodeGen}(IL_n, T)$ . The second step of

the proof derives the predicate  $\text{corrComp}(S, T)$  by unfolding the definitions of the predicates  $\text{corrTrans}$ ,  $\text{corrOpt}$ ,  $\text{corrCodeGen}$ , and  $\text{corrComp}$  and applying the transitivity rule.

Both generation of proof scripts by the compiler and verifying them by the Isabelle/HOL are completely automated.

## 1.5 Overview of our implementation

As mentioned in Section 1.3, we implemented the most challenging part of the PROGENCO compiler, a certifying optimizer and its accompanying SVF. This section gives a brief overview of our implementation.

Figure 1.3 shows the architecture of our compiler front-end. Our implementation comprises two parts: an optimizing compiler front-end and an accompanying SVF formalized in Isabelle/HOL. The front-end has two phases: a translation phase translating  $\mu C$  programs into IL programs and an optimization phase. The front-end takes a  $\mu C$  program  $S$  as input, translates it into an IL program  $IL_0$ , passes  $IL_0$  to an optimization phase which performs several optimizations and returns an optimized IL program  $IL_n$ . Subsequently, the front-end generates an Isabelle/HOL theory, which serves as a translation certificate. The theory contains a proof that  $IL_n$  is a correct optimization of  $IL_0$  according to the definition of a translation correctness predicate  $\text{corrTrans}$  specified in the translation contract part of the SVF. The Isabelle/HOL theory can then be given as input to the theorem prover Isabelle/HOL which serves as a proof checker and verifies the proof of the statement  $\text{corrTrans}(IL_0, IL_n)$  in that theory.

The rest of this section is organized as follows. The IL language and the work-flow of our compiler front-end is described in Section 1.5.1. Section 1.5.2 gives a brief overview of our SVF.

### 1.5.1 Compiler front-end

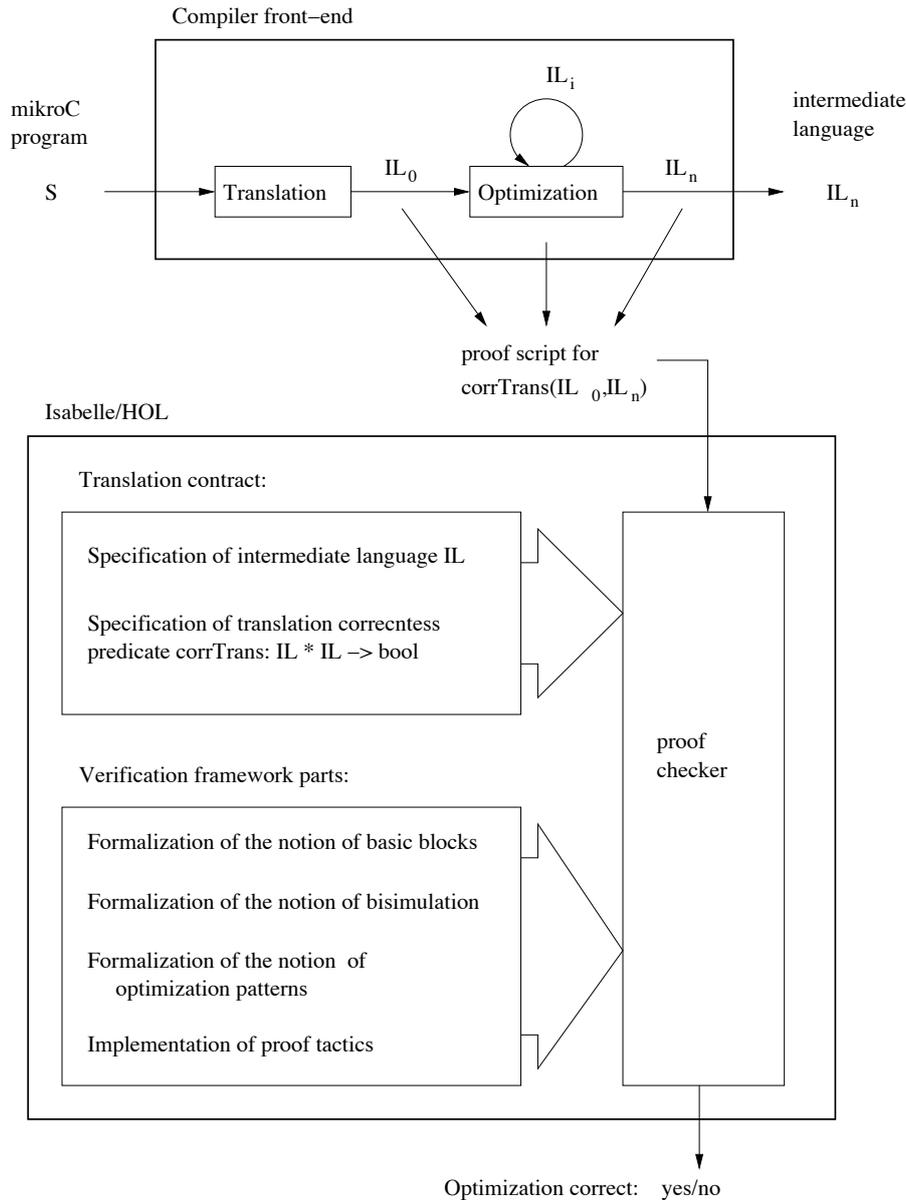
The main features of the language IL are

- a simple type system comprising four types (integer, boolean, integer array, and boolean array),
- an instruction set comprising five instructions (assignment, integers printing, branch, goto, and exit),
- an array-index-out-of-bounds exception,
- program semantics of IL programs is explicitly defined in terms of their behavior observable to the outside world, where the observable behavior of a program is determined by a sequence of integer printed by that program and whether it terminates or not.

Figure 1.4 gives an example of a  $\mu C$  program and its translation into the language IL.

Figure 1.5 gives an overview of the work-flow of our front-end. The front-end takes a  $\mu C$  program  $S$  as input. If  $S$  is well-formed, it is translated into an IL program  $IL_0$ . The translation algorithm is standard [1, 3]. Subsequently, the program  $IL_0$  is passed to the optimization phase which performs the following three optimizations successively: constant folding ( $CF$ ), dead assignment elimination ( $DAE$ ), and loop invariant hoisting ( $LIH$ ). The implementations of these optimizations are standard [3, 1]. However, an unusual and additional feature of our implementation of the  $LIH$  optimization is that it delivers three programs that are results of three intermediate transformations modifying programs locally only, i.e. in the same way as the optimizations  $CF$  and  $DAE$  do. These transformations are the following: nop insertion ( $NI$ ), redundant assignment insertion ( $RAI$ ), and redundant assignment elimination ( $RAE$ ).

- The  $NI$  transformation takes integers  $n$  and  $pc$  as input and inserts a list of  $n$  nop instructions between the  $(pc - 1)$ -th and  $pc$ -th instructions of a program. As the nop instruction is not in



**Fig. 1.3.** The architecture of our certifying compiler front-end

the instruction set of the language IL, we emulate it using goto instructions with appropriately adjusted values of jump destinations. In our example program in Figure 1.4, the compiler detects that the nop instruction from the line [11], which is emulated by the goto instruction [11] goto [12], is a head of the loop and that the lines [12] and [13] contain loop invariant code and the assignment from the line [12] can be moved out of the loop and that this move can be certified using our framework. Therefore, the compiler performs nop insertion transformation with parameters  $n$  and  $pc$  set to the values 1 and 11, respectively. The inserted nop instruction in the line [11] plays the role of a placeholder for the next transformations.

- The *RAI* transformation takes an integer  $pc$  and an assignment instruction  $x := e$ ; as input and replaces a nop instruction at  $pc$ -th program point by  $x := e$ . We call this transformation a redundant assignment insertion because it is used to insert an assignment  $x := e$ ; only at

```

sum(int n, int a []) {
  int i, tmp, res;

  i=0;
  res=0;

  while(n < 2){
    n=n+1;
  }

  do{
    tmp = n*n;
    res = res + tmp + a[i];
    printi (tmp + a[i]);
    i = i + 1;
  } while(i < length(a));

  printi res;
}

main(){
  int b[4];
  int x;

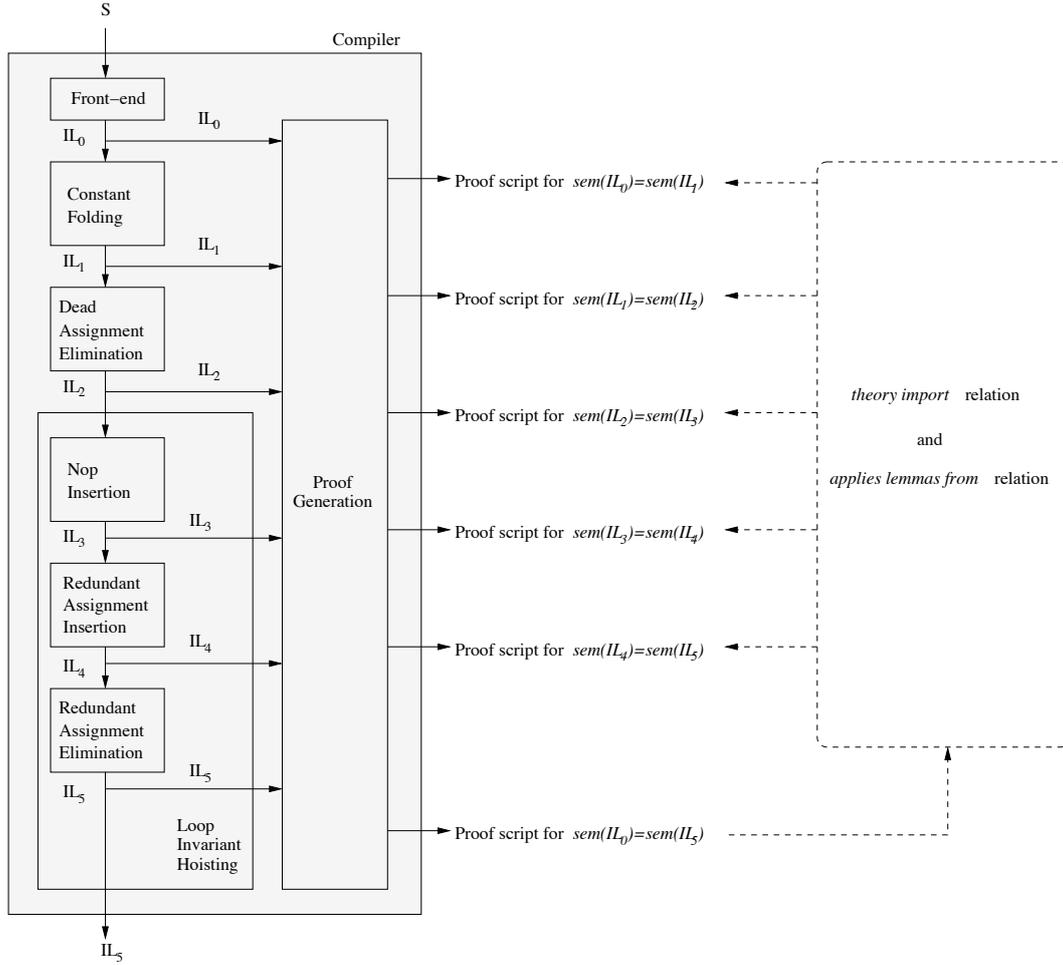
  x=1;
  b[0]=9;
  b[1]=7;
  b[2]=5;
  sum(x,b);
}

vardecls
  n int; a int [4]; i int;
  tmp int; res int; _tI_1 int;
  _tI_2 int; _tI_3 int; _tB_1 bool;
  _tI_4 int; _tI_5 int; _tI_6 int;
  _tI_7 int; _tI_8 int; _tI_9 int;
  _tI_10 int; _tI_11 int; _tI_12 int;
  _tI_13 int; _tI_14 int; _tB_2 bool;
begin
  [0]: _tI_1 = 0;
  [1]: i = _tI_1;
  [2]: _tI_2 = 0;
  [3]: res = _tI_2;
  [4]: _tI_3 = 2;
  [5]: _tB_1 = n < _tI_3;
  [6]: BRANCH ~_tB_1 [11];
  [7]: _tI_4 = 1;
  [8]: _tI_5 = n + _tI_4;
  [9]: n = _tI_5;
  [10]: GOTO [4];
  [11]: GOTO [12];
  [12]: _tI_6 = n * n;
  [13]: tmp = _tI_6;
  [14]: _tI_7 = res + tmp;
  [15]: _tI_8 = a[i];
  [16]: _tI_9 = _tI_7 + _tI_8;
  [17]: res = _tI_9;
  [18]: _tI_10 = a[i];
  [19]: _tI_11 = tmp + _tI_10;
  [20]: PRINTI _tI_11;
  [21]: _tI_12 = 1;
  [22]: _tI_13 = i + _tI_12;
  [23]: i = _tI_13;
  [24]: _tI_14 = 4;
  [25]: _tB_2 = i < _tI_14;
  [26]: BRANCH _tB_2 [11];
  [27]: PRINTI res [28];
  [28]: EXIT;
end

INPUT begin:
(n, 1), (a, [9, 7, 5, 0]),
(i, 0), (tmp, 0),
(res, 0), (_tI_1, 0),
(_tI_2, 0), (_tI_3, 0),
(_tB_1, false), (_tI_4, 0),
(_tI_5, 0), (_tI_6, 0),
(_tI_7, 0), (_tI_8, 0),
(_tI_9, 0), (_tI_10, 0),
(_tI_11, 0), (_tI_12, 0),
(_tI_13, 0), (_tI_14, 0),
(_tB_2, false)
INPUT end

```

Fig. 1.4. An example of an  $\mu C$  program and its translation into the language IL



**Fig. 1.5.** The work-flow of our compiler front-end

a program point  $pc$  at which it is made dead by an equal assignment instruction at another program point  $pc'$ . In our example program in Figure 1.4, the compiler performs the *RAI* transformation with parameters  $x := e$ ; and  $pc$  set to `_tI_6 = n * n`; and 11, respectively.

- The *RAE* transformation takes an integer  $pc$  as input and replaces an assignment  $x := e$ ; at  $pc$ -th program point by a nop instruction. We call this transformation a redundant assignment elimination because it is used to remove an assignment  $x := e$ ; at a program point  $pc$  at which it makes dead an equal assignment instruction at another program point  $pc'$ . In our example program in Figure 1.4, the compiler performs the *RAE* transformation with parameters  $x := e$ ; and  $pc$  set to `_tI_6 = n * n`; and 12, respectively.

As a result of performing the above program transformations *NI*, *RAI*, and *RAE* successively, the 11-th instruction is hoisted out of the loop whose head is the program point 11.

Figures A.1, A.2, A.3, A.4, and A.5 in Appendix A depict program pairs  $(IL_0, IL_1)$ ,  $(IL_1, IL_2)$ ,  $(IL_2, IL_3)$ ,  $(IL_3, IL_4)$ , and  $(IL_4, IL_5)$ , which are the continuation of the example depicted in Figure 1.4, i.e. the program  $IL_0$  in Figure 1.4 is the same as program  $IL_0$  in Figure A.1. Figure 1.6 depicts the program  $IL_0$  and the output of the optimization phase of our compiler, a program  $IL_5$ .

The intermediate optimizations *CF*, *DAE*, *NI*, *RAI*, and *RAE*, produce five intermediate programs:  $IL_1$ ,  $IL_2$ ,  $IL_3$ ,  $IL_4$ , and  $IL_5$ , respectively. For each intermediate optimization with a source program  $IL_i$  and a target program  $IL_{i+1}$ , a pair  $(IL_i, IL_{i+1})$  is build and passed to the proof gener-

<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: _tI_1 = 0; [1]: i = _tI_1; [2]: _tI_2 = 0; [3]: res = _tI_2; [4]: _tI_3 = 2; [5]: _tB_1 = n &lt; _tI_3; [6]: BRANCH ~_tB_1 [11]; [7]: _tI_4 = 1; [8]: _tI_5 = n + _tI_4; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: _tI_6 = n * n; [13]: tmp = _tI_6; [14]: _tI_7 = res + tmp; [15]: _tI_8 = a[i]; [16]: _tI_9 = _tI_7 + _tI_8; [17]: res = _tI_9; [18]: _tI_10 = a[i]; [19]: _tI_11 = tmp + _tI_10; [20]: PRINTI _tI_11; [21]: _tI_12 = 1; [22]: _tI_13 = i + _tI_12; [23]: i = _tI_13; [24]: _tI_14 = 4; [25]: _tB_2 = i &lt; _tI_14; [26]: BRANCH _tB_2 [11]; [27]: PRINTI res [28]; [28]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>	<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: _tI_6 = n * n; [12]: GOTO [13]; [13]: GOTO [14]; [14]: tmp = _tI_6; [15]: _tI_7 = res + tmp; [16]: _tI_8 = a[i]; [17]: _tI_9 = _tI_7 + _tI_8; [18]: res = _tI_9; [19]: _tI_10 = a[i]; [20]: _tI_11 = tmp + _tI_10; [21]: PRINTI _tI_11; [22]: GOTO [23]; [23]: _tI_13 = i + 1; [24]: i = _tI_13; [25]: GOTO [26]; [26]: _tB_2 = i &lt; 4; [27]: BRANCH _tB_2 [12]; [28]: PRINTI res [29]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>
---	--

**Fig. 1.6.** The programs  $IL_0$ , left, and  $IL_5$ , right, which are the source and the target programs of the optimization phase of our compiler.

ation unit of the compiler. For each such pair, the proof generation unit generates an Isabelle/HOL theory file containing a proof of the statement  $\text{corrTrans}(IL_i, IL_{i+1})$ . After all intermediate optimizations are completed and the respective proof scripts are generated, the proof generating unit generates a root proof script which contains a proof of the statement  $\text{corrTrans}(IL_0, IL_5)$ . Theory files can be either inspected by the human and verified interactively using the theorem prover Isabelle/HOL or verified by the Isabelle/HOL in batch mode.

### 1.5.2 Specification and verification framework

The main field of our work was developing the SVF and its underlying concepts and proof techniques within the theorem prover Isabelle/HOL.

In the following, we provide a brief overview of our SVF. The specification part of the SVF comprises the specification of a translation contract for optimizations of IL programs. The verification part of SVF comprises auxiliary specifications, lemmas, and Isabelle/HOL tactics which make up a proof environment for proving IL optimizations correct. The architecture of our SVF is organized hierarchically in layers. In the following, we describe briefly the architecture of the SVF.

Figure 1.7 gives an overview of the architecture of our SVF. The bottom layer comprises the most general formalizations. The top layer comprises the most specific formalizations which are dedicated to particular proof tasks which can be performed using our SVF. Formalizations from upper layers use definitions and apply lemmas from lower layers. The idea behind this architecture is that our SVF provides a set of translation correctness criteria defining semantic equivalence of intermediate programs on different levels of abstraction. The translation contract in our SVF provides the most abstract criterion which compares observable behaviors of two intermediate programs. The most specific criteria in our SVF are defined for particular optimizations and are based on the notions of control flow graphs with blocks (CFGB), and translation relations over pairs of CFGB's (also called optimization patterns). As these criteria are based on different concepts, there is a significant abstraction gap between them. This makes a task of proving that satisfying an optimization specific criterion by two CFGB's declared for two programs  $S$  and  $T$  implies equality of observable behaviors of  $S$  and  $T$  hard. To be able to conduct proofs of such implications, we introduced an intermediate translation correctness criterion which closes the abstraction gap and is based on the notions of bisimulation relation and bisimulation of partial executions of programs. Further, we arranged all criteria and theorems provided by our SVF in a stack of criteria and theorems such that more specific criteria are on top of more abstract criteria and for each two translation correctness criteria  $TCC_1$  and  $TCC_2$  provided by our SVF such that  $TCC_2$  is on top of  $TCC_1$  there is a theorem stating that if two programs fulfill  $TCC_2$ , they fulfill  $TCC_1$ . We anticipate that future FTV systems will have their SVF's structured in the same way. Therefore, we summarize the above discussion by giving a description of arrangement for a set of translation correctness criteria for the whole compiler:

- the formalization of the translation correctness predicate relating the source and the target programs of the compiler is placed at the bottom of the stack,
- the formalizations of the translation correctness criteria relating the source and the target programs of the individual compiler phases are placed in the middle of the stack, and
- the formalizations of the translation correctness criteria relation the source and the target programs of the individual program transformations performed by the compiler are placed atop of the stack.

We start explaining the bottom layer of the architecture and work through to the top layer.

Layer 6:	proof environments specific to particular proof tasks
Layer 5:	translation correctness criteria specific to particular optimizations
Layer 4:	optimization independent translation correctness criterion
Layer 3:	IL type safety proof
Layer 2:	translation contract for IL optimizations
Layer 1:	auxiliary extensions of HOL
Layer 0:	implementation of HOL within the proof assistant Isabelle/HOL

**Fig. 1.7.** The layer architecture of our SVF

*Layer 0: Logic layer.* The logic layer provides an implementation of higher-order logic on which the implementation of a particular SVF is based. Our implementation of the SVF is based on the implementation of HOL provided by the theorem prover Isabelle/HOL in the theory `HOL`.

*Layer 1: Logic extension layer.* The logic extension layer provides auxiliary generic formalizations which are both programming language independent and optimization independent. Our implementation of the SVF applies the formalisms provided by the Isabelle/HOL in the theory `Main` and specifies additional auxiliary definitions involving sets, functions, lists, etc, and proves lemmas about them.

*Layer 2: Translation contract layer.* Layer 2 provides the specification of our translation contract. As our work deals with intermediate language optimizations, we formalized a special case of the translation contract in which the source language  $SL$  and the target language  $TL$  are equal. The translation contract consists of the following formalizations:

- The formalization of an intermediate language IL which comprises the following:
  - The abstract syntax of the language IL is defined by giving formation rules for a syntactic set

$$S, T \in \mathbf{Program}$$

- The operational semantics of the IL language is formalized in terms of configurations

$\sigma \in \mathbf{Configuration}$

and a transition function

$\mathbf{exec} : \mathbf{Program} \times \mathbf{Configuration} \rightarrow \mathbf{Configuration}$

where a configuration  $\sigma$  models a state of execution of an IL program and  $\mathbf{exec}$  computes the successor configuration for an IL program and a configuration.

- The program semantics of the language IL is defined formalizing the notion of observable behavior of IL programs, which is done by giving formation rules for a set  $\mathbf{ObservableBehavior}$ , and by giving the definition of a function

$\mathbf{Sem} : \mathbf{Program} \rightarrow \mathbf{ObservableBehavior}$

which maps IL programs to their observable behaviors.

- The definition of a translation correctness predicate  $\mathbf{corrTrans} : \mathbf{Program} \times \mathbf{Program} \rightarrow \mathbf{Bool}$ . The definition of  $\mathbf{corrTrans}$  is straightforward:

$$\mathbf{corrTrans}(S, T) = (\mathbf{Sem}(S) = \mathbf{Sem}(T))$$

Informally, the definition of  $\mathbf{corrTrans}$  says that two IL programs  $S$  and  $T$  are semantically equivalent iff they have the same observable behaviors.

Summing up, Layer 2 provides the most general translation correctness criterion, the predicate  $\mathbf{corrTrans}$ , which is used in our framework. In our translation contract, the predicate  $\mathbf{corrTrans}$  is used to check the correctness of the optimization phase of our compiler front-end by relating the observable behavior of the source program of the phase to the observable behavior of the target program of the phase. As the optimization phase performs a chain of five program transformations, which results in producing of a chain of five IL programs, this means that the predicate  $\mathbf{corrTrans}$  can be used to relate the observable behavior of the first program of this chain to the observable behavior of the last program from this chain. The  $\mathbf{corrTrans}$  constitutes the bottom element of the aforementioned stack of translation correctness criteria.

*Layer 3: Type safety layer.* Layer 3 provides formalizations of the notions of program type and well-typedness; and a proof that the IL language is type safe. Each translation certificate generated for a source and a target program by our compiler contains a proof that these programs are well-typed w.r.t. their respective program types. Type safety, which expresses a typing invariant of program executions, is a necessary prerequisite when implementing a SVF following the FTV approach. The necessity of proving type safety comes from the fact that in the FTV approach one often reasons inductively about executions of programs and the type safety property of a language allows introducing deciding assumptions which makes proving the inductive steps in that reasoning possible.

*Layer 4: Optimization independent translation correctness criterion layer.* The purpose of Layer 4 is to provide

- the formalization of an optimization independent translation correctness criterion TCC on two IL programs which formulates a property which is independent of optimizations performed by our compiler front-end, bisimulation of two executions of programs.
- a proof of a translation correctness theorem saying the following:
 

*If two IL programs  $S$  and  $T$  fulfill the translation correctness criterion TCC, then  $\mathbf{corrTrans}(S, T)$  holds true.*

The optimization independent translation correctness criterion TCC is an intermediate criterion in our aforementioned stack of criteria. The idea behind this criterion and the translation correctness theorem will be explained later on when describing the content of Layer 5.

Our formalization of the TCC criterion is based on the following formalizations:

1. the formalization of the notion bisimulation relation over pairs of configurations  $\mathcal{R}$ ,
2. the formalization of the notion of a declaration of a control flow graph with blocks (CFGB) for an IL program,
3. the formalization of the notion of well-formedness of a CFGB declaration w.r.t. an IL program,
4. the formalization of an intermediate language of control flow graphs with blocks IL' whose programs are tuples  $(P, B)$  consisting of an IL program  $P$  and a CFGB declaration  $B$ ; the operational semantics is defined in terms of transfers of the flow of control from block position to block position, and the denotational semantics of programs in this language is defined in terms of their observable behaviors in the set **ObservableBehavior**,
5. the formalization of an intermediate language of control flow graphs with blocks IL'' whose programs are tuples  $(P, B)$  as in the IL' language and the operational semantics is defined in terms of block-wise transfers of the flow of control. The definition of the control flow transfers from block to block is based on the definition of control flow transfers from block position to block position in the IL' language. The denotational semantics of IL'' programs is defined in terms of their observable behaviors in the set **ObservableBehavior**,
6. the formalization of a predicate **bisimulation** on two IL'' programs  $(S, B_S)$  and  $(T, B_T)$  and a bisimulation relation  $\mathcal{R}$  which checks if block-wise executions of two IL'' programs  $(S, B_S)$  and  $(T, B_T)$  bisimulate w.r.t.  $\mathcal{R}$ .

Based on these formalizations, our optimization independent translation correctness criterion TCC on a source language program  $S$ , a target language program  $T$ , and bisimulation relation  $\mathcal{R}$  is formulated as a statement of the following form:

*There exist  $B_S$  and  $B_T$  such that the IL'' programs  $(S, B_S)$  and  $(T, B_T)$  fulfill the predicate **bisimulation** w.r.t. the bisimulation relation  $\mathcal{R}$ .*

*Layer 5: Optimization specific correctness criteria layer.* In general, the purpose of Layer 5 is to provide, for each program transformation  $\top$  performed by the compiler, the specification and the verification of a translation correctness criterion  $\text{TCC}_{\top}$  on the source and the target language programs which is specific to that transformation. The specification of  $\text{TCC}_{\top}$  has to fulfill the following requirements:

1.  $\text{TCC}_{\top}$  formulates a translation correctness property which is specific to the transformation  $\top$ . In particular,  $\text{TCC}_{\top}$  has additional, auxiliary parameters such as CFGB declarations and a result of data flow analysis specific to the transformation  $\top$ .
2. The formulation of  $\text{TCC}_{\top}$  enables automatic checking if two programs fulfill that criterion.
3. It is possible to prove a program independent theorem saying that if a source and a target language programs fulfill the criterion  $\text{TCC}_{\top}$ , then they fulfill the transformation specific transformation independent translation correctness criterion TCC.

The verification part of Layer 5 provides the following for each transformation  $\top$ :

1. A proof of a translation correctness theorem saying that
 

*If a source program  $S$ , a target program  $T$ , and other parameters specific to the transformation  $\top$  fulfill the criterion  $\text{TCC}_{\top}$ , then  $S$  and  $T$  fulfill transformation independent translation correctness criterion TCC.*
2. A corollary which is the result of conjoining of the above theorem and the translation correctness provided by Layer 4:
 

*If a source program  $S$ , a target program  $T$ , and other parameters specific to the transformation  $\top$  fulfill the criterion  $\text{TCC}_{\top}$ , then  $\text{corrTrans}(S, T)$  holds true.*

Our implementation of Layer 5 provides the following for each intermediate optimization  $O$  performed by our compiler front-end:

1. The formalization of the notion of a result of data flow analysis  $\mathcal{A}_O$  which is specific to the optimization  $O$  and is always performed by our front-end prior to  $O$ ,
2. The formalization of the bisimulation relation  $\mathcal{R}_O$  as a function of two IL” programs and a data flow analysis result  $\mathcal{A}_O$ .
3. The formalization of a translation relation  $\text{transrel}_O$  on two IL” programs  $(S, B_S)$  and  $(T, B_T)$  and a result of data flow analysis  $\mathcal{A}_O$  which is specific to the optimization  $O$ .
4. The formalization of an optimization correctness criterion  $\text{TCC}_O$  on two IL” programs  $(S, B_S)$  and  $(T, B_T)$  and a result of data flow analysis  $\mathcal{A}_O$  which checks if  $(S, B_S)$ ,  $(T, B_T)$ , and  $\mathcal{A}_O$  are in a translation relation  $\text{transrel}_O$ .
5. A proof of an optimization correctness theorem about two IL” programs  $(S, B_S)$  and  $(T, B_T)$ , a data flow analysis  $\mathcal{A}_O$ , and a bisimulation relation  $\mathcal{R}_O$  which says that
 

*If  $S$  and  $T$  are well-typed and  $(S, B_S)$  and  $(T, B_T)$  are well-formed and  $(S, B_S)$ ,  $(T, B_T)$ , and  $\mathcal{A}_O$  fulfill optimization correctness criterion  $\text{TCC}_O$ , then  $(S, B_S)$  and  $(T, B_T)$  fulfill the optimization independent criterion  $\text{TCC}$  w.r.t.  $\mathcal{R}_O$ .*

The optimization specific criterion  $\text{TCC}_O$  is placed atop of our stack of correctness criteria and the purpose of the intermediate criterion  $\text{TCC}$  becomes clear: For each optimization correctness criterion  $\text{TCC}_O$  and a corresponding optimization correctness theorem, our implementation of Layer 5 provides a proof of a corollary which is the result of conjoining of the optimization correctness theorem and the translation correctness theorem provided by Layer 4 which says the following:

*If  $S$  and  $T$  are well-typed and  $(S, B_S)$  and  $(T, B_T)$  are well-formed and  $(S, B_S)$ ,  $(T, B_T)$ , and  $\mathcal{A}_O$  fulfill optimization correctness criterion  $\text{TCC}_O$ , then  $S$  and  $T$  fulfill the translation correctness predicate  $\text{corrTrans}$ .*

The above corollary is directly applicable in proof scripts generated by the compiler. We explain this by way of example:

*Example 1.1.* Let us assume that the optimization  $O$  transforms an IL program  $S$  in a program  $T$  and that our SVF provides proof tactics allowing for proving well-typedness of an IL program, well-formedness of a CFGB declaration w.r.t to IL program, and if two IL” programs and a data flow analysis result fulfill the predicate  $\text{transrel}_O$ . Then, generating an Isabelle/HOL theory with a proof that an IL program  $T$  is a correct  $O$  optimization of  $S$ , is straightforward. The compiler merely has to generate an Isabelle/HOL theory with

1. constant definitions of  $S$ ,  $T$ ,
2. constant definition of CFGB declarations  $B_S$ ,  $B_T$ ,
3. constant definition of the data flow analysis result  $\mathcal{A}_O$ ,
4. proofs that programs  $S$  and  $T$  are well-typed,
5. proofs that the CFGB declarations  $B_S$  and  $B_T$  are well-formed w.r.t.  $S$  and  $T$ , respectively,
6. proof that  $(S, B_S)$ ,  $(T, B_T)$  and  $\mathcal{A}_O$  fulfill the predicate  $\text{TCC}_O$ ,
7. proof of the translation correctness predicate  $\text{corrTrans}(S, T)$  which, in the first step, applies the corollary provided by Layer 5 for the optimization  $O$  and in the second step discharges its assumptions by application of lemmas proved in 4., 5., and 6..

◇

*Layer 6: Proof environments for particular proof tasks.* Layer 6 provides auxiliary lemmas and proof tactics which are directly applied in proof scripts generated by our compiler in order to prove concrete lemmas about IL programs.

At the end of the description of Layer 5, we give an example which demonstrated how the corollaries about the optimization specific criteria (or, in general, about the program transformation specific criteria) can be directly applied in proof scripts generated by the compiler. Nevertheless, it should be noted that this example also demonstrates that a corollary which is specific to an optimization  $O$  determines the layout of proof scripts which are generated by the compiler in order to certify the optimization  $O$ . Each assumption  $A_i$  of a criterion specific to an optimization  $O$  induces a section in a proof script, which is generated as a translation certificate for the correctness of a concrete optimization  $O$ , which provides a proof of a lemma whose statement is  $A_i$  with appropriately instantiated parameter. Each section providing the proof of  $A_i$  must appear in the proof script before the section in which the corollary specific to the optimization  $O$  is applied. In our proof scripts, the section applying the corollary specific to an optimization is always the last one in the proof script.

In our SVF, we call the statements  $A_i$  with the instantiated parameters and the translation correctness predicates `corrTrans` with the parameters  $S$  and  $T$  instantiated with constants representing IL programs proof tasks.

The example from Layer 5 demonstrates what proof tasks has to be performed by the theorem prover in order to verify a proof script:

1. proofs of well-typedness of IL programs,
2. proofs of well-formedness of CFGB declarations,
3. proofs of optimization specific correctness criteria, and
4. proofs of the translation correctness predicates.

The proof tactics for 1. and 2. are uniformly defined for all optimizations performed by our compiler. The proof tactics for 3. and 4. are specific to optimizations which they are implemented for.

The content of this layer is the most technical one in our SVF as the implementation of the proof tactics provided by this layer uses the programming interface of the theorem prover Isabelle/HOL and the lemmas proved in this layer do not extend the generic framework set up in the lower layers but are merely used by the proof tactics implemented in this layer.

As future implementations of this layer in FTV systems will vary depending of the theorem prover used to implement the SVF and the content of the SVF itself, explaining the implementation of proof tactics and the lemmas is beyond of the scope of this thesis.

Finally, it should be noted that the content of this layer does not increases the size of the TCB in our FTV system. If a proof tactic has a bug, then a call to this tactic can merely result in a false alarm, i.e. in a situation that the theorem prover Isabelle/HOL which acts as a proof checker rejects that tactic due to its inability to prove a statement of interest although it is provably valid.

## 1.6 Related work

For more than thirty years researchers have worked on the problem of compiler correctness. As the problem of compiler correctness is an instance of a general problem of gaining confidence in the output of programs, we differentiate after Blum and Kannan [22], between the following main approaches to achieve this:

- The program verification approach [27] seeks to achieve the confidence in the output of programs by proving that a program is correct. The seminal papers on formal reasoning about program correctness were published by Floyd in [37] and Hoare in [53]. The first compiler correctness proof was published by McCarthy and Painter in [71]. The research on certified compilers follows the program verification approach.

- Program testing method [95] runs programs on test inputs for which the outputs are known and checks if the generated output matches the expected output.
- Program checking uses a program correctness checker which is supplied to the program. A program correctness checker is an algorithm for checking the output of a computation. Given a program and an instance on which the program is run, the checker certifies whether the output of the program on that instance is correct. Thus, program checking yields certificates, mostly in form of mathematical proofs, about program behavior. This is in contrast to testing where only acceptance or rejection answer is delivered by the testing program. The research on certifying compilers follows the program checking approach. The notion of program checking was first proposed by Blum and Kannan in 1989 in [21] and later by Blum and Kannan and Wasserman in [22, 113]. In these papers, the authors demonstrate the usability of the program checking method on selected group-theoretic problems and numeric problems. Nevertheless, the first certifying compiler was published in 1975 by Samet [103].

### 1.6.1 Work on certified compilers

The first proof of the correctness of the compiling algorithm was published by McCarthy and Painter in [71]. They present a paper-and-pencil proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language. The ultimate goal of their work, however, is to set up a formal framework for the future research aiming at machine-checkable proofs that compiler algorithms are correct<sup>1</sup>. Some of the concepts and the proof idea presented in their paper are now standard in both the certified and certifying compilers.

More paper-and-pencil proofs of the correctness of compilers or their parts can be found in [74, 108, 6, 50, 51, 89, 9]. Dave [35] gives a survey of literature about the research on the compiler correctness until the year 2003. In the following, we focus on the work on certified compilers in which correctness proofs are machine-checked.

Probably, the first work on certified compilers that was conducted in a theorem prover and was concerned with both the compiler algorithm correctness and the compiler implementation correctness was presented by Bevier et al. in [14]. In their work, they outline an approach to the verification of the execution environment of programs which are written in a subset of the language Gypsy. The code generator for the Gypsy language is verified in [121] and sits atop a "stack" of verified system components building the execution environment. The target language of the code generator is the Piton [73] assembly language. The stack elements below the code generator are an assembler and linking loader [73], a simple operating system kernel [13], and a microprocessor design [54]. Each of these elements are formally specified and proved correct in the Boyer-Moore theorem prover [26].

Curzon [31, 32, 34, 33] presents a machine-checked verification of the specification of a code generator whose source language is a subset of the structured assembly language Vista [56] and the target language is a subset of the assembly language for the VIPER microprocessor [30]. The code generation correctness proof was conducted in the HOL system [46, 44]

In a more recent work, Nipkow [81] specified and verified a simple lexical analyzer generator in the Isabelle/HOL theorem prover [85].

<sup>1</sup> [71] was published in 1966. At the time, the area of automated theorem proving was in the early stage of development. One of the first tools for automated theorem proving, Edinburgh LCF [72] and Cambridge LCF [90], were implemented in the early 1970's. The successor of LCF is the HOL system [52, 45, 46, 44, 47]. The theorem prover Isabelle/HOL [91, 85], which we used in our work, is among a successor of HOL.

Broy et al. [28] present a formalization of a denotational semantics for a small functional language and a correctness proof of its interpreter conducted in the Isabelle/LCF which is the LCF logic [72] instantiation of the generic theorem prover Isabelle [91].

A substantial body of work [86, 88, 84, 82, 58, 83, 59, 60] on mechanically certifying properties of Java-like languages and their translations to bytecode was made in the Isabelle/HOL by working group of Nipkow in Munich: Nipkow and Klein [82, 58] formalized a subset of the Java Virtual Machine bytecode, proved its type safety, and proved an executable bytecode verifier in the style of Kildall’s algorithm correct. Von Oheimb [87, 112] formalized a sequential subset of the Java language and proved its type safety. In [112], he presents the formalization of a Hoare-like axiomatic semantics of partial correctness and a proof of its soundness w.r.t. the operational semantics and its completeness. Strecker [107] presents a formal proof of correctness for a subset of the Java source language to a subset of the Java bytecode language. The formalizations of the languages involved in the translation are based on the formalizations presented in [82, 58, 87, 112]. Later, Berghofer and Strecker [10] extend the formalization of the compiling algorithm in [107] by adapting it to the source language  $\mu$ Java and a subset of Java Virtual Machine bytecode as the target language formalized in [84], prove the algorithm correct and use the extraction facility of Isabelle/HOL to generate an executable compiler program in the programming language ML. Nipkow [83] presents a formalization of a Java-like language Jinja, which is basically a subset of Java. In this work, he defines a small and a big step operational semantics for Jinja and shows that they are equivalent. Klein and Nipkow [60] extend the formalization of Jinja in [83] by the formalization of a bytecode language, which is a subset of the Java Virtual Machine bytecode language, its operational semantics, a bytecode verifier in the style of the Kildall’s algorithm, and a type safety proof for the bytecode language. Further, they formalize a compiler algorithm which uses the Jinja and the bytecode languages as source and target languages, respectively, and show that compilation of a Jinja program preserves its big-step semantics, the well-typedness of its expressions, and its well-formedness, i.e. that compilation of a well-typed and well-formed Jinja program produces a well-typed and a well-formed bytecode. The result of their work is a unified formal model of both a Java-like source language, a JVM bytecode-like target language, and a corresponding compiler.

CompCert [68, 16, 67, 69, 15, 12, 99] is an ongoing project of the working group of Leroy that aims at developing a realistic and lightly-optimizing<sup>2</sup> compiler CompCert that generates PowerPC code from Clight, a large subset of the C programming language. [68] provides an overview of the project. Most of the compiler phases of the CompCert compiler are specified and verified within the proof assistant Coq [11]. An executable compiler is obtained by automatic extraction of executable Caml code from the Coq specification. The proofs for the correctness of the phases are based on the notion of preservation of observational behavior. In [16, 15], Blazy et al. present the formal verification of the front-end of the CompCert compiler, which translates Clight into the Cminor language, a low-level imperative intermediate language. The specification of the dynamic semantics of Clight is based on the memory model for C-like imperative languages described by Blazy and Leroy in [69]. In [67], Leroy present the formal verification of the back-end of the CompCert compiler. The input language of the back-end of the CompCert compiler is the Cminor language and the output language is PowerPC code. The back-end produces four intermediate programs in four different languages and data-flow based optimizations are performed on intermediate representations in the first of these languages, RTL (Register Transfer Language, also known as “3-address-code” language). In [12], Bertot et al. report on the correctness proof of compiler optimizations based on data-flow analysis and performed on RTL representations. Rideau et al. [99] describe the formal

<sup>2</sup> The CompCert compiler performs two optimizations: constant folding and common subexpressions elimination but no code-moving optimizations.

verification of a compilation algorithm that translates parallel assignments between variables into a semantically equivalent sequence of atomic assignments. The ultimate goal of this work is to have a verified specification of the translation algorithm from which a functional program is extracted by applying the extraction facility of the theorem prover Coq and integrated into the Compcert compiler.

Lacey et al. [63, 62, 61] present a specification language for transformations that combines elements of rewriting, temporal logic and logic programming. In this language, the transformations are specified using rules of the form *action if condition*, where each rule has some free variables. The *action* part of a rule specifies how to transform a program in terms of its free variables, and the *condition* part of the rule is a temporal logic formula which specifies what condition must hold for the free variables of the rule. As a proof of concept, Lacey describes in [61] a specification language for optimizations of program written in a very simple imperative language and provides a paper-and-pencil proof for the correctness of the optimizations.

Lerner et al. [65, 98, 66, 64] follow philosophically similar approach as Lacey. In [65], they present a domain-specific language Cobalt for writing optimization specifications as predicates over the entire control flow graph in a restricted version of temporal logic. The specifications can be checked for soundness by generating from them a set of proof obligations and asking an automatic theorem prover to discharge them. The authors proved by hand that if a Cobalt optimization satisfies these obligations then the specified optimization is sound, i.e. it preserves the semantics of any program it optimizes. Later, Lerner et al. [98, 66] and Lerner in [64] present the successor of Cobalt, a language Rhodium, which uses a separate and extensible set of local propagation and transformation rules. Rhodium is more expressive than Cobalt as it allows for expressing data-flow facts explicitly and in Cobalt it is made implicitly.

Blech et al. [17] present two different formalizations of static single assignment (SSA) form in the Isabelle/HOL, which follow two different approaches, and two correctness proofs of code generation from static single assignment form which verify these formalizations. Their first formalization of the SSA form is based on term graphs [80] and abstract states machines [48, 49, 23] and the second formalization is based on partial orders. In their work, Blech et al. conduct the correctness proofs for both formalizations of the SSA form and formalize sufficient, easily checkable correctness criteria characterizing correct compilation results which can be used in a program checker [38]. The latter is a part of the translation validation approach, which we discuss below in work on certifying compilers.

### 1.6.2 Work on certifying compilers

The first work on a certifying compiler was published by Samet in [103]. In his work, Samet designed and implemented translation result checking program for proving that programs written in a subset of LISP 1.6 are correctly translated into LAP, an assembly language for the PDP-10 computer. The proof algorithm of the checking program is independent of the translation algorithm. For the purpose of the proof procedure, Samet implemented the notions of a canonical form of LISP programs, a rederived form of LAP programs, and the notion of matching of the canonical form of a LISP program and the rederived form of a LAP program, [104, 105]. The algorithm proving translation correct takes the source LISP program and the target LAP program as input, converts them into their canonical and rederived forms, respectively, and proves that they match. The matching procedure manipulates the canonical form to obtain a form that is identical to the rederived form [106].

The idea of the PCC approach, which was described in Section 1.1, was firstly published by Necula and Lee in [78]. In their work, they described how the PCC approach can be applied to implement efficient network packet filters and later on, in [75], they demonstrated how the

approach to can be applied to certify type safety of hand-optimized assembly language programs. Later, Necula and Lee demonstrated the concept of a certifying compiler based on the PCC approach [76, 79]. The demonstration is exemplified by a compiler from a type-safe subset of the C language to optimized DEC Alpha machine code that contains a certifier checking automatically type safety and memory safety of an assembly language program produced by the compiler. In [29], Colby et al. describe design and implementation details of an optimizing compiler that compiles Java bytecode [70] into target code for the Intel x86 architecture [55].

The idea of the FPCC approach, which was described in Section 1.1, was proposed by Appel and Felty in [5, 4].

Lightweight bytecode verification is an instance of PCC, where the proof carried by the bytecode is the type annotation of the code. The lightweight bytecode verification approach can be applied to resource-bounded Java Virtual Machine (JVM) implementations on smart cards, which have to rely on cryptographic methods to ensure that bytecode verification has taken place off-card. In order to allow on-card verification, Rose [102] proposes a sparse annotation of bytecode with types to enable a one-pass verification of well-typedness. In the presence of type annotations, problem of type reconstruction for a bytecode transforms into a type checking problem, which can be solved more easily. Klein and Nipkow [57] formalized a variant of lightweight bytecode verification in [102] and proved it correct.

Nipkow et al. [118, 117, 116, 115] propose a generic framework for PCC which is developed and mechanically verified in Isabelle/HOL. In their framework, they formalize and prove sound a verification condition generator with minimal assumptions on the underlying programming language, safety policy, and safety logic. They instantiated the framework to a simple assembly language [118, 116] and Jinja bytecode [117, 115] and formalized a safety policy for arithmetic overflow. Their approach differs from PCC by Necula and Lee and FPCC by Appel and Felty in that it works with an explicit, executable, verified VCG, which is generated from the verified specification using the extraction facility of the theorem prover Isabelle/HOL.

The TV approach, which was described in Section 1.1, was developed independently by Zimmermann and Gaul [123, 122] and by Pnueli et al. [92]. Zimmermann and Gaul applied the TV approach to code generators using the DEC-Alpha as target architecture. They use the notion of abstract state machines to specify the semantics of the source and the target language and specify the code selection phase using a large set of term-rewrite rules. For these rules, they provide a proof that performing the code selection phase using the rules produces a code which preserves observable behavior of the original intermediate program which served as input for the phase. Checking if no register containing a live value, i.e. a value that is still required, is written is done by an independent program checker. The checking requires that intermediate programs have annotations giving a rule cover for each instruction, a register assignment w.r.t. a rule cover, and a schedule specifying the order of application of term-rewrite rules. The correctness of the assembly phase is validated by another program checker. For checking purposes, they specify a large set of term-rewrite rules for the code selection phase and verify them mechanically.

Pnueli et al. [92] developed the TV approach in the area of safety-critical systems and considered translation from the synchronous multi-clock data-flow language SIGNAL to the asynchronous C code. They use synchronous transition systems to formalize the semantics of the source and the target languages. Their notion of program equivalence is based on the notion of refinement between synchronous transition systems. Their validator is provided by a single refinement proof rule whose premises are mechanically verified.

The implementations of the TV approach by Zimmermann and Gaul [123] and Pnueli et al. [92] demonstrate that the approach works well for code generation with the one-to-many correspondence between program points of the source and the target programs and, as stated by Pnueli

et al. in [92], in all scenarios of translating synchronous languages in which the source and the target programs are limited to a single external loop.

Zuck et al. [7, 125, 124, 126] extended the application area of the TV approach to optimizations performed a compiler targeted at EPIC architecture (the SGI Pro-G4). In their work, they distinguish between structure preserving and structure modifying optimizations. The former ones cover most high-level optimizations and admit a clear definition of the corresponding program points relation. The latter ones admit no such relations. The structure modifying optimizations cover reordering transformations such as loop distribution and fusion, loop tiling, and loop interchange. The implementation of their validator pursues a similar approach to the one presented by Pnueli et al. in [92]: For both transformation classes, they give common semantics to the source and the target language using transition systems. The notion of correct translation is defined in terms of a refinement relation over transition systems. The validation technique for structure preserving transformations is similar to the one in [92]. The validator is equipped with a single proof rule which is proved separately in [124]. It breaks the proof into a set of proof obligations by applying the rule. The proof obligations are then shown to be valid using auxiliary invariants which can be provided by the compiler or inferred by a set of heuristics and analysis techniques. For structure modifying transformations, they propose a set of additional "permutation rules" which the validator may use to deal with these transformations. They provide a paper-and-pencil sketch of the soundness proof for these rules [126].

Necula [77] implements a prototype optimization validator for the GNU C compiler which follows the TV approach. His validator requires no auxiliary informations from the compiler at all. Instead of this, it applies various heuristics to identify the optimizations performed and to recover associated refinement mappings. This can result in a false alarm. A false alarm means that the validator rejects a correct optimization due to its inability to understand precisely what transformation took place. Necula reports that the ratio of false alarms is very low for most optimizations, but for some others about 10% of validations result in false alarms. As stated in [124], the main limitation of Necula's validator is that its heuristics can only be applied to structure-preserving optimizations.

Van Engelen et al. in [109, 110] presents a work that is closely related to Necula's work in [77]. The authors describe an implementation of an automatic TV system in the *vpo* compiler [8] which is able to validate all code-improving transformations in the *vpo* compiler, except those that affect blocks across loop levels. Further, the system is less restrictive than the one presented by Necula [77] as it is also able to validate transformations which change the branch structure of the program. The key idea of the validating system is based on the observation that most of the transformations typically consist of only a few changes performed locally within a region. To validate such a transformation, it is sufficient to identify that region and its entry point and to show that the effects of the changes associated with the transformation on the rest of the programs are the same. The effects of two regions are considered semantically equivalent if they are identical at each exit point of the region. The system automatically detects the region associated to the changes associated to a transformation by identifying a pair of corresponding blocks in the source and the target programs that were syntactically changed and finding the closest block in the control-flow graph that dominates the blocks from the pairs. The blocks in the pair identified by the system need not have the same basic block structure. Symbolic computations of the effects are performed by the CTADDEL system [111].

The Verifix [43, 41, 39] project develops methods for combination of techniques from certified and certifying compilers to construct certifying compilers which are as efficient as typical commercial compilers. The correctness of the Verifix compiler is shown in two steps: The first step uses verification techniques from certified compilers and proves the correctness of the compiling algorithms in the compiler (compiler algorithm correctness). The second step shows compiler im-

plementation correctness by using techniques from program checking [39, 40, 38]. The correctness of the program checker algorithm itself is shown using standard program verification techniques. The correctness of the translation of the program checker algorithm written in a high-level language into machine code is guaranteed by a rigorous implementation of an initial compiler [42, 36]. In [38], Glesner gives an overview of the program checking approach and of the results of applying program checking in optimizing backend transformations.

Credible compilation (CC) is a (paper and pencil) framework for certifying optimizations proposed by Rinard and Marinov [101, 100]. Their approach is a hybrid of the TV and the FTV approaches: it is not type-specialized, as the FTV approach, but it is phase-specialized, as the TV approach. The CC approach is not type-specialized as it translates programs into their representations as constants definitions. The SVF of a credible compiler comprises the definition of the abstract syntax of the intermediate programming language, the definition of the operational semantics, the formalization of a simulation invariant, a simulation criterion *Sim* on two intermediate programs and a set of simulation invariants, and a logic for proving optimizations correct. The simulation criterion *Sim* considers two programs as control flow graphs. The third parameter of *Sim* is a set of simulation invariants, where each simulation invariant in the set defines a pair of corresponding nodes and an invariant. The definition of *Sim* is based on the corresponding node relation defined by the set of simulation invariants. However, as the simulation criterion for two control flow graphs is a function of a relation between the sets of nodes of these graphs, it is not possible to prove a universal transitivity rule which would allow to apply the argument of transitivity to the result of two consecutive optimizations. For this reason, the CC approach is phase-specialized as the TV approach.

Blech and Poetzsch-Heffter present in [19] an approach to certifying code generation phase which can be seen as a hybrid of the approaches TV and FTV. They implemented a compiler for a small subset of the programming language C with a code generation phase which translates intermediate language (IL) programs into the MIPS code. Within the theorem prover Isabelle/HOL, they formalized the operational semantics of the language IL, the operational of the MIPS language, and a translation correctness criterion on an intermediate language program and a MIPS program. The criterion interprets these programs as two STS's with observable behavior defined in terms of sequences of integers printed by the STS's and defines program equivalence of the source and the target programs of the code generation as equality of observable behaviors of their representations as STS's. The compiler performs the code generation phase and translates the source and the target programs the phase into their representations as HOL constants a Isabelle/HOL theory and generates a proof of a lemma which states that these constants fulfill the translation correctness criterion. Thus, the translation step of certification is identical to the one in the FTV approach and the idea of formalizing the translation correctness criterion which interpretes programs as STS's is borrowed from the TV approach. Later, Blech et al. in [20] applied this approach to certification of system transformations performed in adaptable systems. Further, Blech and Gregoire in [18] used the theorem prover Coq and investigated the efficiency and the suitability of this approach for certification of the code generation phase performed during translation of large programs.

## 1.7 Overview of the thesis

This section outlines this thesis. As aforementioned, the focus of the presentation is on the SVF. In the thesis, we present Layers 0 through 5 of our SVF. The presentation of Layer 6, which provides the implementations of proof tactics called in proof scripts, is beyond the scope of this thesis. In general, we do not present proofs of the theorems presented in this thesis as they can be inspected

in the Isabelle/HOL theories in the CD attached to this thesis. We only present proofs for selected theorems for the brevity of the presentation.

There follows chapter-by-chapter outline of the thesis.

## **Preliminaries**

This thesis proposes an approach which applies techniques from mathematical logic and programming language semantics. This chapter briefly surveys the formalisms from higher-order logics we use throughout the thesis.

## **Overview of the SVF**

The main focus of this thesis is on the description of the formal framework which is comprised by the SVF of our FTV system. This chapter gives a brief introduction to our implementation of the SVF. Moreover, the chapter description of the general layer architecture of the SVF for an FTV system. The layer architecture of the SVF is the recurrent theme of this thesis.

## **Translation contract**

This chapter describes the content of Layer 2 of our SVF: the formalization of the translation contract for intermediate language optimizations. The first part of the chapter presents the formalization the abstract syntax of an intermediate language IL. The second part of the chapter presents the formalization the program semantics of IL programs. The last part of the chapter presents the definition of the translation correctness predicate on two IL programs.

## **Type safety of the language IL**

This chapter presents the content of Layer 3 of our SVF. The first part of the chapter presents the formalizations of the notions of program type and well-typedness. The second part of the chapter presents a theorem which states that the IL language is type safe.

## **Optimization independent translation correctness criterion**

This chapter presents the content of Layer 4 of our SVF. The first part of the chapter presents the formalizations of the notion of control flow graphs with blocks, a bisimulation relation, and an optimization independent translation correctness criterion TCC. The second part of the chapter presents an optimization independent translation correctness theorem.

## **Translation correctness criteria for particular optimizations**

This chapter presents the content of Layer 5 of our SVF. For each optimization  $O$  performed by our compiler front-end, the chapter provides a section which presents formal framework provided by Layer 5 for the optimization  $O$ . Each of these sections follows a presentation scheme which adheres to the description of Layer 5 in Section 1.5.2, i.e. consists of five parts which present the following:

- The first part of the section presents the formalization of the notion of data flow analysis result  $\mathcal{A}_O$ .
- The second part of the section presents the formalization of the bisimulation relation  $\mathcal{R}_O$ .

- The third part of the section presents the formalization a translation relation  $\text{transrel}_O$ .
- The fourth part of the section presents the formalization of an optimization correctness criterion  $\text{TCC}_O$ .
- The last part of the section presents an optimization correctness theorem.

### **Evaluation**

This chapter presents the evaluation of our formal framework. The evaluation is split in three parts:

1. The first part provides the estimation of the proof script size and evaluates it.
2. The second part provides the first results of the measurements of time which Isabelle/HOL needs for checking proof scripts generated by our compiler front-end.
3. The third part presents our evaluation of the formal framework presented in this thesis.

### **Conclusions and future work**

The final chapter summarizes the achievements of this thesis. A discussion of the work is presented along with possible future directions of research.



## Chapter 2

---

### Preliminaries

This chapter presents a brief introduction to the theorem prover Isabelle/HOL, basic mathematical notions in HOL, and conventions we adopted in this thesis. In the presentation we assume that the reader is familiar with basic concepts in the realm of compiler construction and mathematical logic. The content of Sections 2.1 and 2.2 is an adapted compilation of the sections with preliminaries in [84, 87].

#### 2.1 Modelling and proving in Isabelle/HOL

The term *higher-order logic* traditionally means a typed logic that permits quantification over functions or sets. In the theorem proving community, higher-order logic is often abbreviated to HOL and refers to Church's Simple Theory of Types, which is a variant of the lambda calculus - a typed lambda calculus. The work reported in this thesis has been conducted with the help of Isabelle/HOL: Isabelle [91] is a generic interactive theorem prover, and Isabelle/HOL an instance of Isabelle which is a formalization of HOL with equality and total polymorphic higher-order functions. The set of the primitive inference rules of HOL comprises the law of the excluded middle, which makes the logic classical, and the system Isabelle/HOL is more than just a theorem prover for HOL, it is a fully fledged specification and programming language. Therefore familiarity with classical predicate logic and functional programming is required when reading this thesis. Coming from a programming perspective, one could characterize Isabelle/HOL as combination of functional programming, logic programming, and quantifiers. The remainder of this section describes the main features of Isabelle/HOL and a selected content of predefined theories we used in our work. The description is made from an abstract perspective, i.e. as independent as possible of the theorem prover.

##### 2.1.1 Terms, types, formulae and theories

The type system is similar to that of typed functional programming languages like ML. There basic types like **bool**, **nat**, and **int**, and type constructors (written postfix) like  $(\tau)\mathbf{list}$  and  $(\tau)\mathbf{set}$ . Functions types are written  $\tau_1 \rightarrow \tau_2$  and represent the type of all total functions. Type variables, which are used to express polymorphism, are written  $\alpha, \beta$  etc.

Terms are formed as in  $\lambda$ -calculus by application and abstraction. The constructions **let**  $x = e_1$  **in**  $e_2$  **end**, **if**  $b$  **then**  $e_1$  **else**  $e_2$  and **case**  $e$  **of**  $p_1 \rightarrow e_1 \mid \dots$  familiar from functional programming are also supported. Formulae are terms of type **bool**.

Modules in Isabelle/HOL are called *theories* to emphasize their mathematical content. They contain collections declarations and definitions of types and constants (which include functions). Besides the basic non-recursive definitions of the form

$$name \equiv term$$

Isabelle/HOL additionally provides several application-oriented definition principles: recursive datatypes, recursive functions and inductively defined sets. Although we used inductively defined sets in our work, they are not presented in this thesis. In the following, we only discuss recursive datatypes and recursive functions.

### 2.1.2 Recursive datatypes and functions

Functional programming is supported by constructs for the definition of recursive datatypes and recursive functions. A simple example is the theory of lists:

```
datatype  $\alpha$  list = Nil          ([])
              | Cons  $\alpha$  ( $\alpha$  list)  (infixr #)
```

```
consts app ::  $\alpha$  list  $\rightarrow$   $\alpha$  list  $\rightarrow$   $\alpha$  list  (infixr @)
primrec
  [] @ ys = ys
  (x#xs) @ ys = x#(xs @ ys)
```

It defines the recursive datatype of lists together with some syntactic sugar, declares a function `app` (with infix syntax `@`), and defines `app` by primitive recursion. The key proof technique in this setting is structural induction on datatypes. Such proofs are performed automatically by Isabelle/HOL.

More complex recursion patterns can be expressed by *well-founded recursion*, which requires a termination ordering to convince Isabelle/HOL of the totality of the defined function.

Totality is always the key requirement when defining a function in HOL since HOL is a logic of total functions, and the introduction of a truly partial function would cause an inconsistency.

### 2.1.3 Isabelle/HOL library

Our formalization of the SVF makes use of many predefined theories. We quickly summarize the most important ones.

#### Cartesian products

The type  $(\tau_1 \times \tau_2)$  of Cartesian products of  $\tau_1$  and  $\tau_2$  comes with projection functions

```
fst ::  $\alpha \times \beta \rightarrow \alpha$ 
fst  $\equiv \lambda(x, y). x$ 
```

```
snd ::  $\alpha \times \beta \rightarrow \beta$ 
snd  $\equiv \lambda(x, y). y$ 
```

and with (postfix) operators for constructing the transitive closure  $R^+$  and transitive reflexive closure  $R^*$  of a relation  $R :: (\alpha \times \alpha)\text{set}$ . Although we used these operators in our work, they are not presented in this thesis.

## Lists

In addition to the basic list constructs show above, the list library contains the following relevant functions:

```
length ::  $\alpha$  list  $\rightarrow$  nat
set    ::  $\alpha$  list  $\rightarrow$   $\alpha$  set
map    ::  $(\alpha \rightarrow \beta) \rightarrow \alpha$  list  $\rightarrow$   $\beta$  list
zip    ::  $\alpha$  list  $\rightarrow$   $\beta$  list  $\rightarrow$   $(\alpha \times \beta)$ list
nth    ::  $\alpha$  list  $\rightarrow$  nat  $\rightarrow$   $\alpha$ 
```

The meaning of `length`, `set`, `map`, and `zip` should be obvious. `nth xs i` selects the  $i$ -th element (starting from 0) and is abbreviated by `xs!i`. The usual notation `[x, y, z]` instead of `x#y#z#[]` is also supported.

## Options

The datatype of optional values

```
datatype  $\alpha$  option = None | Some( $\alpha$ )
```

is used to add a new element `None` to a type and wrap the remaining elements up in `Some`. We use this datatype when modelling partial mappings. Another method to declare a partial mapping will be presented in the next section.

## Mappings

In our work, we frequently reason about partial functions in which domains are modified. Typical application is the state of computation (where variables are mapped to their value during execution of a program). They are called mappings and are defined as functions with optional range type:

```
types  $\alpha \rightsquigarrow \beta = \alpha \rightarrow \beta$  option
```

For the manipulation of mappings the following functions are provided:

```
empty      ::  $\alpha \rightsquigarrow \beta$ 
_(_  $\mapsto$  _) ::  $(\alpha \rightsquigarrow \beta) \rightarrow \alpha \rightarrow \beta \rightarrow (\alpha \rightsquigarrow \beta)$ 
mapof      ::  $(\alpha \times \beta)$ list  $\rightarrow$   $(\alpha \rightsquigarrow \beta)$ 
```

They represent the empty mapping, updating in one place, and turning an association list into a mapping. Their definitions are as follows.

```
empty       $\equiv \lambda k. k$ 
m(x  $\mapsto$  y)  $\equiv \lambda k. \text{if } k = x \text{ then Some}(y) \text{ else } m(k)$ 
mapof([])  = empty
mapof((x, y)#l) = (mapof l)(x  $\mapsto$  y)
```

## 2.2 Programming language formalization style

When formalizing a programming language in a theorem prover, which is also called *embedding*, one has basically two options [84, 87, 24, 2, 25, 96, 97, 114]:

*Deep embeddings* use a separate datatype to define the abstract syntax of the language and define the semantics by formalizing a semantics function which maps the abstract syntax to a semantics. This is useful when doing meta-theory in the language since one can express properties of the syntactic structure and prove generic properties of the language such as type soundness. *Shallow embeddings* define the semantics directly, i.e. each construct in the language is represented directly by some function on a semantic domain. This is advantageous for reasoning about individual programs of the language as the extra syntactic level is avoided.

In this thesis, we only use a deep embedding. The reason for this decision is as follows: We are interested in implementing a specification and verification framework for a certifying optimizer. Within this framework, we want to formalize program independent optimization correctness criteria and respective optimization correctness theorems that can be applied in proofs verifying correctness of an optimization. Such correctness criteria have the form of binary predicates on two programs that express some syntactic patterns for corresponding syntactic fragments of these programs. Further, all proofs of these theorems require introducing certain assumptions saying that the corresponding syntactic program fragments are well-typed and their types comply with the states of computation during execution of the programs. As proving type soundness of the language is the prerequisite for discharging such assumptions, the deep embedding style was a natural choice in our work.

## 2.3 Conventions

In this thesis, we adopted the presentation style of the programming language formalizations from Winskel in [120]. Therefore, in our presentation, we do not use built-in HOL types, such **nat** and **bool**. Instead of this, whenever needed, we introduce fresh syntactic sets associated with the formalism of interest. Further, we do not use polymorphism in our presentation. The only polymorphic types and functions we use in our presentation are the ones already presented in Sections 2.1 and 2.2. Instead of this, for each instantiation of the polymorphic type, we declare a separate syntactic set and a metavariable ranging over this set. The following example demonstrates the type declaration in Isabelle/HOL and how it would be presented using our convention.

*Example 2.1.*

- A type declaration in Isabelle/HOL:

$$\text{types } \textit{int\_bool\_list} = (\text{int list}) \times (\text{bool list})$$

- A declaration of a syntactic set *IntBoolList*:
  - The syntactic sets associated with the definition of *IntBoolList* are the following:
    - numbers **Int**,
    - truth values **Bool**.
  - The metavariables ranging over these sets are the following:
    - *i* ranges over numbers **Int**,
    - *b* ranges over truth values **Bool**.

- The formation rules for the syntactic set *IntBoolList*:

$$\begin{aligned}i &\in \mathbf{Int} = \{0, 1, -1, 2, -2, \dots\} \\b &\in \mathbf{Bool} = \{\mathbf{true}, \mathbf{false}\} \\IntList &\ni il ::= [] \mid i\#il \\BoolList &\ni bl ::= [] \mid b\#bl \\IntBoolList &\ni ibl ::= (il, bl)\end{aligned}$$

◇



## Chapter 3

---

### Overview of the SVF

This section focuses on the architecture of the SVF. As aforementioned in Section 1.5.2, a SVF consists of two main parts, a specification part and a verification part, and the whole formal framework of the SVF is organized hierarchically in layers. The specification part of the SVF provides a translation contract and the verification part of the SVF provides auxiliary specifications, lemmas, and proof tactics which make up a proof environment for proving translations correct. The bottom layer of the framework comprises the most general formalizations and the top layer comprises the most specific formalizations which are dedicated to particular proof tasks, and the formalizations from upper layers use definitions and apply lemmas from lower layers. In the following, we describe the general purpose of the layers how we implemented these layers within our SVF.

Figures 3.1 and 1.7 give overviews of the general architecture of the SVF and the architecture of our SVF, respectively. We start explaining the bottom layer of the architecture and work through to the top layer.

#### 3.1 Layer 0: Higher-order logic

##### **The general purpose of Layer 0:**

Layer 0 provides an implementation of higher-order logic (HOL) within a specification and verification system which is needed to implement logical formalisms within an SVF for a FTV system.

##### **Layer 0 in our implementation:**

Our implementation of the SVF is based on the implementation of HOL within the theorem prover Isabelle/HOL, which provides the implementation of HOL in the theory HOL.

#### 3.2 Layer 1: Logic extension

##### **The general purpose of Layer 1:**

Layer 1 provides auxiliary formalizations in HOL which are both program independent and optimization such as the formalizations of the set theory, the theories of well-founded recursion, well-founded relations, partial mappings, lists, etc.

Further, Layer 1 provides a programming interface for the verification part of the SVF which allows for implementing proof tactics. The proof tactics support automatic verification of proof scripts.

Layer 6:	proof environments specific to particular proof tasks
Layer 5:	translation correctness criteria specific to particular compiler phase
Layer 4:	compiler phase independent translation correctness criterion
Layer 3:	type safety proofs
Layer 2:	translation contract
Layer 1:	auxiliary HOL extensions
Layer 0:	implementation of HOL within a theory prover

**Fig. 3.1.** The layer architecture of the SVF

**Layer 1 in our implementation:**

The Isabelle/HOL provides the implementations of the above formalisms in the theory `Main`. By way of example, we list some of them, which we used in our implementation of the SVF.

- the set theory,
- basic predefined types  $\alpha$ ,  $nat$ ,  $\alpha list$ ,  $int$ ,  $real$ ,  $\alpha option$ , etc.
- the type definition mechanism,
- the `datatype` definition package,
- the mechanisms of defining recursive functions,
- Isabelle/HOL's package for inductive definitions, etc.

Further, we formalized a small library of auxiliary specifications and auxiliary lemmas about lists, sets, mappings, etc, which proved to be useful in our proofs.

During our work on the SVF, we used two instantiations of Isabelle, Isabelle/HOL and Isabelle/Isar. We used Isabelle/Isar to formalize Layers 1 through 4 of the SVF architecture and Isabelle/HOL to implement proof tactics in Layer 5. Isabelle/Isar is an extension of Isabelle/HOL providing a proof language which is perfectly suitable for setting up specifications and performing interactive proofs. The proof language of Isabelle/Isar is highly apprehensible for the human reader which is of particular interest when it comes to form an opinion about the translation contract provided by Layer 2. Isabelle/HOL is a predecessor of Isabelle/Isar and provides direct programming interface to the underlying ML system. We used that interface to write ML functions computing proof tactics for particular proof tasks.

### 3.3 Layer 2: Translation contract

#### The general purpose of Layer 2:

Layer 2 provides the specification of a translation contract which comprises

- the formalization of the source language  $SL$  including
  1. the formalization of the abstract syntax of  $SL$  programs which is done by giving the definition of a HOL type of  $SL$  programs,  $\mathbf{Program}_S$ ,
  2. the formalization of the meaning of an  $SL$  program which is done by giving the definition of a HOL type  $\tau_S$ ,
  3. the formalization of a program semantics function  $\mathbf{Sem}_S : \mathbf{Program}_S \rightarrow \tau_S$  which maps  $SL$  programs to their meanings of the type  $\tau_S$ ,
- the formalization of the target language  $TL$  including
  1. the formalization of the abstract syntax of  $TL$  programs which is done by giving the definition of a HOL type of  $TL$  programs,  $\mathbf{Program}_T$ ,
  2. the formalization of the meaning of a  $TL$  program which is done by giving the definition of a type  $\tau_T$ ,
  3. the formalization of a program semantics function  $\mathbf{Sem}_T : \mathbf{Program}_T \rightarrow \tau_T$  which maps  $TL$  programs to their meanings of the type  $\tau_T$ ,
- the formalization of a transitive relation

$$\_ \leq_R \_ : \tau_S \times \tau_T \rightarrow \mathbf{Bool}$$

which allows to compare a meaning of an  $SL$  program with a meaning of a  $TL$  program.

- the specification of the translation correctness predicate  $\mathbf{corrTrans} : \mathbf{Program}_S \times \mathbf{Program}_T \rightarrow \mathbf{Bool}$  which takes a source language program  $S$  and a target language program  $T$  and checks if  $T$  is a correct translation of  $S$ . In the FTV approach, the definition of  $\mathbf{corrTrans}$  has a straightforward form:

$$\mathbf{corrTrans}(S, T) \equiv \mathbf{Sem}_S(S) \leq_R \mathbf{Sem}_T(T)$$

The following should be noted about the translation contract:

Firstly, the transitivity property of the relation  $\leq_R$  is essential for an SVF which follows the FTV approach as it enables the compiler to generate an *explicit* translation certificate containing a proof of the statement  $\mathbf{corrTrans}(S, T)$  for a source program  $S$  and a target program  $T$  even if  $T$  is the output of a chain of program transformations performed by the compiler phases.

Secondly, the formalizations of the languages  $SL$  and  $TL$  include the formalizations of their underlying type systems. In particular, if the program semantics functions for  $SL$  and  $TL$  are partial mappings which are well-defined only for well-typed programs, then Layer 2 also provides the formalizations of well-typedness for both  $SL$  and  $TL$  by giving the definitions of program well-typedness predicates  $\mathbf{wt}_S : \mathbf{Program}_S \rightarrow \mathbf{Bool}$  and  $\mathbf{wt}_T : \mathbf{Program}_T \rightarrow \mathbf{Bool}$  for the  $SL$  and  $TL$ , respectively, that take programs in the languages  $SL$  and  $TL$ , respectively, and check if they are well-typed. In this case, the definition of  $\mathbf{corrTrans}$  has the following form:

$$\mathbf{corrTrans}(S, T) \equiv \text{if } \mathbf{wt}_S(S) \wedge \mathbf{wt}_T(T) \text{ then } \mathbf{Sem}_S(S) \leq_R \mathbf{Sem}_T(T) \text{ else False}$$

#### Layer 2 in our implementation:

As the purpose of our SVF is to serve as a verification framework for intermediate language optimizations and we are interested in translation certificates which attest to the equality of two intermediate language program semantics, we formalized a special case of the translation contract in which

1. the source language  $SL$  and the target language  $TL$  are equal,
2. the program semantics functions are total functions, and
3. the relation  $\leq_R$  is the equality relation.

Our translation contract comprises the following:

- The formalization of an intermediate language IL includes
  - the abstract syntax of the language IL which is defined by giving formation rules for a set **Program**.
  - the notion of a configuration which models the state of execution of an IL program which is defined by giving formation rule for a set **Configuration**
  - the operational semantics of the language IL which is formalized in terms of configurations  $\sigma \in \mathbf{Configuration}$  and a transition function

$$\text{exec} : \mathbf{Program} \times \mathbf{Configuration} \rightarrow \mathbf{Configuration} .$$

The transition function  $\text{exec}$  computes the successor configuration  $\text{exec}(P, \sigma)$  for an IL program  $P$  and a configuration  $\sigma$ .

- the definition of a function

$$M : \mathbf{Program} \times \mathbf{Configuration} \times \mathbf{Nat} \rightarrow \mathbf{Configuration}$$

which models a simple machine executing IL programs. We use this function in our proofs when we want to formalize a configuration which is the result of partial execution of an IL program  $P$ : Given that  $\sigma_0$  denotes an initial configuration for the program  $P$ , then  $M(P, \sigma_0, n)$  denotes a configuration which is the result of  $n$  successive applications of the transition function  $\text{exec}$  to  $\sigma_0$ .

- the notion of observable behavior of an IL program: A transfer of execution from a configuration  $\sigma$  to the successor configuration  $\text{exec}(P, \sigma)$  can result in behavior which is observable to the outside world. This behavior can be either printing an integer or termination of execution, which we model by an integer token and a termination token. Therefore, we model the observable behavior of an IL program as a sequence of tokens printed during its execution. The set of observable behaviors is formalized by giving formation rules for a set **ObservableBehavior**.
- the program semantics for the IL language which is defined by a function

$$\text{Sem} : \mathbf{Program} \rightarrow \mathbf{ObservableBehavior}$$

which is total and maps IL programs to their observable behaviors. The definition of  $\text{Sem}$  is based on the operational semantics of IL and on the notion of partial executions of IL programs.

- the definition of a translation correctness predicate  $\text{corrTrans} : \mathbf{Program} \times \mathbf{Program} \rightarrow \mathbf{Bool}$  which defined as follows.

$$\text{corrTrans}(S, T) \equiv \text{Sem}(S) = \text{Sem}(T)$$

Informally, the definition of  $\text{corrTrans}$  says that two IL programs  $S$  and  $T$  are semantically equivalent iff they have the same observable behaviors, i.e. they print the same sequences of tokens.

### 3.4 Layer 3: Type safety proofs

#### The general purpose of Layer 3:

Layer 3 provides proofs that the source and the target languages,  $SL$  and  $TL$ , respectively, are type safe. In the following, we explain why the SVF for a FTV system has to provide these proofs.

The type safety is a property of a language which is a necessary prerequisite whenever one wants to prove an invariant property about partial executions of a program  $P$  in that language. The proof of such a property is by induction on the length of partial execution of  $P$  and to prove the inductive step one has to derive the statement that the invariant holds for a configuration  $\sigma_{n+1}$  which is the result of partial execution of the length  $n + 1$  from the assumption that the invariant holds for a configuration  $\sigma_n$  which is the result of partial execution of the length  $n$ . As  $\sigma_{n+1}$  is the result of application of transition function  $\text{exec}$  to  $\sigma_n$ , one has to show that the invariant holds for the successor configuration  $\text{exec}(P, \sigma_n)$ . To prove this, one has to introduce some additional assumptions about  $\sigma_n$  which enables the derivation of the value of  $\text{exec}(P, \sigma_n)$ . For instance, if we want to conduct the proof of the inductive step by cases over instructions of  $P$ , then we have to prove first that the program counter  $pc$  in the current state of execution  $\sigma_n$  is valid and points to one of the instructions of  $P$ , i.e. to show that  $0 \leq pc < n$  holds, where  $n$  is the length of the instruction list of  $P$ . However, the configuration  $\sigma_n$  fulfills the property  $0 \leq pc < n$  only, if  $P$  is well-typed and the language is type safe.

The same argumentation holds in case of proofs of invariant properties about partial executions of two programs such as bisimulation relation - except that in this case, there is an aggravating factor to be added: As the compiler phases transform programs in such a way that they need to make different number of transitions to produce the same observable behaviors, one has to formalize what does it mean that the lengths of partial executions of a source and a target program are corresponding.

In summary, the necessity of proving type safety of the languages involved in the translation contract is a feature which distinguishes the FTV approach from the TV approach and it comes from the fact that the FTV approach reasons inductively about the whole executions of programs which is in contrast to the TV approach which reasons about executions of small fragments of programs and takes type safety of the underlying system for granted, cf. Section 1.1 for our discussion on the approaches to certifying compilers.

#### Layer 3 in our implementation:

The formal framework of our implementation of Layer 3 provides the following:

1. a simple type system for the language IL including formation rules for a set of types of IL programs **ProgramType**,
2. the definition of a program well-typedness predicate

$$\text{wtp} : \mathbf{Program} \times \mathbf{ProgramType} \rightarrow \mathbf{Bool}$$

where an expression  $\text{wtp}(P, \Phi)$  says that an IL program  $P$  is well-typed w.r.t. to a program type  $\Phi$ ,

3. the definition of a configuration conformance predicate

$$\_, \_ \vdash \_ \surd : \mathbf{ProgramType} \times \mathbf{Program} \times \mathbf{Configuration} \rightarrow \mathbf{Bool}$$

where an expression  $P, \Phi \vdash \sigma \surd$  says that a configuration  $\sigma$  conforms to a program type  $\Phi$  and a program  $P$ , and

4. a type safety theorem which, informally, says the following:
 

*Given that  $\sigma_0$  denotes an initial configuration for an IL program  $P$  then,*

$$\text{wtp}(P, \Phi) \text{ implies } \forall n. P, \Phi \vdash \mathbf{M}(P, \sigma_0, n) \surd.$$

In other words, the theorem says that if an IL program is well-typed w.r.t. to a program type, then all partial executions of that program are type safe.

### 3.5 Layer 4: Compiler phase independent translation correctness criterion

#### The general purpose of Layer 4:

The general purpose of Layer 4 is to provide

- the formalization of a translation correctness criterion  $TCC$  on two programs, a source language program  $S$  and a target language program  $T$ ; and other auxiliary parameters describing  $S$  and  $T$  which we abbreviate here as a parameter  $AuxParams$ . The definition of  $TCC$  expresses a sufficient condition of  $\text{corrTrans}(S, T)$  in terms of notions which are not specific to any program transformations performed by the compiler phases, such as the notions of a refinement relation or simulation relation or bisimulation of executions. The formulation of the criterion  $TCC$  is akin to the formulation of translation correctness criteria used in the proof rules in the TV approach and its exact definition is dependent on how the program semantics functions  $\text{Sem}_S$  and  $\text{Sem}_T$  and the transitive relation  $\leq_R$  used in the definition of  $\text{corrTrans}$  are defined. In general, the definition of  $TCC$  is a function of the translation correctness predicate and a proof technique used by the FTV system. For instance, in our SVF, the relation  $\leq_R$  is defined as the equality and the program semantics function  $\text{Sem}$  defines a sequence of output tokens printed by a program. Therefore, we defined the predicate  $TCC(S, T, AuxParam)$  as a predicate of the form

$$\exists B_S B_T \mathcal{R}. \text{bisimulation}(S, T, B_S, B_T, \mathcal{R})$$

where the tuples  $(S, B_S)$  and  $(T, B_T)$  denote a source and a target control flow graphs with blocks and the parameter  $\mathcal{R}$  denotes a bisimulation relation between the sets of configurations and  $\text{bisimulation}(S, T, B_S, B_T, \mathcal{R})$  denotes a property of bisimulation of block-wise executions of  $S$  and  $T$  w.r.t.  $\mathcal{R}$ . Our formulation of the translation correctness criterion  $TCC$  is program transformation independent for the following reasons: Firstly, it is optimization independent as the bisimulation relation  $\mathcal{R}$ , which has to be formalized for an optimization, is  $\exists$ -quantified. Secondly, it is independent of a concrete optimization as the parameters  $B_S$  and  $B_T$ , whose concrete values have to be provided by the compiler for concrete programs  $S$  and  $T$ , are  $\exists$ -quantified.

- a proof of a translation correctness theorem (TCT) whose statement has the following form:

$$TCC(S, T, AuxParams) \implies \text{corrTrans}(S, T) \quad (\text{TCT})$$

In the following, we put forward the motivation behind this layer: As mentioned in Section 1.1, all approaches to certifying compiler have in common that they strive for automatic verification of translation correctness. For instance, the TV approach regards the source and the target programs of a compiler run as STS's and seeks for proof techniques for automatic verification that the target STS refines the source STS. The general strategy of the TV approach to achieve this is to translate the source and the target programs  $S$  and  $T$  into their semantic abstractions  $STS_S$  and  $STS_T$  and to automatically check if they fulfill a translation correctness criterion  $TCC$  by executing the procedure call  $TCC(STS_S, STS_T)$ . Then, the refinement relation between  $STS_S$  and  $STS_T$  holds implicitly by the lemma of the form

$$TCC(STS_S, STS_T) \implies STS_S \sqsubseteq STS_T$$

which is proved separately, e.g. in the theorem prover. The key idea behind that strategy is that checking  $STS_S \sqsubseteq STS_T$  is an undecidable problem, checking  $TCC(STS_S, STS_T)$  is a decidable

problem, checking  $TCC(STS_S, STS_T)$  can be done efficiently, and the compiler generates (hopefully) only programs  $S$  and  $T$  such the procedure calls  $TCC(STS_S, STS_T)$ , where  $STS_S$  and  $STS_T$  are the STS representations of  $S$  and  $T$ , successfully terminate.

Other approaches to certifying compiler, including the FTV approach, follow the analogous strategy. In particular, the strategy of the FTV approach to achieve the automation includes the following steps: Firstly, the source and the target programs of a compiler run,  $S$  and  $T$ , and data structures describing those programs are translated into their syntactical representations as HOL constants in a proof script. Secondly, the proof of  $\text{corrTrans}(S, T)$  is initialized by application of the translation correctness theorem whose statement is analogous to the above statement (TCT). Thirdly, the assumptions of the theorem are discharged automatically by application of a dedicated proof tactic. There is, however, one feature which distinguishes the FTV approach from the TV approaches: As opposed to the TV approach, which is phase-specialized and provides proof techniques for certifying one-phase transformations, the FTV approach seeks for proof techniques for certifying whole transformation chains. Consequently, for each program transformation  $T$ , a SVF following the FTV approach has to provide a transformation correctness criterion  $TCC_T$ , whose definition is specific to the program transformation  $T$ , and a translation correctness theorem about  $TCC_T$  which is analogous to the statement TCT.

At the first glance, this means for our FTV system, which certifies the chain of five optimizations  $CF$ ,  $DAE$ ,  $NI$ ,  $RAI$ , and  $RAE$ , that we can consider each optimization  $O$  as one program transformation and that our SVF should contain a layer providing the formalizations of optimization correctness criteria:  $TCC_{CF}$ ,  $TCC_{DAE}$ ,  $TCC_{NI}$ ,  $TCC_{RAI}$ , and  $TCC_{RAE}$ , respectively, with corresponding proofs of optimization correctness theorems of the form as follows, see below for the explanation.

$$TCC_{CF}(S, T, B_S, B_T, \mathcal{A}_{CF}) \implies \text{corrTrans}(S, T)$$

$$TCC_{DAE}(S, T, B_S, B_T, \mathcal{A}_{DAE}) \implies \text{corrTrans}(S, T)$$

$$TCC_{NI}(S, T, B_S, B_T, \mathcal{A}_{NI}) \implies \text{corrTrans}(S, T)$$

$$TCC_{RAI}(S, T, B_S, B_T, \mathcal{A}_{RAI}) \implies \text{corrTrans}(S, T)$$

$$TCC_{RAE}(S, T, B_S, B_T, \mathcal{A}_{RAE}) \implies \text{corrTrans}(S, T)$$

In the above list, each predicate  $TCC_{O_n}$  has parameters  $S$ ,  $T$ ,  $B_S$ , and  $B_T$  which denote the same values as for the previously mentioned predicate  $\text{bisimulation}$  and its additional parameter  $\mathcal{A}_{O_n}$  denotes the result of data flow analysis which was performed prior to the optimization  $O_n$ . Informally, the predicate  $TCC_{O_n}(S, T, B_S, B_T, \mathcal{A}_O)$  denotes that control flow graphs with blocks  $(S, B_S)$  and  $(T, B_T)$  are in a translation relation which is a function of the optimization  $O$  and  $\mathcal{A}_O$ .

However, designing our correctness criterion layer in this way would lead us to a SVF which is not modular: Each time we would introduce a new optimization  $O_{n+1}$  into our compiler we would have to prove the  $n + 1$ -th optimization correctness theorem of the form

$$TCC_{O_{n+1}}(S, T, B_S, B_T, \mathcal{A}_{O_{n+1}}) \implies \text{corrTrans}(S, T)$$

which would force us to prove for the  $n + 1$ -th time a property that is actually both optimization independent, namely,

*if there exist two control flow graphs with blocks  $(S, B_S)$  and  $(T, B_T)$  and a bisimulation relation  $\mathcal{R}$  such that block-wise executions of  $S$  and  $T$  bisimulate w.r.t.  $\mathcal{R}$ , then the observable behaviors of  $S$  and  $T$  are equal.*

Using our notation, we can express this property concisely as follows.

$$\exists B_S B_T \mathcal{R}. \text{bisimulation}(S, T, B_S, B_T, \mathcal{R}) \implies \text{corrTrans}(S, T) \quad (\text{TCT}')$$

As (TCT') is a special case of (TCT), this leads to us to the second approach to the design of our SVF in which we factor out the statement (TCT') from the optimization correctness criteria  $\text{TCC}_O$ :

If we know the algorithm of an optimization  $O$  and the algorithm of the data flow analysis performed prior to that optimization, then we are able to formalize a bisimulation relation  $\mathcal{R}_O$  and a set  $\mathcal{A}_O$  comprising results of data flow analyses performed prior to that optimization such that we can prove the following statement, which is program independent and specific to the optimization  $O$ :

*if two control flow graphs with blocks programs  $(S, B_S)$  and  $(T, B_T)$  are in a translation relation which is a function of  $O$ , and  $\mathcal{A}_O$ , then block-wise executions of  $S$  and  $T$  bisimulate w.r.t.  $\mathcal{R}_O$*

Using our notation, we can express this statement concisely as follows.

$$\text{TCC}_O(S, T, B_S, B_T, \mathcal{A}_O) \implies \text{bisimulation}(S, T, B_S, B_T, \mathcal{R}_O) \quad (\text{OptSpecCT})$$

In the following, we call a theorem which proves a statement of the form of (OptSpecCT) an *optimization criterion correctness theorem*.

Now, we can redesign the translation correctness criteria layer of our SVF: We split this layer into two layers which are designed as follows:

*Layer 4: Transformation independent translation correctness criterion layer:* In general, we designate Layer 4 to provide the formalization of a *transformation independent translation correctness criterion* TCC and to prove a *transformation independent translation correctness theorem* whose statement is based on the definition of the TCC criterion. The statement of this theorem is analogous to the statement (TCT).

*Layer 5: Transformation specific correctness criteria layer:* In general, we designate Layer 5 to provide, for each transformation  $T$  performed by the compiler, the following:

- the formalization of a *transformation correctness criterion*  $\text{TCC}_T$  which is specific to the definition of the transformation  $T$ ,
- the proof of a corresponding  $\text{TCC}_T$  *criterion correctness theorem* whose statement says that the specification of the  $\text{TCC}_T$  criterion is correct in the following sense:

*If two programs are the source and the target programs of the transformation  $T$  and they fulfill the transformation correctness criterion  $\text{TCC}_T$ , then the programs fulfill the transformation independent translation correctness criterion TCC provided by Layer 4.*

The statement of this theorem is analogous to (OptSpecCT).

- the proof of a *transformation correctness theorem* whose statement is the result of conjoining statements of two theorems: the transformation independent translation correctness theorem provided by Layer 4 and the  $\text{TCC}_T$  criterion correctness theorem provided by Layer 5. The theorem has the following form:

*If two programs are the source and the target programs of the transformation  $T$  and they fulfill the transformation correctness criterion  $\text{TCC}_T$ , then the programs fulfill the translation correctness predicate  $\text{corrTrans}$  provided by the translation contract layer.*

Besides providing the translation contract, providing a transformation correctness theorem for each transformation  $T$  performed by the compiler is the main purpose of the SVF in an FTV system. Such a theorem is directly applicable in translation certificates generated by the compiler.

**Layer 4 in our implementation:**

In our implementation of the SVF, Layer 4 provides all formalizations which are needed to formulate and to prove an optimization independent translation correctness theorem which has the form similar to (TCT). In the following, we present a brief overview of our implementation of Layer 4. Among other things, Layer 4 provides the following:

1. formalization of the notions of basic blocks and block positions,
2. formalization of the set of declarations of control flow graphs with blocks (CFGB) **BlckPosEnv**. A CFGB declaration  $B \in \mathbf{BlckPosEnv}$  is always defined for an IL program  $P$  and it describes the structure of a CFGB which adheres to the structure of a control flow graph (CFG) which corresponds to  $P$ .
3. formalization of a well-formedness predicate on CFGB declarations and IL programs,

$$\mathbf{wfB} : \mathbf{Program} \times \mathbf{BlckPosEnv} \rightarrow \mathbf{Bool}$$

which checks if a CFGB declaration is well-formed w.r.t. an IL program,

4. formalization of an intermediate language  $IL''$ , which is a language of control flow graphs with blocks<sup>1</sup>.  $IL''$  programs are tuples  $(P, B)$  consisting of an IL program  $P$  and a CFGB declaration  $B$ :

$$(P, B) \in \mathbf{Program}'' ::= \mathbf{Program} \times \mathbf{BlckPosEnv}$$

and the operational semantics is defined in terms of block-wise transfers of the flow of control. The denotational semantics of an  $IL''$  program is defined as a sequence of tokens printed by that program, i.e. we give the definition of a program semantics function which is a mapping from the set of  $IL''$  programs to the set **ObservableBehavior**,

5. formalization of the notion of a bisimulation relation  $\mathcal{R}$  as a predicate on the set **Configuration**  $\times$  **Configuration**.

$$\mathbf{BisimulationRelation} = \mathcal{P}(\mathbf{Configuration} \times \mathbf{Configuration})$$

6. formalization of the notion of bisimulation relation between partial executions of two  $IL''$  programs w.r.t. a bisimulation relation  $\mathcal{R}$ . The bisimulation relation is formalized by a predicate **bisimulation** on two  $IL''$  programs and a bisimulation relation  $\mathcal{R}$ .

$$\mathbf{bisimulation} : \mathbf{Program}'' \times \mathbf{Program}'' \times \mathbf{BisimulationRelation} \rightarrow \mathbf{Bool}$$

7. a proof of a transformation independent translation correctness theorem which, informally, says the following:

*if there exist program types  $\Phi_S$  and  $\Phi_T$ , CFGB declarations  $B_S$  and  $B_T$ , and a bisimulation relation  $\mathcal{R}$  such that  $\mathbf{wtp}(S, \Phi_S)$  and  $\mathbf{wtp}(T, \Phi_T)$  and  $\mathbf{wfB}(S, B_S)$  and  $\mathbf{wfB}(T, B_T)$  and  $\mathbf{bisimulation}(S, T, B_S, B_T, \mathcal{R})$ , then  $S$  and  $T$  fulfill the translation correctness predicate  $\mathbf{corrTrans}$  provided by the translation contract layer.*

---

<sup>1</sup> Layer 4 also provides formalization of an intermediate language  $IL'$ . The definition of  $IL''$  is based on the definition of  $IL'$ .

Using our notation, we can express the statement of this theorem as follows.

$$\begin{aligned}
& \exists \Phi_S \Phi_T B_S B_T \mathcal{R}. \\
& \quad \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \\
& \quad \text{bisimulation}((S, B_S), (T, B_T), \mathcal{R}) \\
& \implies \\
& \text{corrTrans}(S, T)
\end{aligned}$$

The theorem is generic in the sense that its statement is optimization independent and not specific to any kind of optimization. For this reason, it is possible to reuse the theorem by defining a different bisimulation relations for particular IL optimizations and to prove optimization specific versions of the theorem after instantiating its  $\exists$ -quantified variable  $\mathcal{R}$  in its assumption. The  $\exists$ -quantified variables  $B_S$  and  $B_T$  will be instantiated in a concrete proof script by concrete constants representing CFGB declarations for concretes source and a target programs.

### 3.6 Layer 5: Translation correctness criteria for particular compiler phases

#### The general purpose of Layer 5:

In general, we designate Layer 5 to provide all formalizations which are needed to formulate translation correctness criteria which are specific to program transformations performed by the compiler and to prove corresponding translation correctness theorems, see the previous section for the motivation behind this layer.

#### Layer 5 in our implementation:

For each optimization  $O$  performed by our compiler, Layer 5 provides the following:

- the formalization of a bisimulation relation  $\mathcal{R}_O$  whose definition is specific to the optimization  $O$ ,
- the formalization of an optimization correctness criterion  $TCC_O(IL_S, IL_T, B_S, B_T, \mathcal{A}_O)$
- a proof of an optimization specific correctness theorem whose statement is equal to the statement (OptSpecCT).

For each intermediate optimization phase  $O$  performed by our compiler, Layer 5 provides the following:

1. The formalization of a set of results of data flow analyses, which are performed prior to the optimization  $O$ , **DataFlowAnalysisResult** $_O$ .
2. The specification of a concrete bisimulation relation  $\mathcal{R}_O$  for the optimization  $O$ . The relation  $\mathcal{R}_O$  is specified by a function **bisimrel** $_O$  which computes  $\mathcal{R}_O$  from two control flow graphs with blocks and the result of data flow analysis.

$$\text{bisimrel}_O : \mathbf{Program}'' \times \mathbf{Program}'' \times \mathbf{DataFlowAnalysisResult}_O \rightarrow \mathbf{BisimulationRelation}$$

3. The specification of an optimization correctness criterion  $TCC_O$ , whose definition is based on the notion of a translation relation between IL'' programs

$$TCC_O : \mathbf{Program}'' \times \mathbf{Program}'' \times \mathbf{DataFlowAnalysisResult}_O \rightarrow \mathbf{Bool}$$

where an expression  $TCC_O((S, B_S), (T, B_T), \mathcal{A}_O)$  denotes that control flow graphs with blocks  $(S, B_S)$  and  $(T, B_T)$  are in a translation relation which is a function of the optimization  $O$  and the data flow analysis result  $\mathcal{A}_O$ .

4. The verification of the criterion  $\text{TCC}_O$ : To verify the correctness of the criterion  $\text{TCC}_O$ , Layer 5 provides a proof of a  $\text{TCC}_O$  criterion correctness theorem which, informally, says the following:
- if two control flow graphs with blocks  $(S, B_S)$  and  $(T, B_T)$  and a data flow analysis result  $\mathcal{A}_O$  fulfill the optimization correctness criterion  $\text{TCC}_O$ , then block-wise executions of  $S$  and  $T$  bisimulate w.r.t. the bisimulation relation defined by the predicate  $\text{bisimrel}_O$  for the optimization  $O$ .*

Using our notation, we can express this statement concisely as follows.

$$\begin{aligned} & \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \\ & \text{TCC}_O((S, B_S), (T, B_T), \mathcal{A}_O) \\ \implies & \\ & \text{bisimulation}(S, T, B_S, B_T, \text{bisimrel}_O((S, B_S), (T, B_T), \mathcal{A}_O)) \end{aligned}$$

5. The proof of an optimization correctness theorem whose statement is specific to the optimization  $O$  which, informally, says the following:
- if two control flow graphs with blocks  $(S, B_S)$  and  $(T, B_T)$  and a data flow analysis result  $\mathcal{A}_O$  fulfill the optimization correctness criterion  $\text{TCC}_O$ , then  $S$  and  $T$  fulfill the translation correctness predicate  $\text{corrTrans}$  provided by the translation contract layer.*

Using our notation, we can express this statement concisely as follows.

$$\begin{aligned} & \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \\ & \text{TCC}_O((S, B_S), (T, B_T), \mathcal{A}_O) \tag{I} \\ \implies & \\ & \text{corrTrans}(S, T) \end{aligned}$$

The proof of Theorem (I) is straightforward as its statement is the result of conjoining the optimization independent translation correctness theorem provided by Layer 4 and the  $\text{TCC}_O$  criterion correctness theorem provided by Layer 5 in 4.

Theorem (I) is directly applicable in proof scripts generated by the compiler: Let us assume that the optimization  $O$  transforms an IL program  $S$  in a program  $T$ . Thus, given that our SVF provides proof tactics allowing for proving statements of the form  $\text{wtp}(P, \Phi)$ ,  $\text{wfB}(P, B)$ , and  $\text{TCC}_O((S, B_S), (T, B_T), \mathcal{A}_O)$ , then generating an Isabelle/HOL theory with a proof that  $S$  and  $T$  are semantically equivalent, is straightforward. The compiler merely has to generate an Isabelle/HOL theory with

1. constant definitions of  $S, T, B_S, B_T, \mathcal{A}_O, \Phi_S, \Phi_T$ ,
2. proofs that programs  $S$  and  $T$  are well-typed w.r.t. program types  $\Phi_S$  and  $\Phi_T$ ,
3. proofs that the CFGB declarations  $B_S$  and  $B_T$  are well-formed w.r.t.  $S$  and  $T$ , respectively,
4. proof of the predicate  $\text{TCC}_O((S, B_S), (T, B_T), \mathcal{A}_O)$
5. proof of the translation correctness predicate  $\text{corrTrans}(S, T)$  which, in the first step, applies Theorem (I) and in the second step discharges its assumptions using the results in 2., 3., and 4..

Finally, it should be noted that the above corollary defines a particular proof script layout which is used by the proof generation unit of our compiler.

### 3.7 Layer 6: Proof environments specific to particular proof tasks

Layer 6 provides lemmas and proof tactics which are applied in proof scripts generated by our compiler. In our implementation of the SVF, the content of this layer is the most technical one as

the implementation of the proof tactics provided by this layer uses the programming interface of the theorem prover Isabelle/HOL and the lemmas proved in this layer do not extend the generic framework set up in the lower layers but are merely used by the proof tactics implemented in this layer as a kind of "glue code" lemmas.

In the following, we discuss two issues concerning proof tactics: Firstly, in interactive theorem proving using Isabelle/HOL, the proof tactics are usually applied as procedures searching for the proof of a statement in the proof state space and sometimes they fail to find the proof. Below, we show why computing proof tactics in our SVF is a decidable problem. Secondly, we sketch the idea of the algorithm, which is used to compute proof tactics.

From the technical point of view, proof tactics in Isabelle/HOL are higher-order ML functions and, to put it simply, they function similarly to transition functions in the operational semantics of programming languages: they take a proof state in the form of a stack of proof goals and compute the successor proof state in which the top element of the goal stack is replaced by a sequence of proof goals. Further, Isabelle/HOL provides tacticals which are ML constructs allowing for combining tactics into new tactics. For instance, Isabelle/HOL provides a tactical `EVERY` which takes a list of tactics  $[t_0, \dots, t_n]$  and combines them into a tactic which takes a proof state and computes the successor proof state which is the result of successive applications of the tactics from the list  $[t_0, \dots, t_n]$ . Using the above analogy to the operational semantics, we can say that if application of a tactic to an initial proof state corresponds to application of a transition function to the initial state of execution for a program, then application of tactic `EVERY`  $[t_0, \dots, t_n]$  to that proof state corresponds to partial execution of a program which starts from the initial state of execution for that program. As the compiler generates proof scripts in which programs are represented as constants and all lemmas are statements with those constants as parameters, the compiler can, for each lemma in the script, deterministically compute a tactic of the form `EVERY`  $[t_0, \dots, t_n]$  which proves the lemma.

As aforementioned at the end of the previous section, Layer 4 and 5 provide a set of translation correctness theorems which give raise to a uniform proof script layout for each optimization performed by the compiler. For example, if we want to generate proof scripts with proofs of the statements of the form `corrTrans`( $S, T$ ) and we want to apply Theorem (I) from the previous section, then the layout of such proof scripts consists of the following parts:

- Part 1.:* provides constant definition of a source program  $S$ ,
- Part 2.:* provides constant definition of a target program  $T$ ,
- Part 3.:* provides constant definition of a program type  $\Phi_S$ ,
- Part 4.:* provides constant definition of a program type  $\Phi_T$ ,
- Part 5.:* provides constant definition of a CFGB declaration  $B_S$ ,
- Part 6.:* provides constant definition of a CFGB declaration  $B_T$ ,
- Part 7.:* provides constant definition of a result of data flow analysis  $\mathcal{A}_O$ ,
- Part 8.:* provides proof of the predicate `wtp`( $S, \Phi_S$ ),
- Part 9.:* provides proof of the predicate `wtp`( $T, \Phi_t$ ),
- Part 10.:* provides proof of the predicate `wfB`( $S, B_S$ ),
- Part 11.:* provides proof of the predicate `wfB`( $T, B_T$ ),
- Part 12.:* provides proof of the predicate `TCCO`( $(S, B_S), (T, B_T), \mathcal{A}_O$ )
- Part 13.:* provides proof of the predicate `corrTrans`( $S, T$ )

Using this property, we implemented, for each optimization  $O$  and each lemma  $l_n$  which has to be proved in a proof script which was generated to certify a concrete result of application of  $O$ , a parameterized tactic, i.e. an ML function of the form

```
fun f_n x_1 ... x_{n-1} = let ... in EVERY [f_1(x_1), ..., f_{n-1}(x_{n-1})] end
```

which proves the lemma  $l_n$ , if the values of the parameters  $x_0, \dots, x_{n-1}$  are properly set. The parameterized tactics have two kinds of parameters:

1. parameters which are identifiers of constants defined in the proof script, e.g. constants representing IL programs or constants representing the results of data flow analysis, and
2. parameters which are lemma names of lemmas which have been proven earlier in the script.

Thus, if Layer 6 provides, for each optimization phase, an ML structure with signatures of parameterized tactics which are needed to prove all lemmas in proof scripts generated to certify that optimization phase, then generation of proof scripts for an optimization phase is straightforward: After an optimization is completed, the compiler passes data structures representing the source and the target programs; and the result of data flow analysis performed prior to the optimization to a proof generation unit. The proof generation unit generates an abstract syntax tree which describes the proof script layout: The leaves of tree, which are records with fields defining proper values of the actual parameters in calls to parameterized tactics, like lemma names, program identifiers, etc. are unparsed to strings representing lemmas and calls to parameterized tactics which prove that lemmas. The nodes of the tree are interpreted as string concatenation operators. The proof generation unit unparses the tree in the same manner as the standard compiler does during the code generation phase.

As future implementations Layer 6 will vary depending of the theorem prover used to implement the SVF and the content of the SVF itself, explaining the implementation of proof tactics and "glue code" lemmas is beyond of the scope of this thesis. In Section 3.8, we merely give a small example which illustrates the purpose of "glue code" lemmas in our SVF.

Finally, we note that the content of this layer does not increases the size of the TCB in our FTV system. If a proof tactic has a bug, then a call to this tactic can merely result in a false alarm, i.e. in a situation that the proof checker rejects that tactic due to its inability to prove a statement of interest although it is provably valid.

### 3.8 Proof checking: an example

As mentioned in Section 1.5.1, the theory files generated by our compiler can be either inspected by the human and verified interactively using the theorem prover Isabelle/HOL or Isabelle/HOL can be used as a proof checker in batch mode. This section illustrates the purpose of "glue code" lemmas in our SVF and the work-flow of proof checking in batch mode by small example. In the example, we use the Isabelle/HOL syntax.

*Example 3.1.* Let us consider again the work-flow of our compiler, cf. Figure 1.5. It follows from the definition of `corrTrans` that proof of `corrTrans(IL0, IL5)` would be straightforward, if we proved intermediate optimizations *CF*, *DAE*, *NI*, *RAI*, and *RAE* correct and if we had the following "glue code" lemma which would allow us to derive `corrTrans(IL0, IL5)` from intermediate correctness results directly in one proof step.

**Lemma 3.2.** (*"Glue code" lemma from Layer 6*)

```
[| corrTrans(P0,P1);
   corrTrans(P1,P2);
   corrTrans(P2,P3);
   corrTrans(P3,P4);
   corrTrans(P4,P5)
|]
==>
corrTrans(P0,P5)
```

□

The proof of this lemma is trivial as by the definition of `corrTrans` we have to derive the conclusion `Sem(P0) = Sem(P5)` from the following premises

- `Sem(P0) = Sem(P1)`,
- `Sem(P1) = Sem(P2)`,
- `Sem(P2) = Sem(P3)`,
- `Sem(P3) = Sem(P4)`, and
- `Sem(P4) = Sem(P5)`.

It should be noted that the root theory `Main` in the theorem prover Isabelle/HOL already contains the lemma `trans`:

```
[| P = Q; Q = R |] ==> P = R
```

which we could apply four times to derive the equation `corrTrans(IL0,IL5)` in our example. Nevertheless, the statement of Lemma 3.2 reflects a fact about the layout of our proof scripts which is program independent, namely, that we always prove the optimizations *CF*, *DAE*, *NI*, *RAI*, and *RAE* in corresponding proof scripts and then show the equation `corrTrans(IL0,IL5)` as a direct consequence of the results proved in those proof scripts. Omitting "glue code" lemmas would lead to complex and cluttered proof tactics.

In the following, we demonstrate how Lemma 3.2 is applied during the proof checking. In our example, we assume that our compiler generated all proof scripts with Isabelle/HOL theories as visualized in Figure 1.5, i.e. it generated the following files:

1. a file `CF.thy` with the theory containing a proof of the lemma

**Lemma 3.3.** `corrTrans(IL0,IL1)`

□

2. a file `DAE.thy` with the theory containing a proof of the lemma

**Lemma 3.4.** `corrTrans(IL1,IL2)`

□

3. a file `NI.thy` with the theory containing a proof of the lemma

**Lemma 3.5.** `corrTrans(IL2,IL3)`

□

4. a file `RAI.thy` with the theory containing a proof of the lemma

**Lemma 3.6.** `corrTrans(IL3,IL4)`

□

5. a file `RAE.thy` with the theory containing a proof of the lemma

**Lemma 3.7.** `corrTrans(IL4,IL5)`

□

6. a file `Root.thy` with the root theory containing the preamble

```
theory Root = CF + DAE + NI + RAI + RAE:
```

and a file `Root.ML` with a proof of the main lemma

```
"corrTrans(IL0,IL5)";
```

In the following, we describe the proof checking steps, which are made by the proof checker in our scenario:

1. step: The file `Root.thy` is given as input to the theorem prover Isabelle/HOL.
2. step: The preamble in the `Root.thy` causes the theorem prover to read files `CF.thy`, `DAE.thy`, `NI.thy`, `RAI.thy`, and `RAE.thy`; and to check proofs of Lemmas 3.3, 3.4, 3.5, 3.6, and 3.7. Once the proofs are verified, the theorem prover extends the theory `Root` by these lemmas.
3. step: As the theory file contains no constant definitions, the Isabelle/HOL starts to read the proof script `Root.ML`.
4. step: The theorem prover encounters the statement of the main lemma in the theory `Root`

```
Goal "corrTrans(IL0,IL5)";
```

parses the statement, and starts checking the proof of the lemma. The proof of the lemma consists of a single tactic call as follows, see the explanation below.

```
by (Root.main_lemma_tac "Lemma 1.3" "IL1"
    "Lemma 1.4" "IL2"
    "Lemma 1.5" "IL3"
    "Lemma 1.6" "IL4"
    "Lemma 1.7");
```

The line of the form `by (S.t)`; is interpreted by the theorem prover as a call of a tactic `t` which is defined in the structure `S`. The tactic `main_lemma_tac` is defined as follows (we omit some technical details here).

```
structure Root =
struct
...

fun main_lemma_tac s1 p1 s2 p2 s3 p3 s4 p4 s5 =
  EVERY [res_inst_tac [("P1",p1),("P2",p2),("P3",p3),("P4",p4)]
        (thm "Lemma 1.2") 1,
        resolve_tac (thm s1) 1,
        resolve_tac (thm s2) 1,
        resolve_tac (thm s3) 1,
        resolve_tac (thm s4) 1,
        resolve_tac (thm s5) 1
        ];
...
end
```

The call to the tactic `Root.main_lemma_tac` results in application of the tactic

```
EVERY [res_inst_tac [("P1","IL1"),("P2","IL1"),("P3","IL3"),("P4","IL4")]
        (thm "Lemma 1.2") 1,
        resolve_tac (thm "Lemma 1.3") 1,
        resolve_tac (thm "Lemma 1.4") 1,
```

```

      resolve_tac (thm "Lemma 1.5") 1,
      resolve_tac (thm "Lemma 1.6") 1,
      resolve_tac (thm "Lemma 1.7") 1
    ]

```

to the current state of the proof. Application of the first tactic from the list to the conclusion of the main lemma (proof in backward fashion), where the free variables of the lemma: P1, P2, P3, and P4, are instantiated with the constant identifiers of constants representing programs in the example, results in splitting of the current proof state in five subgoals which are pushed on the stack.

1. subgoal: corrTrans(IL0,IL1)
2. subgoal: corrTrans(IL1,IL2)
3. subgoal: corrTrans(IL2,IL3)
4. subgoal: corrTrans(IL3,IL4)
5. subgoal: corrTrans(IL4,IL5)

Then, successive application of the remaining tactics from the list results in successive popping the top elements from this stack as the semantics of a tactic of the form `resolve_tac (thm lemmaname)` is that Isabelle/HOL applies the lemma `lemmaname` to the top element of the stack and removes that element from the stack. When the list of tactics is completely proceed, the stack is empty and thus the proof of the main lemma is successfully checked.

◇

The example illustrates

- how the generated proof scripts are checked,
- how the proofs in our SVF are structured in program dependent and program independent parts using "glue code" lemmas and the Isabelle/HOL tacticals.
- how the proof script layout and "glue code" lemmas are adjusted in a way which allows for generating proof scripts realized as unparsing of abstract syntax trees in prefix order.
- that the computation of a tactic proving a lemma is realized as visiting in infix order nodes of an imaginary proof tree, which corresponds to the proof of that lemma.

## Chapter 4

---

### Translation contract

This section presents the content of Layer 2 of our implementation of the SVF. In Section 3.3, we motivated that the general purpose of Layer 2 is to provide the formalization of a translation contract for translations from a source into a target language programs. Also there, we explained what requirements have to be fulfilled by a translation contract provided by a SVF which follows the FTV approach and mentioned that, as we are interested in translation certificates attesting to the optimization correctness, we formalized a special case of the translation contract in which

1. the source language and the target language are equal,
2. the program semantics functions are total functions, and
3. the relation between the sets of meanings of programs, which is used in the definition of the translation correctness predicate is defined as the equality.

In particular, our translation contract comprises the following:

- The formalization of an intermediate language IL which includes
  - The definition of the abstract syntax of the language IL.
  - The definition of the program semantics function for the language IL.
- The formalization of a translation correctness predicate `corrTrans` on two IL programs which checks if their program semantics are equal.

The rest of this section is organized as follows. The formalization of IL is presented in Sections 4.1 and 4.2 that present the abstract syntax of the language IL and the definition of the program semantics function, respectively. Section 4.3 presents the specification of the translation correctness predicate.

#### 4.1 Abstract syntax of IL

This section presents abstract syntax of the language IL. IL is a small intermediate language of goto programs which provides, together with its formally defined semantics, a basis for the optimization phase of our compiler. As is standard, we present the syntax in adherence to the presentation style given by Winskel in [120].

We begin the presentation by listing the syntactic sets associated with IL:

- finite set of identifiers **Variable**,
- natural numbers **Nat**, consisting of positive integers with zero,
- numbers **Int**, consisting of positive and negative integers with zero,
- truth values **Bool** = {`true`, `false`}
- types **Type**,

- values **Value**,
- indexes **Index**,
- operands **Operand**,
- l-values **LValue**,
- expressions **Expression**,
- variable declarations **VariableDecl**,
- lists of variable declarations **VariableDeclList**,
- instructions **Instruction**,
- lists of instructions **InstructionList**,
- program declarations **ProgramDecl**, and
- programs **Program**.

Before we describe how the syntactic sets are built-up, we define metavariables ranging over these sets and notational conventions we follow in this thesis:

- Metavariables are represented by lower-case letters,
- $a, v$  range over identifiers **Variable**,
- $n$  ranges over natural numbers **Nat**,
- $i$  ranges over numbers **Int**,
- $b$  ranges over truth values **Bool**,
- $\tau$  ranges over types **Type**,
- $val$  ranges over values **Value**,
- $idx$  ranges over indexes **Index**,
- $o$  ranges over operands **Operand**,
- $lv$  ranges over l-values **LValue**,
- $vd$  ranges over variable declarations **VariableDecl**,
- $vds$  ranges over lists of variable declarations **VariableDeclList**,
- $instr$  ranges over instructions **Instruction**,
- $instrs$  ranges over lists of instructions **InstructionList**,
- $Pdcl$  ranges over program declarations **ProgramDecl**,
- $S, T, P$  range over programs **Program**,
- The metavariables we use to range over the syntactic categories can be primed or subprimed,
- $(s, s), (s, s, s), \dots$  range over n-tuples of syntactic structures  $s$ , e.g.  $(b, b, b)$  ranges over product set **Bool**  $\times$  **Bool**  $\times$  **Bool**,
- $[s, \dots, s]$  ranges over lists of syntactic structures  $s$ , e.g.  $[b, \dots, b]$  ranges over lists of truth values.

Definition 4.1 gives formation rules for the syntactic sets. An IL program  $P$  is a pair  $(Pdcl, I)$  consisting of a program declaration  $Pdcl$  and an input  $I$ . A program declaration is a pair  $(vds, instrs)$  consisting of a list of variable declarations  $vds$  and a list of instructions  $instrs$ . A variable declaration is a pair  $(v, \tau)$  consisting of an identifier  $v$  and a type  $\tau$ . Each element of a variable declaration list  $vds$ ,  $(v, \tau)$ , indicates that the declared type of identifier  $v$  is  $\tau$ . A declared type of an identifier can be either boolean type `bool`, integer type `int`, boolean array type `barray( $n$ )`, or integer array `ibarray( $n$ )`. In an array declaration, the value of the parameter  $n$  indicates that its declared array type comprises only lists with  $n$  elements. The instruction set **Instruction** comprises five commands:

1. assignment  $lv := e$ ,
2. print integer `printi( $e$ )`,
3. conditional branch `branch( $e, n$ )`,
4. unconditional branch `goto( $n$ )`, and

5. exit exit.

An input  $I$  for a program  $P = (Pdcl, I)$  is a list of pairs  $(v, val)$ , where  $v$  is an identifier and  $val$  is a value.

**Definition 4.1.**

	$v, a \in \mathbf{Variable} = \{v_0, v_1, \dots, v_n, a_0, a_1, \dots, a_m\}$
<b>Type</b> $\ni \tau$	$::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{barray}(n) \mid \mathbf{iarray}(n)$
<b>IntArray</b> $\ni il$	$::= [] \mid i\#il$
<b>BoolArray</b> $\ni bl$	$::= [] \mid b\#bl$
<b>Value</b> $\ni val$	$::= b \mid i \mid il \mid bl$
<b>Index</b> $\ni idx$	$::= i \mid v$
<b>Operand</b> $\ni o$	$::= i \mid b \mid v \mid a[idx]$
<b>Expression</b> $\ni e$	$::= o \mid o_1 + o_2 \mid o_1 - o_2 \mid o_1 * o_2 \mid -o \mid$ $o_1 \wedge o_2 \mid \neg o \mid o_1 \vee o_2 \mid$ $o_1 = o_2 \mid o_1 \neq o_2 \mid o_1 < o_2 \mid o_1 \leq o_2$
<b>LValue</b> $\ni lv$	$::= v \mid a[idx]$
<b>Instruction</b> $\ni instr$	$::= lv := e \mid \mathbf{printi}(e) \mid \mathbf{branch}(e, n) \mid$ $\mathbf{goto}(n) \mid \mathbf{exit}$
<b>InstructionList</b> $\ni instrs$	$::= [] \mid instr\#instrs$
<b>VariableDecl</b> $\ni vd$	$::= (v, \tau)$
<b>VariableDeclList</b> $\ni vds$	$::= [] \mid vd\#vds$
<b>Input</b> $\ni I$	$::= [] \mid (v, val)\#I$
<b>ProgramDecl</b> $\ni Pdcl$	$::= (vds, instrs)$
<b>Program</b> $\ni P, S, T$	$::= (Pdcl, I)$

## 4.2 Semantics of IL

This section presents a formal semantics definition of a language IL. The section is divided in three parts which are organized as follows. The first part of the section introduces syntactic sets and metavariables associated with the semantics definition of IL. The second part of the section presents the definition of the operational semantics of IL. The last part of the section presents the definition of a program semantics function for the language IL.

We begin the presentation by listing the syntactic sets associated with the semantics of IL:

- program termination flag values  
**TerminationFlag** = {T, NT},
- array-index-out-of-bounds exception flag values  
**ArrayIndexFlag** = {AB<sub>ok</sub>, AB},
- program counters **ProgramCounter**,
- output buffers **OutputBuffer**,
- states of computations **State**, and
- configurations, i.e. states of program executions, **Configuration**;

and define metavariables ranging over these sets:

- $tf$  ranges over program termination flags **TerminationFlag**,
- $af$  ranges over array-index-out-of-bounds exception flags **ArrayIndexFlag**,
- $pc$  ranges over program counters **ProgramCounter**,
- $b$  ranges over buffers **OutputBuffer**,
- $s$  ranges over states of computations **State**, and

- $\sigma$  ranges over configurations **Configuration**.

Definition 4.2 gives formation rules for the syntactic set **Configuration**. We use elements of the **Configuration** set to model states of executions IL programs. A configuration  $\sigma \in \mathbf{Configuration}$  is a tuple  $(tf, af, pc, b, s)$  which we consider as a result of partial execution of a program. The components of  $\sigma$  have their meanings as follows.

- The value of the flag  $tf \in \mathbf{TerminationFlag}$  indicates whether a program terminated,  $tf = T$ , or not,  $tf = NT$ .
- The value of the flag  $af \in \mathbf{ArrayIndexFlag}$  indicates whether execution of the last preceding instruction caused an array-index-out-of-bounds exception,  $af = AB$ , or not,  $af = AB_{ok}$ .
- The meaning of the program counter  $pc$  is standard. Its value indicates which instruction of an executed program will be fetched as next.
- The set **OutputBuffer** models a set of states of an output buffer. During execution of a program, a buffer  $b \in \mathbf{OutputBuffer}$  can be put into a state whose form is either  $\mathbf{FLUSH}(n)$  or  $\mathbf{WRITE}(n, i)$ . The meaning of a state of the buffer component  $b$  is always related to a partial execution that produced a configuration  $\sigma$  comprising  $b$ .
  1. The meaning of a state of the form  $\mathbf{FLUSH}(n)$  is the following:
    - a) The last preceding executed instruction of the partial execution producing  $\sigma$  was a non-printing instruction.
    - b)  $n$  integers have been written into the buffer during the partial execution producing  $\sigma$ .
  2. The meaning of a state of the form  $\mathbf{WRITE}(n, i)$  is the following:
    - a) The last preceding executed instruction of the partial execution producing  $\sigma$  was a `printi` instruction.
    - b)  $n$  integers have been written into the buffer during the partial execution producing  $\sigma$ .
    - c) The  $i$  was  $n$ -th integer written into the buffer.

Thus, the value of the buffer component  $b$  in a configuration describes observable part of program behavior resulting from execution of the last preceding instruction.

- We use elements of the set **State** to model states of computations resulting from all partial executions of IL programs. A state  $s \in \mathbf{State}$  is a partial mapping from identifiers to values. In an initial configuration  $\sigma_0$  for a program  $P = (Pdcl, I)$ , the domain of the component  $s_0$  is a function of the input  $I = [(v_0, val_0), \dots, (v_n, val_n)]$ ,  $\text{dom}(s_0) = \text{set}(\text{map fst } I)$ , and it remains unchanged for all partial executions of  $P$ . The value of the state component  $s$  in a configuration belongs to a non-observable part of program behavior.

#### Definition 4.2.

$\mathbf{TerminationFlag} \ni tf ::= T \mid NT$ $\mathbf{ArrayIndexFlag} \ni af ::= AB_{ok} \mid AB$ $\mathbf{OutputBuffer} \ni b ::= \mathbf{FLUSH}(n) \mid \mathbf{WRITE}(n, i)$ $pc \in \mathbf{ProgramCounter} = \mathbf{Nat}$ $s \in \mathbf{State} = \mathbf{Variable} \rightsquigarrow \mathbf{Value}$ $\mathbf{Configuration} \ni \sigma ::= (tf, af, pc, b, s)$
--

In the following, we present the definitions of functions which constitute the operational semantics of IL.

Definition 4.3 defines the function `evalo` which evaluates an operand  $o$  in the context of a state of computation  $s$ , `evalo(o, s)`. According to the definition, an operator  $o$  is evaluated in a context of a state of computation  $s$ . The result of the evaluation is a tuple of the form  $(af, oval)$  whose components denote the following: The value of the component  $af$  is either equal  $AB_{ok}$  or equal  $AB$ .

A result tuple with the component  $af$  set to  $\text{AB}_{\text{ok}}$  indicates that evaluation of  $o$  caused no array-index-out-of-bounds exception. Otherwise, it is set to  $\text{AB}$ . The component  $af$  is relevant only when considering evaluation results of indexed operators, i.e. operators having either the form  $a[i]$  or the form  $a[v]$ . As for the operators of the former form, the definition of  $\text{evalo}$  requires that  $a$  must be either a well-defined integer list or a well-defined boolean list in  $s$  and that  $i$  is a valid index of that list. As for the operators of the latter form, the definition requires additionally that  $v$  must be a well-defined integer in  $s$  that is a valid index of the list. The value of the component  $oval$  is either of the form  $\text{Some}(val)$  or is equal  $\text{None}$ . A result tuple with the component  $oval$  set to a value of the form  $\text{Some}(val)$  indicates that all identifiers of  $o$  are well-defined in  $s$ . Otherwise, it is set to  $\text{None}$ . The component is not relevant when considering evaluation results of constant operands, i.e. operands having the form either  $i$  or  $b$ . Further, the reader should note that according to the definition of  $\text{evalo}$  the tuples resulting from evaluation of indexed operands,  $\text{evalo}(a[i], s)$  and  $\text{evalo}(a[v], s)$ , can not be of the form  $(\text{AB}_{\text{ok}}, \text{None})$  which is in contrast to evaluation of variable operands. Section 5.4 presents a type safety theorem which we proved for the language IL that captures the issue of operand evaluation too: Applying the theorem, we proved a corollary for each operand variant containing variables, i.e.  $v$ ,  $a[i]$ , and  $a[v]$ , that says the following: if a program is well-formed and the operand variant is in the set of operands of the program then all variables of the operands are well-defined in a context of a current state of computation of the program and evaluation of the operand w.r.t. the state yields a result tuple that has one of two forms only: either  $(\text{AB}_{\text{ok}}, \text{Some}(val))$  or  $(\text{AB}, \text{None})$ .

**Definition 4.3.**

**OptionalValue**  $\ni oval ::= \text{Some}(val) \mid \text{None}$

**evalo** : **Operand**  $\times$  **State**  $\longrightarrow$  **ArrayIndexFlag**  $\times$  **OptionalValue**

$\text{evalo}(i, s) = (\text{AB}_{\text{ok}}, \text{Some}(i))$

$\text{evalo}(b, s) = (\text{AB}_{\text{ok}}, \text{Some}(b))$

$\text{evalo}(v, s) = (\text{AB}_{\text{ok}}, s(v))$

$$\text{evalo}(a[i], s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(ill!i)) & \text{if } s(a) = \text{Some}(il) \wedge 0 \leq i < \text{length}(il), \\ (\text{AB}_{\text{ok}}, \text{Some}(bll!i)) & \text{if } s(a) = \text{Some}(bl) \wedge 0 \leq i < \text{length}(bl), \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases}$$

$$\text{evalo}(a[v], s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(ill!i)) & \text{if } s(a) = \text{Some}(il) \wedge s(v) = \text{Some}(i) \wedge 0 \leq i < \text{length}(s(a)), \\ (\text{AB}_{\text{ok}}, \text{Some}(bll!i)) & \text{if } s(a) = \text{Some}(bl) \wedge s(v) = \text{Some}(i) \wedge 0 \leq i < \text{length}(s(a)), \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases}$$

Definition 4.4 defines a function  $\text{evale}$  which evaluates an expression  $e$  in the context of a state of computation  $s$ ,  $\text{evale}(e, s)$ . The evaluation of expressions is similar to the evaluation of operands in the following sense. Firstly, an expression  $e$  is evaluated in a context of a state of computation  $s$ . Secondly, evaluation of  $e$  w.r.t.  $s$ ,  $\text{evale}(e, s)$ , yields a tuple  $(af, oval)$  which is an encoding of the result. The meaning of the tuple is the same as by tuples resulting from evaluation of operands. Thirdly, the definition of  $\text{evale}$  requires that types of operand evaluation results conform with the type of evaluated expression. Here, types play an even more important role than it was the case by the evaluation of operands themselves as they were checked there only during evaluation of indexed operators. For example, the definition of  $\text{evale}$  requires that in order to evaluate on expression  $o_1 + o_2$  w.r.t. a state the results of evaluations of operands  $o_1$  and  $o_2$  w.r.t. to the state must be well-defined integer values.

**Definition 4.4.**

$$\begin{aligned}
& \text{evale} : \text{Expression} \times \text{State} \longrightarrow \text{ArrayIndexFlag} \times \text{OptionalValue} \\
& \text{evalo}(o, s) = (\text{evalo}(o, s)) \\
& \text{evale}(o_1 + o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(i_1 + i_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_2)), \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 - o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(i_1 - i_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_2)), \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 * o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(i_1 * i_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_2)), \\ (\text{AB}, \text{None}), & \text{otherwise} \end{cases} \\
& \text{evale}(-o, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(-i)) & \text{if } \text{evalo}(o, s) = (\text{AB}_{\text{ok}}, \text{Some}(i)), \\ (\text{AB}, \text{None}), & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 \wedge o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(b_1 \wedge b_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(b_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(b_2)) \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(\neg o, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(\neg b)), & \text{if } \text{evalo}(o, s) = (\text{AB}_{\text{ok}}, \text{Some}(b)), \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 \vee o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(b_1 \vee b_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(b_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(b_2)) \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 = o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(i_1 = i_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_2)) \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 \neq o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(i_1 \neq i_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_2)) \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 < o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(i_1 < i_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_2)) \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases} \\
& \text{evale}(o_1 \leq o_2, s) = \begin{cases} (\text{AB}_{\text{ok}}, \text{Some}(i_1 \leq i_2)) & \text{if } \text{evalo}(o_1, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_1)) \wedge \\ & \text{evalo}(o_2, s) = (\text{AB}_{\text{ok}}, \text{Some}(i_2)) \\ (\text{AB}, \text{None}) & \text{otherwise} \end{cases}
\end{aligned}$$

Definition 4.5 defines an instruction execution function  $\text{execi}$  which computes a successor configuration  $\text{execi}(pc, b, s, instr)$  for an instruction  $instr$  in the context of a program counter  $pc$ , an output buffer  $b$ , a state of computation  $s$ . The values of  $pc$ ,  $b$ , and  $s$  model component values of a configuration  $\sigma_n$  whose components  $tf$  and  $af$  are set to  $\text{NT}$  and  $\text{AB}_{\text{ok}}$ , respectively, i.e. a configuration that is a result of an exception-free partial execution of a program, and the result of execution of  $instr$  is a configuration that corresponds to successor configuration of  $\sigma_n$ ,  $\sigma_{n+1}$ , in the execution of the program. In the following, we call configurations produced by exception-free partial executions the error-free ones and configurations produced by partial executions which raised an array-index-out-of-bounds exception the erroneous ones. As execution of an instruction makes only sense in a context of an error-free configuration and executing of an instruction yields a

successor configuration, `execi` computes a successor configuration doing without values of the flags `tf` and `af`. In the following, we discuss instruction evaluation rules for each instruction separately.

*Evaluation of assignment instructions.* Execution of an assignment does not terminate execution of a program. Because of this, `execi` always returns a successor configuration with the value of the termination flag `tf` set to `NT`. Further, execution of an assignment a program does not modify observable behavior of the program. On the other hand, `execi` has to take into account that the last preceding executed instruction could possibly have modified observable behavior of the program by writing an integer into the buffer and that the state of the buffer can have the form `WRITE( $n, i$ )`. For this reason, `execi` always returns a configuration with the buffer component flushed, `flush( $b$ )`. The values of remaining components in the successor configuration depend on the result of evaluation of the assignment's expression in the following way: If evaluation of the expression yields a well-defined value then `execi` returns an error-free configuration with values of exception flag `af`, program counter `pc`, and state of computation `s` set to `ABok`, `Suc( $pc$ )`, and properly updated `s`, respectively. Otherwise, `execi` returns an erroneous configuration with `af` set to `AB` and unchanged `pc` and `s`.

*Evaluation of printi instructions.* Execution of a `printi` instruction of a program neither terminates execution of the program nor updates current state of computation but it does modify observable behavior of the program. For this reason, `execi` computes values of termination flag `tf`, exception flag `af`, and program counter `pc` in a successor configuration in the same way as in the case of the assignment instruction and leaves the state of computation unchanged. The modification of observable behavior is done by updating the state of output buffer `b`. Here, `execi` must take into account two possible cases of the last preceding executed instruction:

1. The last preceding executed instruction has been a non-`printi` instruction and the buffer is flushed, i.e. it has the form `FLUSH( $n$ )`. In this case, `execi` evaluates `printi`'s expression to an integer value, writes the value into the buffer, and increments the buffer counter. The last two steps are modeled by the buffer state `WRITE( $Suc(n), i$ )` in the successor configuration.
2. The last preceding executed instruction has been a `printi` instruction and the buffer non-empty, i.e. it has the form `WRITE( $n, i$ )`. In this case, `execi` evaluates `printi`'s expression to an integer value, flushes the buffer, writes the value into the buffer, and increments the buffer counter. The last three steps are modeled by the buffer state `WRITE( $Suc(n), i$ )` in a successor configuration.

*Evaluation of branch instruction.* Execution of a branch instruction of a program neither terminates execution of the program nor updates the current state of computation nor modifies observable behavior of the program. For this reason, `execi` computes values of termination flag `tf`, exception flag `af`, buffer state `b` in a successor configuration in the same way as in the assignment instruction case and leaves state of computation `s` unchanged. Computation of the value of program counter is standard. `execi` evaluates `branch`'s expression to a boolean value `b` and sets program counter to either `Suc( $pc$ )` or to `n` according to the value of `b`.

*Evaluation of goto instructions.* Execution of a `goto` instruction of a program neither terminates execution of the program nor updates current state of computation nor modifies observable behavior of the program nor can produce an erroneous successor configuration. For this reason, `execi` computes values of termination flag `tf`, exception flag `af`, buffer state `b`, and state of computation `s` in a successor configuration in the same way as in the branch instruction case. Computation of the value of program counter is standard. `execi` sets program counter to the value of `goto`'s unconditional destination `n`.

*Evaluation of exit instructions.* Execution of an exit instruction of a program flushes output buffer and terminates execution of the program. The values of other components of a successor configuration are carried from the predecessor configuration over.

Summarizing the above, an erroneous successor configuration  $\sigma_{n+1}$  arises from an error-free predecessor configuration  $\sigma_n$  by flushing output buffer in  $\sigma_n$  and "freezing" the remaining components of  $\sigma_n$ , i.e. carrying the components from  $\sigma_n$  to  $\sigma_{n+1}$  over.

**Definition 4.5.**

$$\begin{aligned}
& \text{flush} : \mathbf{OutputBuffer} \rightarrow \mathbf{OutputBuffer} \\
& \text{flush}(\text{FLUSH}(n)) = \text{FLUSH}(n) \\
& \text{flush}(\text{WRITE}(n, i)) = \text{FLUSH}(n) \\
\\
& \text{execi} : \mathbf{ProgramCounter} \times \mathbf{OutputBuffer} \times \mathbf{State} \times \mathbf{Instruction} \rightarrow \mathbf{Configuration} \\
& \text{execi}(pc, b, s, v := e) = \\
& \quad \begin{cases} (\text{NT}, \text{AB}_{\text{ok}}, \text{Suc}(pc), \text{flush}(b), s(v \mapsto val)), & \text{if } \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \\ (\text{NT}, \text{AB}, pc, \text{flush}(b), s), & \text{otherwise} \end{cases} \\
& \text{execi}(pc, b, s, a[i] := e) = \\
& \quad \begin{cases} (\text{NT}, \text{AB}_{\text{ok}}, \text{Suc}(pc), \text{flush}(b), s(a \mapsto s(a)[\text{nat}(i)] := val)), & \text{if } \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \\ (\text{NT}, \text{AB}, pc, \text{flush}(b), s), & \text{otherwise} \end{cases} \\
& \text{execi}(pc, b, s, a[v] := e) = \\
& \quad \begin{cases} (\text{NT}, \text{AB}_{\text{ok}}, \text{Suc}(pc), \text{flush}(b), s(a \mapsto s(a)[\text{nat}(s(v))] := val)), & \text{if } \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \\ (\text{NT}, \text{AB}, pc, \text{flush}(b), s), & \text{otherwise} \end{cases} \\
& \text{execi}(pc, \text{FLUSH}(n), s, \text{printi}(e)) = \\
& \quad \begin{cases} (\text{NT}, \text{AB}_{\text{ok}}, \text{Suc}(pc), \text{WRITE}(\text{Suc}(n), val), s), & \text{if } \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \\ (\text{NT}, \text{AB}, pc, \text{flush}(b), s), & \text{otherwise} \end{cases} \\
& \text{execi}(pc, \text{WRITE}(n, i), s, \text{printi}(e)) = \\
& \quad \begin{cases} (\text{NT}, \text{AB}_{\text{ok}}, \text{Suc}(pc), \text{WRITE}(\text{Suc}(n), val), s), & \text{if } \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \\ (\text{NT}, \text{AB}, pc, \text{flush}(b), s), & \text{otherwise} \end{cases} \\
& \text{execi}(pc, b, s, \text{branch}(e, n)) = \\
& \quad \begin{cases} (\text{NT}, \text{AB}_{\text{ok}}, n, \text{flush}(b), s), & \text{if } \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \wedge val \\ (\text{NT}, \text{AB}_{\text{ok}}, \text{Suc}(pc), \text{flush}(b), s), & \text{if } \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \wedge \neg val \\ (\text{NT}, \text{AB}, pc, \text{flush}(b), s), & \text{otherwise} \end{cases} \\
& \text{execi}(pc, b, s, \text{goto}(n)) = (\text{NT}, \text{AB}_{\text{ok}}, n, \text{flush}(b), s) \\
& \text{execi}(pc, b, s, \text{exit}) = (\text{T}, \text{AB}_{\text{ok}}, pc, \text{flush}(b), s)
\end{aligned}$$

Definition 4.6 defines a transition definition  $\text{exec}$  which computes a successor configuration  $\text{exec}(P, \sigma)$  for an IL program  $P$  and a configuration  $\sigma$ .  $\text{exec}$  is a wrapper function that takes into account that a configuration resulting from a partial execution of  $P$  can have one of the following three forms:

1. A configuration can result from a partial program execution of  $P$  whose last preceding executed instruction has been an exit instruction. In this case, the configuration has the form  $(\text{T}, af, pc, b, s)$  as execution of exit instruction sets the value of termination flag  $tf$  to  $\text{T}$ .
2. A configuration can result from a partial execution of  $P$  with a property that execution of the last preceding executed instruction raised an array-index-out-of-bound exception. In this case the configuration is an erroneous one and it has the form  $(tf, \text{AB}, pc, b, s)$ .
3. A configuration can result from a partial execution of  $P$  whose last preceding executed instruction has been neither an exit instruction nor its execution caused an exception. In this case, the configuration is an error-free one and it has the form  $(\text{NT}, \text{AB}_{\text{ok}}, pc, b, s)$ .

In the first two cases, evaluation of the expression  $\text{exec}(P, \sigma)$  yields unchanged configuration  $\sigma$ . In other words,  $\text{exec}$  functions as projection  $\lambda(P, \sigma'). \sigma'$  passing "frozen" configuration  $\sigma$  on. In the

third case, the input configuration is an error-free one and  $\text{exec}$  computes its successor by dropping flags  $tf$  and  $af$ , which are not needed for performing the task of computation, looking up for an instruction  $\text{instrs!pc}$  to be executed as next one, and evaluating expression  $\text{execi}(pc, b, s, \text{instrs!pc})$ .

**Definition 4.6.**

$$\begin{aligned} \text{exec} &: \mathbf{Program} \times \mathbf{Configuration} \rightarrow \mathbf{Configuration} \\ \text{exec}(((vds, instrs), I), (T, af, pc, b, s)) &= (T, af, pc, b, s) \\ \text{exec}(((vds, instrs), I), (tf, AB, pc, b, s)) &= (tf, AB, pc, b, s) \\ \text{exec}(((vds, instrs), I), (NT, AB_{ok}, pc, b, s)) &= \text{execi}(pc, b, s, \text{instrs!pc}) \end{aligned}$$

Definition 4.7 defines a function  $\text{init}$  which computes an initial configuration  $\text{init}(P)$  for an IL program  $P$ . The components of an initial configuration for a program  $P$  are defined as follows.

- The value of the program termination flag is set to  $NT$  so as to indicate that execution of  $P$  is about to begin.
- The value of the array-index-out-of-boundss exception flag is set to  $AB_{ok}$  so as to indicate that no no exceptions has been raised yet.
- As the first instruction to be executed in a program is always the first one in the instruction list of  $P$ , the value of the program counter is set to 0.
- The value of the output buffer is set to  $\text{FLUSH}(0)$  so as to indicate that no  $\text{printi}$  instructions has been executed yet.

**Definition 4.7.**

$$\begin{aligned} \text{init} &: \mathbf{Program} \rightarrow \mathbf{Configuration} \\ \text{init}(Pdcl, I) &= (NT, AB_{ok}, 0, \text{FLUSH}(0), \text{mapof}(I)) \end{aligned}$$

Definition 4.8 defines a machine function  $M$  which models a simple machine executing IL programs.  $M$  computes a configuration  $M(P, \sigma, n)$  which is a result of  $n$  successive program execution transitions starting from a configuration  $\sigma$ .

**Definition 4.8.**

$$\begin{aligned} M &: \mathbf{Program} \times \mathbf{Nat} \rightarrow \mathbf{Configuration} \\ M(P, \sigma, 0) &= \sigma \\ M(P, \sigma, \text{Suc}(n)) &= \text{exec}(P, M(P, \sigma, n)) \end{aligned}$$

At the beginning of this thesis, we informally defined semantics of an IL program as its observable behavior and the observable behavior of a program as a sequence of integers printed by the program. Now, with the definition of the operational semantics at hand, we can formalize the notion of the observable behavior and give the definition of a program semantics function for the language IL.

We begin by introducing syntactic sets associated with the notion of the observable behavior:

- tokens **Token**,
- emissions **Emission**, and
- observable behavior **ObservableBehavior**;

and metavariables ranging over these sets:

- $tok$  ranges over tokens **Token**,

- $em$  ranges over emissions **Emission**, and
- $seq$  ranges over observable behaviors **ObservableBehavior**.

Our formalization of sequences of integers printed by a program differs from standard formalizations of sequences as it must take into account the following constraints:

- Sequences can be finite or infinite (standard constraint).
- Our formalization of sequences must take into account that the sequences model observable behaviors of programs and programs can terminate or not. As termination is also a part of observable behavior of a program, a sequence of integers printed by the program alone does not completely specify its behavior. Consequentially, the equality of two integer sequences printed by two different programs alone can not be the main criterion for observable behavior equivalence of the programs as the first one can terminate directly after printing a sequence and the second one can print the same sequence and then start looping infinitely.
- The notion of observable behavior has to be formalized in a way that allows to reason formally about equivalence of program behaviors. In particular, it has to support reasoning by inductive arguments.

Hence, we model sequences of integers printed by IL programs as sets of indexed tokens  $seq \in \mathbf{ObservableBehavior}$  where each token  $tok \in \mathbf{Token}$  in  $seq$  has either the form  $\mathbf{WRITETOK}(n, i)$  or the form  $\mathbf{ENDTOK}(n)$ , cf. Definition 4.9. The modelling relation between observable behavior of a program  $P$  and a sequence  $seq$  is defined as follows.

- A token of the form  $\mathbf{WRITETOK}(n, i_n)$  is an element of  $seq$  iff  $P$  prints a sequence with a prefix  $\{i_0, i_1, \dots, i_n\}$ .
- A token of the form  $\mathbf{ENDTOK}(n)$  is an element of  $seq$  iff  $P$  prints a sequence  $\{i_0, i_1, \dots, i_{n-1}\}$  and terminates.

**Definition 4.9.**

$$\begin{aligned} \mathbf{Token} \ni tok &::= \mathbf{WRITETOK}(n, i) \mid \mathbf{ENDTOK}(n) \\ seq \in \mathbf{ObservableBehavior} &= \mathcal{P}(\mathbf{Token}) \end{aligned}$$

Let us consider a transition of execution from a configuration to successor configuration for a program. As the transition does not necessarily changes observable behavior of the program and if it does it, then the change can result either from printing an integer or from termination of the program, we formalized an auxiliary notion of emission which captures all possible behaviors of a program during a transition and allows us to formally describe overall observable behavior of a program during its whole execution.

Definition 4.10 gives formation rule for the syntactic set **Emission** that models the set of emissions and definition of a function **emission** that computes emission  $\mathbf{emission}(\sigma)$  for a configuration  $\sigma$ . If one takes into account the forms of tokens then an emission  $em \in \mathbf{Emission}$  of a program has one of three forms:

1. An emission of the form **NOEMIT** is associated to a transition of a program iff the program neither prints an integer during the transition nor terminates during the transition.
2. An emission of the form  $\mathbf{EMIT}(\mathbf{WRITETOK}(n, i_n))$  is associated to a transition of a program iff the program has already printed a sequence of integers  $\{i_0, i_1, \dots, i_{n-1}\}$  and it prints integer  $i_n$  during the transition.
3. An emission of the form  $\mathbf{EMIT}(\mathbf{ENDTOK}(n))$  is associated to a transition iff the program has already printed a sequence of integers  $\{i_0, i_1, \dots, i_{n-1}\}$  and it terminates during the transition.

**Definition 4.10.**

<p><b>Emission</b> <math>\ni em ::= \text{EMIT}(tok) \mid \text{NOEMIT}</math></p> <p><b>emission</b> : <b>Configuration</b> <math>\rightarrow</math> <b>Emission</b></p> <p><math>\text{emission}(\Gamma, af, pc, \text{FLUSH}(n), s) = \text{EMIT}(\text{ENDTOK}(n))</math></p> <p><math>\text{emission}(\text{NT}, af, pc, \text{FLUSH}(n), s) = \text{NOEMIT}</math></p> <p><math>\text{emission}(\text{NT}, af, pc, \text{WRITE}(n, i), s) = \text{EMIT}(\text{WRITETOK}(n, i))</math></p>
---

With the above definitions at hand, we can give a definition of a program semantics function for the IL language, cf. Definition 4.11. The program semantics is defined by a function **Sem** which maps IL programs to their observable behaviors, cf. Figure As sequences resulting from execution of IL program can possibly be infinite, the definition of **Sem** is given in declarational style and is based both on operational semantics and trace semantics. An informal idea of the algorithm underlying **Sem** is as follows. We let the machine **M** execute a program *P* forever by applying successively the transition function **exec** in an infinite loop. In doing so, we generate an infinite trace of configurations *t* where each element of the trace is a configuration resulting from *n* successive transitions starting from initial configuration  $\text{init}(P)$ ,  $t(n) = \mathbf{M}(P, n)$ . Each trace element  $t(i+1)$  carries information about changes of observable behavior of *P* during transition from  $t(i)$  to  $t(i+1)$  encoded in the state of output buffer. The algorithm builds an output sequence of tokens from the trace in three steps:

1. First step inspects iteratively all elements of *t* in the order of their producing,  $t(0), t(1), \dots$ , and computes an intermediate sequence of emmissions of *P*.
2. Second step filters out **NOEMIT** emissions from the intermediate sequence.
3. Third step builds output sequence of tokens by stripping **EMIT** constructors from elements of the sequence resulting from the second step.

**Definition 4.11.**

<p><b>Sem</b> : <b>Program</b> <math>\rightarrow</math> <b>ObservableBehavior</b></p> <p><math>\text{Sem}(P) = \{tok \mid \exists n. \text{emission}(\mathbf{M}(P, n)) = \text{EMIT}(tok)\}</math></p>
--

### 4.3 Translation correctness predicate

This section presents the definition of a translation correctness predicate which is based on the definition of program semantics given in the previous section.

As IL programs are defined as pairs  $(Pdecl, I)$  consisting of a program declarations *Pdecl* and an input *I*, there are in general two possible definitions of the correctness predicate that seem to be reasonable:

The first one is defined as a predicate **corrTrans** in Definition 4.12. According to the definition of **corrTrans**, two programs are semantically equivalent iff their observable behaviors are the same. In this thesis, we use **corrTrans** as translation correctness predicate.

**Definition 4.12.**

<p><b>corrTrans</b> : <b>Program</b> <math>\times</math> <b>Program</b> <math>\rightarrow</math> <b>Bool</b></p> <p><math>\text{corrTrans}(S, T) \equiv \text{Sem}(S) = \text{Sem}(T)</math></p>
--

The second one, shown in Definition 4.13, uses additionally two predicates `wfi` and `conform` with signatures

`wfi` : **ProgramDecl** × **Input** → **Bool**

and

`conform` : **Input** × **Input** → **Bool**,

respectively.

**Definition 4.13.**

`corrTrans2` : **ProgramDecl** × **ProgramDecl** → **Bool**  
`corrTrans2`( $Pdcl_S, Pdcl_T$ ) ≡  
 $\forall I_S I_T. \text{wfi}(Pdcl_S, I_S) \wedge \text{wfi}(Pdcl_T, I_T) \wedge \text{conform}(I_1, I_2)$   
 $\longrightarrow \text{Sem}(Pdcl_S, I_S) = \text{Sem}(Pdcl_T, I_T)$

Informally, expressions `wfi`( $Pdcl, I$ ) and `conform`( $I_S, I_T$ ) say that program input  $I$  is well-formed w.r.t. to a program declaration  $Pdcl$  and that inputs  $I_S$  and  $I_T$  conform to each other according to the definition of `conform`. A formal definition of `wfi` will be given later on. A formal definition of `conform` is not given in this thesis. The most commonly used definition of `conform`( $I_S, I_T$ ) is equality of two inputs,  $I_S = I_T$ . Although the `corrTrans2` is more suitable for verification of intraprocedural optimizations, which are subject of this thesis too, we made a design decision to favor the predicate `corrTrans` over `corrTrans2`. The reasons for making such decision are as follows.

Firstly, our IL programs comprise only one procedure  $Pdcl$  and input definition  $I$ ; and IL instruction set contains no procedure calls. For this prototype scenario, we wanted to investigate the problem of verification of optimizations performed on standalone programs with known inputs using a correctness predicate which makes no assumptions weakening its statement. Here, the reader should note that it is easier to prove a statement

`corrTrans2`( $Pdcl_S, Pdcl_T$ )

for two IL program declarations  $Pdcl_S$  and  $Pdcl_T$  than to prove a statement

`corrTrans`(( $Pdcl_S, I_S$ ), ( $Pdcl_T, I_T$ ))

for the same program declarations and concrete inputs  $I_S$  and  $I_T$ . To prove `corrTrans2`( $Pdcl_S, Pdcl_T$ ), we are allowed to introduce two fresh variables  $I'_S \in \mathbf{Input}$  and  $I'_T \in \mathbf{Input}$  and to make three assumptions:

1. `wfi`( $Pdcl_S, I'_S$ ),
2. `wfi`( $Pdcl_T, I'_T$ ), and
3. `conform`( $I'_1, I'_2$ )

and from this we have to derive the equation  $\text{Sem}(Pdcl_S, I'_S) = \text{Sem}(Pdcl_T, I'_T)$ . In contrast, if we want to prove `corrTrans`(( $Pdcl_S, I_S$ ), ( $Pdcl_T, I_T$ )) for two programs ( $Pdcl_S, I_S$ ) and ( $Pdcl_T, I_T$ ) then we are not allowed to make any assumptions and we have to discharge the assumptions

1. `wfi`( $Pdcl_S, I_S$ ),
2. `wfi`( $Pdcl_T, I_T$ ), and
3. `conform`( $I_1, I_2$ )

first before we derive from them the equation  $\text{Sem}(Pdcl_S, I_S) = \text{Sem}(Pdcl_T, I_T)$ .

Secondly, our framework can be easily adapted to allow verifications according to the definition of `corrTrans2` as it provides more proof techniques than necessary to prove a corresponding statement `corrTrans2(Pdcl_S, Pdcl_T)`.

Thirdly, in some programming languages there is a calling convention that local variables are assumed to have predefined values (for example in the Java language). This means that if we extend our language IL by procedure calls with this calling convention, then we have to adapt syntax and semantics of a new language by splitting states of computations and inputs for new programs in global and local parts. As in this new scenario local parts on inputs will have to be treated exactly in the same way as in our framework, large parts of our work will be directly reusable. In particular, our proof environments providing proof techniques for discharging aforementioned assumptions about well-formedness of program declarations since well-formedness of program declarations is one of prerequisites of well-typedness property of IL programs which is formalized in the next section.



## Chapter 5

---

### Type safety of the language IL

This section presents the content of Layer 3 of our implementation of the SVF. In Section 3.4, we motivated that the general purpose of Layer 3 is to provide proofs that the source and the target languages are type safe. Also there, we explained why the type safety proofs have to be provided by a SVF which follows the FTV approach. As we are interested in translation certificates attesting to the optimization correctness and our translation contract provides the formalization of only one language IL, our implementation of Layer 3 provides a formal framework for one language only, IL. The formal framework of our implementation of Layer 3 provides the following:

1. The formalization of a simple type system for the language IL.
2. The formalization of the notion of well-typedness of IL programs.
3. The formalization of the notion of type safe execution of an IL program.
4. A proof of a type safety theorem saying that if an IL program is well-typed w.r.t. to a program type, then all partial executions of that program are type safe.

The rest of this section is organized as follows. Section 5.1 presents definitions of predicates checking well-formedness of IL programs and their parts. Section 5.2 gives a definition of a predicate that says when an IL program is considered as well-typed. Section 5.3 gives a definition of a configuration conformance predicate that says when a configuration conforms with a program type. The last Section 5.4 presents type safety theorem.

#### 5.1 Well-formedness

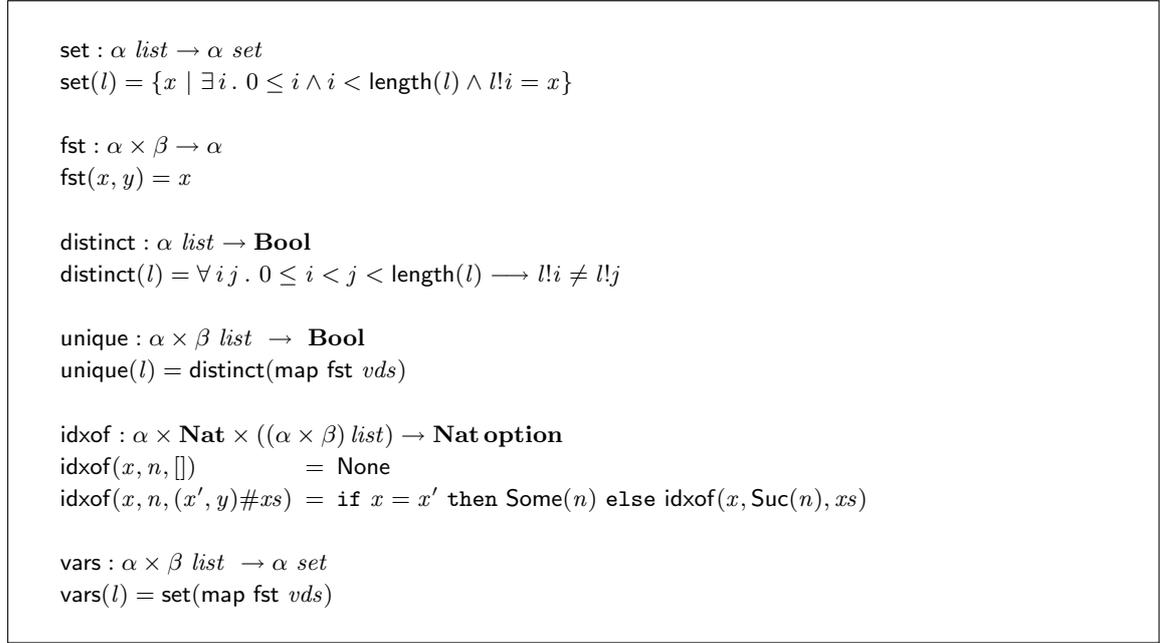
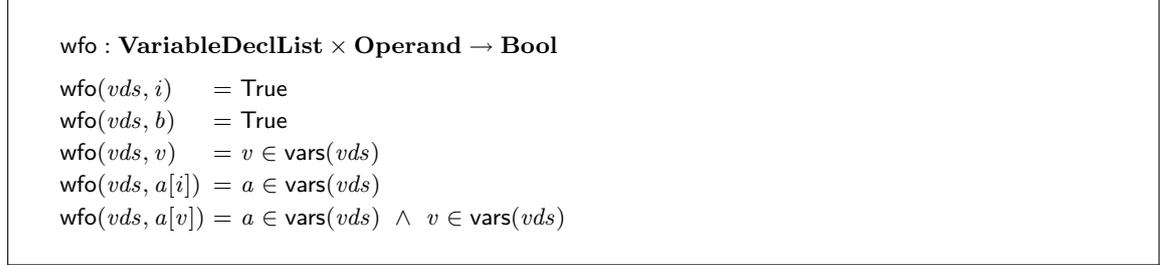
This section presents a formal specification of well-formedness of IL programs. Figure 5.1 shows the definitions of auxiliary functions used in this sections.

Definition 5.1 defines a `wfvds` which checks if a variable declaration list `vds` is well-formed: `wfvds(vds)` holds true iff the variable declaration list contains no doubly declared identifiers.

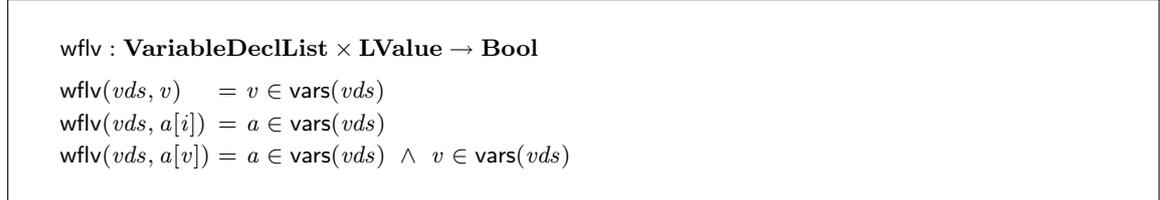
**Definition 5.1.**

<pre>wfvds : VariableDeclList → Bool wfvds(vds) = unique(vds)</pre>
---

Definition 5.2 defines a predicate `wfo` which specifies when an operator `o` is well-formed w.r.t. a variable declaration list `vds`: `wfo(vds, o)` holds true for an operand `o` and a variable declaration list `vds` iff `o` is either a constant value or all variables in `o` declared in `vds`.

**Fig. 5.1.** Auxiliary function definitions**Definition 5.2.**

Definition 5.3 defines a predicate **wflv** which specifies when an l-value  $lv$  is well-formed w.r.t. a variable declaration list  $lds$ : **wflv**( $lds, lv$ ) holds true for an l-value  $lv$  and a variable declaration list  $lds$  iff  $lv$  is either a declared variable  $v$  or  $lv$  has the form of an indexed operator whose all variables are declared in  $lds$ .

**Definition 5.3.**

Definition 5.4 defines a predicate **wfe** which specifies when an expression  $e$  is well-formed w.r.t. a variable declaration list  $lds$ : **wfe**( $lds, e$ ) holds true for an expression  $e$  and a variable declaration list  $lds$  iff  $e$  fulfills one of the following three conditions:

1.  $e$  is an operator  $o$  and  $o$  is well-formed w.r.t. the variable declaration list  $lds$  or
2.  $e$  is a unary expression of the form  $unop\ o$ , where  $unop$  stands for a unary operator,  $unop \in \{-, \neg\}$ , and  $o$  well-formed w.r.t. the variable declaration list  $lds$  or

3.  $e$  is a binary expression of the form  $o_1 \text{ binop } o_2$  where  $\text{binop}$  stands for a binary operator,  $\text{binop} \in \{+, -, *, \wedge, \vee, =, \neq, <, \leq\}$ , and  $o_1$  and  $o_2$  are well-formed w.r.t. the variable declaration list  $\text{vds}$ .

**Definition 5.4.**

**wfe** : **VariableDeclList**  $\times$  **Expression**  $\rightarrow$  **Bool**

$$\begin{aligned} \text{wfe}(\text{vds}, o) &= \text{wfo}(\text{vds}, o) \\ \text{wfe}(\text{vds}, o_1 + o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, o_1 - o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, o_1 * o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, -o) &= \text{wfo}(\text{vds}, o) \\ \text{wfe}(\text{vds}, o_1 \wedge o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, \neg o) &= \text{wfo}(\text{vds}, o) \\ \text{wfe}(\text{vds}, o_1 \vee o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, o_1 = o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, o_1 \neq o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, o_1 < o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \\ \text{wfe}(\text{vds}, o_1 \leq o_2) &= \text{wfo}(\text{vds}, o_1) \wedge \text{wfo}(\text{vds}, o_2) \end{aligned}$$

Definition 5.5 defines a predicate  $\text{wfi}$  which formulates well-formedness requirements for an instruction  $\text{instr}$  which must hold w.r.t. a variable declaration list  $\text{vds}$  if  $\text{instr}$  is the  $pc$ -th instruction of an instruction list  $\text{instrs}$ , the flow of control is currently at the program point  $pc$ , and the machine  $M$  is about to execute this instruction. Basically, there are two well-formedness requirements for an instruction in a program declaration to be executable by the machine  $M$ :

1. Each variable occurring in the instruction must be declared in the variable declaration list of the program declaration.
2. After executing the instruction, the flow of control must be again at a program point, i.e. the flow of control must remain within the bounds of the instruction list of the program declaration.

**Definition 5.5.**

**wfi** : **ProgramDecl**  $\times$  **ProgramCounter**  $\times$  **Instruction**  $\rightarrow$  **Bool**

$$\begin{aligned} \text{wfi}((\text{vds}, \text{instrs}), pc, lv := e) &= \text{wflv}(\text{vds}, lv) \wedge \text{wfe}(\text{vds}, e) \wedge \text{Suc}(pc) < \text{length}(\text{instrs}) \\ \text{wfi}((\text{vds}, \text{instrs}), pc, \text{printi}(e)) &= \text{wfe}(\text{vds}, e) \wedge \text{Suc}(pc) < \text{length}(\text{instrs}) \\ \text{wfi}((\text{vds}, \text{instrs}), pc, \text{branch}(e, dst)) &= \text{wfe}(\text{vds}, e) \wedge \text{Suc}(pc) < \text{length}(\text{instrs}) \wedge \\ &\quad dst < \text{length}(\text{instrs}) \\ \text{wfi}((\text{vds}, \text{instrs}), pc, \text{goto}(dst)) &= dst < \text{length}(\text{instrs}) \\ \text{wfi}((\text{vds}, \text{instrs}), pc, \text{exit}) &= \text{True} \end{aligned}$$

Definition 5.6 defines a predicate  $\text{wfil}$  which takes a program declaration  $(\text{vds}, \text{instrs})$  as input and checks if the instruction list  $\text{instrs}$  is well-formed w.r.t. the variable declaration list  $\text{vds}$ :  $\text{wfil}(\text{vds}, \text{instrs})$  holds true iff each instruction in the instruction list  $\text{instrs}$  is well-formed w.r.t. the program declaration  $(\text{vds}, \text{instrs})$  and its index in  $\text{instrs}$ .

**Definition 5.6.**

**wfil** : **ProgramDecl**  $\rightarrow$  **Bool**

$$\text{wfil}(\text{vds}, \text{instrs}) = \forall pc < \text{length}(\text{instrs}). \text{wfi}((\text{vds}, \text{instrs}), pc, \text{instrs}[pc])$$

Definition 5.7 defines a predicate `wfl` which checks if an input  $I$  is well-formed w.r.t. a variable declaration list  $vs$ : `wfl(vs, I)` holds for an input  $I$  and a variable declaration list  $vs$  iff there is a one-to-one correspondence between the elements of the list  $vs$  and the elements of the list  $I$  that is defined as follows: the  $i$ -th element of the variable declaration list  $vs$  is a variable declaration  $(v, \tau)$  iff the  $i$ -th element of the input list  $I$  is a pair  $(v, val)$  such that the value  $val$  has the type of  $\tau$ .

**Definition 5.7.**

```

typeof : Value → Bool
typeof( $b$ ) = bool
typeof( $i$ ) = int
typeof( $bl$ ) = barray(length( $bl$ ))
typeof( $il$ ) = ibarray(length( $il$ ))

wfl : VariableDeclList × Input → Bool
wfl( $vs, I$ ) = length( $vs$ ) = length( $I$ ) ∧
  ∀(( $v, \tau$ ), ( $v', val$ )) ∈ set(zip( $vs, I$ )). ( $v = v'$ ) ∧ (typeof( $val$ ) =  $\tau$ )

```

Definition 5.8 defines a well-formedness predicate `wfprg` on IL programs. An IL program  $((vs, instrs), I)$  is well-formed iff its variable declaration list  $vs$  is well-formed and its instruction list  $instrs$  is well-formed w.r.t.  $vs$  and input list  $I$  are well-formed w.r.t.  $vs$ .

**Definition 5.8.**

```

wfprg : Program → Bool
wfprg(( $vs, instrs$ ),  $I$ ) = wfvds( $vs$ ) ∧ wfil( $vs, instrs$ ) ∧ wfl( $vs, I$ )

```

## 5.2 Well-typedness

This section presents formalization of the notion well-typedness of IL programs.

A standard formalization of assigning types to programs (typing) [86, 88, 84, 82, 58, 83, 59, 60] comprises three parts: The first part formalizes the notion of a typing environment. The second part formalizes the notion of program type. The third part specifies typing by giving a finite set of derivation rules to build inductively a typing relation over pairs consisting of programs and program types in the context of a typing environment. The fourth part specifies a well-typedness predicate that says when a program is well-typed in the context of a typing environment. A standard formulation of the well-typedness predicate says that a program is well-typed w.r.t. a typing environment and a program type iff the pair consisting of the program and the program type is in the typing relation built in the context of the typing environment.

As our IL language and its type system are very simple, we can simplify the above well-typedness formalization scheme: The first and the second parts collapse to one part since in our formalization a program type and a typing environment for a program are the same. Further, we are concerned with a special case of the typing relation which comprises only one pair consisting of the program and its type. For this reason, it is not necessary to define explicitly a typing environment for a program and to define an inductive typing relation; the third and the fourth parts collapse into one part which specifies typing by means of a well-typedness predicate which has a program and a program type as parameters and is defined by primitive recursion.

We begin our presentation by introducing a syntactic set of program types **ProgramType** and a metavariable  $\Phi$  ranging over program types  $\Phi$ .

Definition 5.9 gives formation rule for the syntactic set **ProgramType**. A program type  $\Phi$  is a list of type identifiers  $\tau$  which is always defined for a program  $((vds, instrs), I)$  as follows.

map snd  $vds$

For this reason, our program well-typedness predicate could actually do without the program type parameter but we left it for brevity. As aforementioned,  $\Phi$  is an encoding of both a program type and a typing environment for a program. The interpretation of  $\Phi$  as an implicit typing environment  $\{v_0 \mapsto \tau_0, \dots, v_{n-1} \mapsto \tau_{n-1}\}$  for a program  $((vds, instrs), I)$  is done as follows: The variable declaration  $(v_i, \tau_i)$  is the  $i$ -th element of the variable declaration list  $vds$  iff binding  $v_i \mapsto \tau_i$  is in the typing environment for  $((vds, instrs), I)$ .

**Definition 5.9.**

$$\mathbf{ProgramType} \ni \Phi ::= [] \mid \tau \# \Phi$$

Definition 5.10 defines a predicate  $wtv$  which checks if a variable is well-typed. Checking is done in the context of three further parameters: a program  $((vds, instrs), I)$ , a program type  $\Phi$ , and a type identifier  $\tau$ ; and the variable well-typedness predicate  $wtv$  requires for a variable  $v$  to have a type  $\tau$  in the context of a program  $((vds, instrs), I)$  and a program type  $\Phi$  that if the  $i$ -th element of the variable declaration list  $vds$  is a variable declaration  $(v, \tau)$  then  $\tau$  is the  $i$ -th element of the program type  $\Phi$ .

**Definition 5.10.**

$$\begin{aligned} wtv : \mathbf{ProgramType} \times \mathbf{ProgramDecl} \times \mathbf{Type} \times \mathbf{Variable} &\rightarrow \mathbf{Bool} \\ wtv(\Phi, (vds, instrs), \tau, v) &= \begin{cases} \Phi!(\text{idxof}(v, 0, vds)) = \tau & \text{if } (v, \tau) \in \text{set}(vds), \\ \text{False} & \text{otherwise} \end{cases} \end{aligned}$$

Definition 5.11 defines a predicate  $wtio$  which checks if an operand  $o$  is of the type `int` in the context of a program declaration  $(vds, instrs)$  and a program type  $\Phi$ : An operator  $o$  is of the type `int` in the context of the program declaration  $(vds, instrs)$  and a program type  $\Phi$  iff

1.  $o$  is an integer constant  $i$  or
2.  $o$  is a well-typed integer variable  $v$  in the context of  $(vds, instrs)$  and  $\Phi$  or
3.  $o$  is an indexed operator of the form  $a[i]$  and the variable  $a$  is a well-typed integer array variable in the context of  $(vds, instrs)$  and  $\Phi$  or
4.  $o$  is an indexed operator of the form  $a[v]$  and the variables  $a$  and  $v$  are well-typed integer array and integer variables, respectively, in the context of  $(vds, instrs)$  and  $\Phi$ .

**Definition 5.11.**

$$\begin{aligned} wtio : \mathbf{ProgramType} \times \mathbf{ProgramDecl} \times \mathbf{Operand} &\rightarrow \mathbf{Bool} \\ wtio(\Phi, Pdcl, i) &= \text{True} \\ wtio(\Phi, Pdcl, b) &= \text{False} \\ wtio(\Phi, Pdcl, v) &= wtv(v, \text{int}, Pdcl, \Phi) \\ wtio(\Phi, Pdcl, a[i]) &= \exists n. wtv(\Phi, Pdcl, \text{iarray}(n), a) \\ wtio(\Phi, Pdcl, a[v]) &= \exists n. wtv(\Phi, Pdcl, \text{iarray}(n), a) \wedge wtv(\Phi, Pdcl, \text{int}, v) \end{aligned}$$

Definition 5.12 defines a predicate `wtbo` which checks if an operand  $o$  is of the type `bool` in the context of a program declaration  $(vds, instrs)$  and a program type  $\Phi$ : An operator  $o$  is of the type `bool` in the context of the program declaration  $(vds, instrs)$  and a program type  $\Phi$  iff

1.  $o$  is a boolean constant  $b$  or
2.  $o$  is a well-typed boolean variable  $v$  in the context of  $(vds, instrs)$  and  $\Phi$  or
3.  $o$  is an indexed operator of the form  $a[i]$  and the variable  $a$  is a well-typed boolean array variable in the context of  $(vds, instrs)$  and  $\Phi$  or
4.  $o$  is an indexed operator of the form  $a[v]$  and the variables  $a$  and  $v$  are well-typed boolean array and integer variables, respectively, in the context of  $(vds, instrs)$  and  $\Phi$ .

**Definition 5.12.**

**wtbo** : **ProgramType**  $\times$  **ProgramDecl**  $\times$  **Operand**  $\rightarrow$  **Bool**

$\text{wtbo}(\Phi, Pdcl, i) = \text{False}$

$\text{wtbo}(\Phi, Pdcl, b) = \text{True}$

$\text{wtbo}(\Phi, Pdcl, v) = \text{wtv}(v, \text{bool}, Pdcl, \Phi)$

$\text{wtbo}(\Phi, Pdcl, a[i]) = \exists n. \text{wtv}(\Phi, Pdcl, \text{barray}(n), a)$

$\text{wtbo}(\Phi, Pdcl, a[v]) = \exists n. \text{wtv}(\Phi, Pdcl, \text{barray}(n), a) \wedge \text{wtv}(\Phi, Pdcl, \text{int}, v)$

Definition 5.13 defines a predicate `wtbe` which checks if an expression  $e$  is of the type `bool` in the context of a program declaration  $(vds, instrs)$  and a program type  $\Phi$ : An expression  $e$  is of the type `bool` in the context of the program declaration  $(vds, instrs)$  and a program type  $\Phi$  iff

1.  $e$  is a well-typed boolean operator  $o$  in the context of  $(vds, instrs)$  and  $\Phi$  or
2.  $e$  is a binary expression of the form  $o_1 \text{ bop } o_2$  where  $\text{bop}$  stands for a binary boolean operator,  $\text{bop} \in \{\wedge, \vee\}$ , and  $o_1$  as well as  $o_2$  are both well-typed boolean operators in the context of  $(vds, instrs)$  and  $\Phi$  or
3.  $e$  is a unary expression of the form  $\neg o$  and  $o$  is a well-typed boolean operator  $o$  in the context of  $(vds, instrs)$  and  $\Phi$  or
4.  $e$  is a binary expression of the form  $o_1 \text{ bop } o_2$  where  $\text{bop}$  stands for an integer comparison operator,  $\text{bop} \in \{=, \neq, <, \leq\}$ , and  $o_1$  as well as  $o_2$  are both well-typed integer operators in the context of  $(vds, instrs)$  and  $\Phi$  or

**Definition 5.13.**

**wtbe** : **Expression**  $\times$  **ProgramDecl**  $\times$  **ProgramType**  $\rightarrow$  **Bool**

$\text{wtbe}(o, Pdcl, \Phi) = \text{wtbo}(o, Pdcl, \Phi)$

$\text{wtbe}(o_1 + o_2, Pdcl, \Phi) = \text{False}$

$\text{wtbe}(o_1 - o_2, Pdcl, \Phi) = \text{False}$

$\text{wtbe}(o_1 * o_2, Pdcl, \Phi) = \text{False}$

$\text{wtbe}(\neg o, Pdcl, \Phi) = \text{False}$

$\text{wtbe}(o_1 \wedge o_2, Pdcl, \Phi) = \text{wtbo}(o_1, Pdcl, \Phi) \wedge \text{wtbo}(o_2, Pdcl, \Phi)$

$\text{wtbe}(o_1 \vee o_2, Pdcl, \Phi) = \text{wtbo}(o_1, Pdcl, \Phi) \wedge \text{wtbo}(o_2, Pdcl, \Phi)$

$\text{wtbe}(\neg o, Pdcl, \Phi) = \text{wtbo}(o, Pdcl, \Phi)$

$\text{wtbe}(o_1 = o_2, Pdcl, \Phi) = \text{wtio}(o_1, Pdcl, \Phi) \wedge \text{wtio}(o_2, Pdcl, \Phi)$

$\text{wtbe}(o_1 \neq o_2, Pdcl, \Phi) = \text{wtio}(o_1, Pdcl, \Phi) \wedge \text{wtio}(o_2, Pdcl, \Phi)$

$\text{wtbe}(o_1 < o_2, Pdcl, \Phi) = \text{wtio}(o_1, Pdcl, \Phi) \wedge \text{wtio}(o_2, Pdcl, \Phi)$

$\text{wtbe}(o_1 \leq o_2, Pdcl, \Phi) = \text{wtio}(o_1, Pdcl, \Phi) \wedge \text{wtio}(o_2, Pdcl, \Phi)$

Definition 5.14 defines a predicate `wtie` which checks if an expression  $e$  is of the type `int` in the context of a program declaration  $(vds, instrs)$  and a program type  $\Phi$ : An expression  $e$  is of the type `int` in the context of the program declaration  $(vds, instrs)$  and a program type  $\Phi$  iff

1.  $e$  is a well-typed integer operator  $o$  in the context of  $(vds, instrs)$  and  $\Phi$  or
2.  $e$  is a binary expression of the form  $o_1 \text{ bop } o_2$  where  $\text{bop}$  stands for a binary integer operator,  $\text{bop} \in \{+, -, *\}$ , and  $o_1$  as well as  $o_2$  are both well-typed integer operators in the context of  $(vds, instrs)$  and  $\Phi$  or
3.  $e$  is a unary expression of the form  $-o$  and  $o$  is a well-typed integer operator  $o$  in the context of  $(vds, instrs)$  and  $\Phi$ .

**Definition 5.14.**

**wtie** : **Expression**  $\times$  **ProgramDecl**  $\times$  **ProgramType**  $\rightarrow$  **Bool**

$$\begin{aligned} \text{wtie}(o, Pdcl, \Phi) &= \text{wtio}(o, Pdcl, \Phi) \\ \text{wtie}(o_1 + o_2, Pdcl, \Phi) &= \text{wtio}(o_1, Pdcl, \Phi) \wedge \text{wtio}(o_2, Pdcl, \Phi) \\ \text{wtie}(o_1 - o_2, Pdcl, \Phi) &= \text{wtio}(o_1, Pdcl, \Phi) \wedge \text{wtio}(o_2, Pdcl, \Phi) \\ \text{wtie}(o_1 * o_2, Pdcl, \Phi) &= \text{wtio}(o_1, Pdcl, \Phi) \wedge \text{wtio}(o_2, Pdcl, \Phi) \\ \text{wtie}(-o, Pdcl, \Phi) &= \text{wtio}(o, Pdcl, \Phi) \\ \text{wtie}(o_1 \wedge o_2, Pdcl, \Phi) &= \text{False} \\ \text{wtie}(\neg o, Pdcl, \Phi) &= \text{False} \\ \text{wtie}(o_1 \vee o_2, Pdcl, \Phi) &= \text{False} \\ \text{wtie}(o_1 = o_2, Pdcl, \Phi) &= \text{False} \\ \text{wtie}(o_1 \neq o_2, Pdcl, \Phi) &= \text{False} \\ \text{wtie}(o_1 < o_2, Pdcl, \Phi) &= \text{False} \\ \text{wtie}(o_1 \leq o_2, Pdcl, \Phi) &= \text{False} \end{aligned}$$

Definition 5.15 defines a predicate `wtblv` which checks if an l-value  $lv$  is of the type `bool` in the context of a program declaration  $(vds, instrs)$  and a program type  $\Phi$ : An l-value  $lv$  is of the type `bool` in the context of the program declaration  $(vds, instrs)$  and a program type  $\Phi$  iff

1.  $lv$  is a well-typed boolean variable  $v$  in the context of  $(vds, instrs)$  and  $\Phi$  or
2.  $lv$  is of the form  $a[i]$  and the variable  $a$  is a well-typed boolean array variable in the context of  $(vds, instrs)$  and  $\Phi$  or
3.  $o$  is of the form  $a[v]$  and the variables  $a$  and  $v$  are well-typed boolean array and integer variables, respectively, in the context of  $(vds, instrs)$  and  $\Phi$ .

**Definition 5.15.**

**wtblv** : **ProgramType**  $\times$  **ProgramDecl**  $\times$  **LValue**  $\rightarrow$  **Bool**

$$\begin{aligned} \text{wtblv}(\Phi, Pdcl, v) &= \text{wtv}(\Phi, Pdcl, \text{bool}, v) \\ \text{wtblv}(\Phi, Pdcl, a[i]) &= \exists n. \text{wtv}(\Phi, Pdcl, \text{barray}(n), a) \wedge \\ \text{wtblv}(\Phi, Pdcl, a[v]) &= \exists n. \text{wtv}(\Phi, Pdcl, \text{barray}(n), a) \wedge \\ &\quad \text{wtv}(\Phi, Pdcl, \text{int}, v) \end{aligned}$$

Definition 5.16 defines a predicate `wtilv` which checks if an l-value  $lv$  is of the type `int` in the context of a program declaration  $(vds, instrs)$  and a program type  $\Phi$ : An l-value  $lv$  is of the type `int` in the context of the program declaration  $(vds, instrs)$  and a program type  $\Phi$  iff

1.  $lv$  is a well-typed integer variable  $v$  in the context of  $(vds, instrs)$  and  $\Phi$  or
2.  $lv$  is of the form  $a[i]$  and the variable  $a$  is a well-typed integer array variable in the context of  $(vds, instrs)$  and  $\Phi$  or
3.  $o$  is of the form  $a[v]$  and the variables  $a$  and  $v$  are well-typed integer array and integer variables, respectively, in the context of  $(vds, instrs)$  and  $\Phi$ .

**Definition 5.16.**

$$\begin{aligned} \text{wtilv} &: \mathbf{ProgramType} \times \mathbf{ProgramDecl} \times \mathbf{LValue} \rightarrow \mathbf{Bool} \\ \text{wtilv}(\Phi, Pdcl, v) &= \text{wtv}(\Phi, Pdcl, \mathbf{int}, v) \\ \text{wtilv}(\Phi, Pdcl, a[i]) &= \exists n. \text{wtv}(\Phi, Pdcl, \mathbf{iarray}(n), a) \wedge \\ \text{wtilv}(\Phi, Pdcl, a[v]) &= \exists n. \text{wtv}(\Phi, Pdcl, \mathbf{iarray}(n), a) \wedge \\ &\quad \text{wtv}(\Phi, Pdcl, \mathbf{int}, v) \end{aligned}$$

Definition 5.17 defines a predicate `wtinstr` which checks if an instruction `instr` is well-typed in the context of a program declaration  $(vds, instrs)$  and a program type  $\Phi$ : An instr `instr` is well-typed in the context of the program declaration  $(vds, instrs)$  and a program type  $\Phi$  iff

1. `instrs` is an assignment instruction `lv:=e` and the l-value `lv` and the expression `e` are both of the type either `int` or `bool` in the context of  $(vds, instrs)$  and  $\Phi$  or
2. `instrs` is a printi instruction `printi(e)` and `e` is well-typed integer expression in the context of  $(vds, instrs)$  and  $\Phi$  or
3. `instrs` is a branch instruction `branch(e, dst)` and `e` is well-typed boolean expression in the context of  $(vds, instrs)$  and  $\Phi$  or
4. `instrs` is a goto instruction `goto(dst)` or an exit instruction `exit`.

**Definition 5.17.**

$$\begin{aligned} \text{wtinstr} &: \mathbf{ProgramDecl} \times \mathbf{ProgramType} \times \mathbf{Instruction} \rightarrow \mathbf{Bool} \\ \text{wtinstr}(\Phi, Pdcl, lv:=e) &= \text{wtblv}(\Phi, Pdcl, lv) \wedge \text{wtbe}(\Phi, Pdcl, e) \vee \\ &\quad \text{wtilv}(\Phi, Pdcl, lv) \wedge \text{wtie}(\Phi, Pdcl, e) \\ \text{wtinstr}(\Phi, Pdcl, \text{printi}(e)) &= \text{wtie}(Pdcl, \Phi, e) \\ \text{wtinstr}(\Phi, Pdcl, \text{branch}(e, dst)) &= \text{wtbe}(Pdcl, \Phi, e) \\ \text{wtinstr}(\Phi, Pdcl, \text{goto}(dst)) &= \mathbf{True} \\ \text{wtinstr}(\Phi, Pdcl, \text{exit}) &= \mathbf{True} \end{aligned}$$

Definition 5.18 defines a predicate `wtvds` which checks if a variable declaration list `vds` is well-typed w.r.t. a program type  $\Phi$ : A variable declaration list `vds` is well-typed w.r.t. a program type  $\Phi$  iff the following holds true for all type identifiers: a type identifier  $\tau_i$  is the  $i$ -th element of the list  $\Phi$  iff there exists a variable  $v_i$  such that the variable declaration  $(v_i, \tau_i)$  is the  $i$ -th element of the list `vds`.

**Definition 5.18.**

$$\begin{aligned} \text{wtvds} &: \mathbf{ProgramType} \times \mathbf{VariableDeclList} \rightarrow \mathbf{Bool} \\ \text{wtvds}(\Phi, vds) &= (\Phi = \text{map snd } vds) \end{aligned}$$

Definition 5.19 defines a predicate `wtinstrs` which checks if an instruction list is well-typed in the context of a program type and a variable declaration list: An instruction list `instrs` of a program declaration  $(vds, instrs)$  is well-typed in the context of a program type  $\Phi$  iff

1. the `instrs` list is not empty and
2. each instruction in `instrs` is well-typed in the context of the program type  $\Phi$  and the program declaration  $(vds, instrs)$ .

**Definition 5.19.**

$$\begin{aligned}
& \text{wtinstrs} : \mathbf{ProgramType} \times \mathbf{ProgramDecl} \rightarrow \mathbf{Bool} \\
& \text{wtinstrs}(\Phi, (vds, instrs)) = 0 < \text{length}(instrs) \wedge \\
& \quad \forall pc. 0 \leq pc < \text{length}(instrs) \longrightarrow \text{wtinstr}(\Phi, (vds, instrs), instrs!pc)
\end{aligned}$$

Definition 5.20 defines a predicate **wtp** which checks if a program is well-typed w.r.t. a program type: A program  $((vds, instrs), I)$  is well-typed w.r.t. a program type  $\Phi$  iff

1. the program is well-formed and
2. the variable declaration  $vds$  is well-typed w.r.t. to the program type  $\Phi$  and
3. the instruction list  $instrs$  is well-typed w.r.t. to the program type  $\Phi$  and the variable declaration list  $vds$ .

**Definition 5.20.**

$$\begin{aligned}
& \text{wtp} : \mathbf{ProgramType} \times \mathbf{Program} \rightarrow \mathbf{Bool} \\
& \text{wtp}(\Phi, ((vds, instrs), I)) = \text{wfprg}((vds, instrs), I) \wedge \\
& \quad \text{wtvds}(\Phi, vds) \wedge \\
& \quad \text{wtinstrs}(\Phi, instrs)
\end{aligned}$$

### 5.3 Conform configuration

This section presents a formalization of hierarchically organized notions of conformance which are necessary to express type soundness of a programming language. The notion and the notation of the conformance we use in this section was inspired by von Oheimb's work [87] on analysing type safety of Java in Isabelle/HOL.

Definition 5.21 defines a variable conformance predicate which checks if a state maps a variable to a value that conforms to a type: The definition uses the notation  $s \vdash v :: \preceq \tau$  to formalize that the value of a variable  $v$  conforms to a type  $\tau$  in the context of a state  $s$  iff

1.  $s$  maps  $v$  to a boolean and  $\tau$  is equal **bool** or
2.  $s$  maps  $v$  to an integer and  $\tau$  is equal **int** or
3.  $s$  maps  $v$  to a boolean array of the length equal  $n$  and  $\tau$  is equal **barray**( $n$ ) or
4.  $s$  maps  $v$  to an integer array of the length equal  $n$  and  $\tau$  is equal **iarray**( $n$ ).

**Definition 5.21.**

$$\begin{aligned}
& \_ \vdash \_ :: \preceq \_ : \mathbf{State} \times \mathbf{Variable} \times \mathbf{Type} \rightarrow \mathbf{Bool} \\
& s \vdash v :: \preceq \tau = \begin{cases} \text{True} & \text{if } s(v) = \text{Some}(b) \wedge \tau = \text{bool}, \\ \text{True} & \text{if } s(v) = \text{Some}(i) \wedge \tau = \text{int}, \\ \text{True} & \text{if } s(v) = \text{Some}(bl) \wedge \tau = \text{barray}(n) \wedge \text{length}(bl) = n, \\ \text{True} & \text{if } s(v) = \text{Some}(il) \wedge \tau = \text{iarray}(n) \wedge \text{length}(il) = n, \\ \text{False} & \text{otherwise} \end{cases}
\end{aligned}$$

The notion of conformance to a type can be extended to conformance to a list of types. For this, we introduce two new syntactic sets: variable lists **VariableList** and type identifier lists **TypeList**; and two metavariables  $vl$  and  $\tau l$  ranging over variable lists **VariableList** and type lists **TypeList**, respectively. Definition 5.22 gives formation rules for these sets.

**Definition 5.22.**

$$\begin{aligned} \mathbf{VariableList} &\ni vl ::= [] \mid v\#vl \\ \mathbf{TypeList} &\ni \tau l ::= [] \mid \tau\#\tau l \end{aligned}$$

Definition 5.23 defines a predicate  $\_ \vdash \_ [::\preceq] \_$  which checks if a list of variables conforms pointwise to a list of types in the context of a state: A list of variables  $vl$  conforms pointwise to a type list  $\tau l$  in the context of a state  $s$ ,  $s \vdash vl [::\preceq] \tau l$ , iff each variable  $vl!i$  from the variable list  $vl$  conforms to the corresponding type  $\tau l!i$  from the type list  $\tau l$ .

**Definition 5.23.**

$$\begin{aligned} \_ \vdash \_ [::\preceq] \_ &: \mathbf{State} \times \mathbf{VariableList} \times \mathbf{TypeList} \rightarrow \mathbf{Bool} \\ s \vdash vl [::\preceq] \tau l &= \forall (v, \tau) \in \mathbf{zip}(vl, \tau l). s \vdash v ::\preceq \tau \end{aligned}$$

Definition 5.24 defines a configuration conformance predicate which checks if a configuration conforms to a program type and a program. The definition uses the notation  $\Phi, ((vds, instrs), I) \vdash \sigma \surd$  to formalize that a configuration  $\sigma = (tf, af, pc, b, s)$  conforms to a program type  $\Phi$  and a program  $((vds, instrs), I)$  iff

1. the program counter  $pc$  points to a valid program point in the instruction list  $instrs$  and
2. a variable  $v$  is declared in the variable declaration list  $vds$  iff the state  $s$  maps  $v$  into a well-defined value and
3. all declared variables are mapped by the state  $s$  into values that are conform to types declared for these variables.

**Definition 5.24.**

$$\begin{aligned} \_, \_ \vdash \_ \surd &: \mathbf{ProgramType} \times \mathbf{Program} \times \mathbf{Configuration} \rightarrow \mathbf{Bool} \\ \Phi, ((vds, instrs), I) \vdash (tf, af, pc, b, s) \surd &= pc < \mathbf{length}(instrs) \wedge \\ &\quad \mathbf{dom}(s) = \mathbf{set}(\mathbf{vars}(vds)) \wedge \\ &\quad s \vdash (\mathbf{vars}(vds)) [::\preceq] \Phi \end{aligned}$$

Definition 5.25 defines a configuration reachability predicate which checks for two configurations and a program if one configuration is the result of partial execution of the program starting from the other configuration. The definition uses the notation  $P \vdash \sigma \longrightarrow_M \sigma'$  to express that a configuration  $\sigma'$  is reachable from a configuration  $\sigma$  during execution of a program  $P$  by the machine function  $M$  iff  $M$  can produce  $\sigma'$  after finitely many successive transitions starting from the configuration  $\sigma$ .

**Definition 5.25.**

$$\begin{aligned} \_ \vdash \_ \longrightarrow_M \_ &: \mathbf{Program} \times \mathbf{Configuration} \times \mathbf{Configuration} \rightarrow \mathbf{Bool} \\ P \vdash \sigma \longrightarrow_M \sigma' &= \exists n. M(P, \sigma, n) = \sigma' \end{aligned}$$

## 5.4 Type safety theorem

This section uses the definitions of well-formedness, well-typedness, and conformance from sections 5.1, 5.2, and 5.3, respectively, and formalizes the notion of type safety for the IL language and sets up a type safety theorem which we proved in Isabelle/HOL.

With the definitions of well-formedness, well-typedness, and conformance at hand, we can formulate the following theorem about the type safety of the IL language: If a program  $P$  is well-typed w.r.t. a program type  $\Phi$  and a configuration  $\sigma$  conforms to  $P$  and  $\Phi$ , program execution makes transition from  $\sigma$  to the successor configuration  $\text{exec}(P, \sigma)$ , then  $\text{exec}(P, \sigma)$  conforms to  $P$  and  $\Phi$ .

**Theorem 5.26 (Type safety of the IL language).**

$$\text{wtp}(P, \Phi) \wedge P, \Phi \vdash \sigma \surd \implies P, \Phi \vdash \text{exec}(P, \sigma) \surd$$

□

We proved Theorem 5.26 in Isabelle/HOL. However, to be able to apply the result proved in Theorem 5.26 in our translation correctness framework, we need a corollary about the type safety of partial executions of programs starting from the initial configurations for those programs. For the proof of the corollary, we proved first the following two statements:

The first statement is expressed by Lemma 5.27 and it says that initial configuration for an IL program  $P$ , which is well-typed w.r.t. a program type  $\Phi$ , conforms to  $P$  and  $\Phi$ .

**Lemma 5.27 (Conformance of initial configuration).**

$$\text{wtp}(P, \Phi) \implies P, \Phi \vdash \text{init}(P) \surd$$

□

The second statement is expressed by Corollary 5.28 and it says that partial execution of a well-typed program starting from a configuration that conforms to that program and its type produces a configuration that also conforms to the program and the program type.

**Corollary 5.28 (Type safety of partial executions).**

$$\begin{aligned} & \text{wtp}(P, \Phi) \wedge P, \Phi \vdash \sigma \surd \wedge P \vdash \sigma \longrightarrow_M \sigma' \\ & \implies \\ & P, \Phi \vdash \sigma' \surd \end{aligned}$$

*Proof.* To prove the corollary, we have to show that the conclusion  $P, \Phi \vdash \sigma' \surd$  is derivable from the premises:  $\text{wtp}(P, \Phi)$ ,  $P, \Phi \vdash \sigma \surd$ , and  $P \vdash \sigma \longrightarrow_M \sigma'$ . The proof is straightforward if we first unfold the definition of the reachability predicate in  $P \vdash \sigma \longrightarrow_M \sigma'$ , skolemize the  $\exists$ -quantified

variable  $n$ , and rewrite the conclusion and the premises. The resulting conclusion  $M(P, \sigma, n)$  can be easily derived from the premises  $\text{wtp}(P, \Phi)$  and  $P, \Phi \vdash \sigma \surd$ , if we first prove an auxiliary statement

$$\begin{aligned} & \text{wtp}(P, \Phi) \\ & \implies \\ & P, \Phi \vdash \sigma \surd \longrightarrow P, \Phi \vdash M(P, \sigma, n) \surd \end{aligned}$$

by induction over number of execution transitions  $n$ .

The proof of the induction begin is straightforward as by the definition of  $M$  the following equation it holds

$$P, \Phi \vdash M(P, \sigma, 0) \surd = P, \Phi \vdash \sigma \surd$$

To show the induction step, we have to show

$$\begin{aligned} & \text{wtp}(P, \Phi) \wedge P, \Phi \vdash \sigma \surd \longrightarrow P, \Phi \vdash M(P, \sigma, n) \surd \\ & \implies \\ & P, \Phi \vdash \sigma \surd \longrightarrow P, \Phi \vdash M(P, \sigma, \text{Suc } n) \surd. \end{aligned}$$

As by the definition of the function  $M$  holds  $M(P, \sigma, \text{Suc } n) = \text{exec}(P, M(P, \sigma, n))$ , we can rewrite the premises and the conclusion as follows.

$$\begin{aligned} & \text{wtp}(P, \Phi) \wedge P, \Phi \vdash M(P, \sigma, n) \surd \\ & \implies \\ & P, \Phi \vdash \text{exec}(P, M(P, \sigma, n)) \surd \end{aligned}$$

The last holds by Theorem 5.26. □

Now, we can set up Corollary 5.29 which states that partial execution of a well-typed program starting from its initial configuration always produces a configuration that conforms to the program and the program type.

**Corollary 5.29 (Type safety of partial program executions).**

$$\begin{aligned} & \text{wtp}(P, \Phi) \wedge P \vdash \text{init}(P) \longrightarrow_M \sigma \\ & \implies \\ & P, \Phi \vdash \sigma \surd \end{aligned}$$

□

The proof of Corollary 5.29 is straightforward and applies the results proved in Lemma 5.27 and Corollary 5.28.

## Chapter 6

---

### Optimization independent translation correctness criterion

This section presents the content of Layer 4 of our implementation of the SVF. In Section 3.5, we motivated that our implementation of Layer 4 provides the following:

1. the formalization of a translation correctness criterion TCC on the source and the target programs whose definition is independent of optimizations performed by our compiler and expresses a sufficient condition of the translation correctness predicate provided by the translation contract,
2. a proof of an optimization independent translation correctness theorem saying that if a source and a target programs  $S$  and  $T$ , respectively, fulfill the criterion TCC, then  $\text{corrTrans}(S, T)$  holds true.

Also there, we explained the motivation behind the TCC criterion: For each optimization  $O$  performed by our compiler, the implementation of the SVF has to provide a translation correctness criterion  $\text{TCC}_O$  on the source and the target programs and an optimization correctness theorem which are specific to the optimization  $O$ . The statement of such a theorem has the following form:

*If the source and the target programs of an optimization  $O$ ,  $S$  and  $T$ , respectively, fulfill the criterion  $\text{TCC}_O$ , then they fulfill the translation correctness predicate  $\text{corrTrans}(S, T)$ .*

The proof of this theorem has to be conducted in two steps:

1. The first step proves and applies a theorem of the following form:  
*If  $S$  and  $T$  fulfill the criterion  $\text{TCC}_O$ , then  $S$  and  $T$  fulfill the criterion TCC.*
2. The second step is the same for all optimizations supported by the SVF: It proves and applies a theorem which has the following form:  
*If  $S$  and  $T$  fulfill the criterion TCC, then  $\text{corrTrans}(S, T)$  holds true.*

As the second step is independent of the optimizations, we factored out the formalization of the criterion TCC and a proof of the theorem that proves the second step and inserted them into a dedicated Layer 4. Layer 5 provides the formalizations of the optimization specific criteria  $\text{TCC}_O$ , proofs of the first steps, and proofs of the optimization specific translation correctness theorems themselves. Doing this way, we make the SVF more structured:

- Layer 4 provides only formalizations which are independent of program transformations.
- Layer 5 provides only formalizations which are specific to concrete program transformations performed by the compiler.

Specifying and verifying a translation correctness criterion  $\text{TCC}_O$  for a new optimization requires less effort as only Layer 5 is affected by that change.

## 6.1 Overview

This section gives an overview of Layer 4. Layer 4 provides the formalizations of the following:

*The notion of a declaration of a control flow graph with blocks (CFGB):* Our notion of a CFGB declaration can be characterized as follows:

1. A CFGB declaration is always defined for an IL program.
2. A CFGB declaration is not a part of the syntax of an IL program which is defined for<sup>1</sup> but a separate data structure which declares the structure of the graph.
3. A CFGB declaration for an IL program does not define the graph explicitly but rather implicitly. A possible explicit definition of the CFGB for a program would be, for example, to define its node set as a mapping from block identifiers to lists of program points which are allocated to that identifiers and its edge set as a mapping from block identifiers to block identifier sets which would render the successor relation between the block sets. Instead of this, we formalized the notions of a block position and a block position descriptor which is a tuple of the form  $(pid, bid, bsize, bidx, pc)$  and whose components elaborately describe the "includes" relationship between the block  $bid$  and the program point  $pc$ : The meaning of a block position descriptor  $(pid, bid, bsize, bidx, pc)$  for an IL program  $P$  is that the program point  $pc$  is allocated to a block position with the identifier  $pid$  and that  $pid$  is included in a block with the identifier  $bid$  whose length is equal  $bsize$  and that  $pc$  has the index  $bidx$  within the block  $bid$ , i.e. that the  $pc$ -th instruction of  $P$  is the  $bidx$ -th instruction of  $bid$ .
4. A CFGB declaration itself is a tuple of mappings which define
  - the set of block position descriptors as a mapping from block position identifiers to block position descriptors,
  - the "includes" relation between the set of block positions and the set of blocks, and
  - the successor and predecessor relations between the sets of block positions.
5. The formalization of CFGB declarations enables one to encode CFGs with nested blocks. This feature is used in proofs of the correctness of the *NI* optimizations.
6. The notion of CFGB declarations is formalized by giving formation rules for the set **BlckPosEnv**.

The motivation for the formalization of the CFGB declarations is as follows.

Firstly, for each optimization in the chain of five optimizations *CF*, *DAE*, *NI*, *RAI*, and *RAE*, the compiler performs data flow analysis. The analysis is performed on a control flow graph (CFG), which is generated from an IL program, and its result is used by the compiler as a justification for an optimization. Optimizations are internally realized by the compiler as modifications of CFGs. Thus, each optimization results in two data structures encoding CFGs for two IL programs, source and target programs, and a data structure encoding the result of the data flow analysis performed on the CFG of the source program. Apparently, these encodings can be used by the proof generation unit of the compiler to generate a proof script with correctness proof for a concrete optimization, provided that the following is formalized within a SVF:

- the notion of CFGs,
- the notion of data flow analysis results,
- a predicate on two CFGs of source and target programs  $S$  and  $T$ ,  $CFG_S$  and  $CFG_T$ , and a data flow analysis result  $\mathcal{A}$  that expresses what does it mean that  $\mathcal{A}$  "justifies" applying

<sup>1</sup> For example, one can embed control flow graphs with blocks in the abstract syntax of the IL language by defining a new intermediate language whose programs are lists of lists of IL instructions.

optimization  $O$  to  $CFG_S$  and that  $CFG_T$  is a result of this optimization (optimization correctness criterion).

- the proof a theorem saying that if  $CFG_S$ ,  $CFG_T$ , and  $\mathcal{A}$  fulfill the above predicate, then the original programs  $S$  and  $T$ , which they represent, are semantically equivalent.

Secondly, in this thesis, structure preserving optimizations, like CF or DAE, have in common that they do not modify the sets of edges and nodes of the CFG of a source program but merely the node labels - the program instructions. Formalizing a SVF for such optimizations is relatively straightforward as the corresponding program points relation between program points of the source and the target programs and the corresponding nodes relation between nodes of the CFGs of these programs are one-to-one correspondences. In contrast to such optimizations, the NI optimization is a structure modifying transformation which modifies the sets of nodes and edges of the CFG of a source program with corresponding program points relations between the source and the target programs being one-to-many correspondences. As a consequence, all correspondences, which are of interest when formalizing an optimization correctness criterion for NI optimizations and a corresponding optimization correctness theorem, are one-to-many correspondences: corresponding nodes relation and corresponding edges relation. In order to be able to reason formally about such relations, one has to introduce to the SVF a formalization of CFGs with nested blocks.

*A well-formedness predicate on CFGB declarations and IL programs:*

$$\text{wfB} : \mathbf{Program} \times \mathbf{BlckPosEnv} \rightarrow \mathbf{Bool}$$

which checks if a CFGB declaration for an IL program adheres to the CFG of that program.

*An intermediate language of control flow graphs with blocks IL'*: Sections 4.1 and 4.2 presented the formalizations of the abstract syntax and the semantics of the intermediate language IL which can also be seen as a language of control flow graphs with the operational semantics defined by means of node-wise transitions of the flow of control in the following sense:

- For each IL program  $P$ , there exists a unique CFG with the node set comprising all program points of  $P$  and the edge set comprising node pairs which render the successor relation between the sets of program points of  $P$ . The nodes of the CFG are labeled with IL instructions which are at the respective program points in  $P$ .
- The definition of the operational semantics of the IL language is based on the definition of the configuration  $\sigma = (tf, af, pc, b, s)$  which is the result of partial execution of  $P$  and the value of its program counter component  $pc$  denotes both a program point of  $P$  and a node of the CFG which corresponds to  $P$ . As during execution of a program the flow of control transfers from program point to program point, we can also say that the flow of control transfers from node to node of the CFG which corresponds to that program and therefore we can say that IL is also a language of control flow graphs with node-wise transitions of the flow of control.

We formalized an intermediate language IL' which, according to the above analogy, can be seen as a language of control flow graphs with blocks whose operational semantics is defined by means of block-position-wise transfers of the flow of control:

- Each IL' program is a pair  $(P, B)$  consisting of an IL program  $P$  and a CFGB declaration  $B$ .

$$(P, B) \in \mathbf{Program}' ::= \mathbf{Program} \times \mathbf{BlckPosEnv}$$

- The definition of the operational semantics of the IL' language is based on the notion of an augmented configuration  $\sigma'$  which is the result of partial execution of an IL' program and has the form of a tuple  $(\beta, \sigma)$  consisting of a configuration  $\sigma$  and a component  $\beta$

that contains, among other things, block position descriptor  $(pid, bid, bsize, bidx, pc)$  whose value denotes a block position in  $B$ . The transition function of the IL' language computes the successor configuration  $\sigma'_{n+1} = (\beta_{n+1}, \sigma_{n+1})$  from a configuration  $\sigma'_n = (\beta_n, \sigma_n)$  by computing successor configuration  $\sigma_{n+1} = \text{exec}(P, \sigma_n)$  and computing the  $\beta_{n+1}$  component from  $B, \beta_n$ , and  $\sigma_{n+1}$ . The idea behind the augmented configuration  $(\beta, \sigma)$  is that the value its component  $\beta$  indicates current position of the flow of control in the CFGB declared by  $B$  and at the same time the value of the program counter component  $pc$  in  $\sigma$  indicates current position of the flow of control in the CFG that corresponds to  $P$ . This enables one to conduct proofs of statements comparing the operational semantics of IL programs and corresponding IL' programs.

- The semantics of an IL' program is defined as a sequence of tokens printed by that program, i.e. we give the definition of a program semantics function that is a mapping from the set **Program'** to the set **ObservableBehavior**,

We proved for all IL' programs  $(P, B)$  and their corresponding IL programs  $P$  that their observable behaviors are equal.

*An intermediate language of control flow graphs with blocks IL''*: Based on the formalization of the IL' language, we formalized an intermediate language IL'' which can be also seen as a language of control flow graphs with blocks whose operational semantics is defined by means of block-wise transfers of the flow of control:

- IL'' programs have the same syntax as IL' programs, i.e. they are tuples  $(P, B)$  consisting of an IL program  $P$  and a CFGB declaration  $B$ .

$$(P, B) \in \mathbf{Program}'' ::= \mathbf{Program} \times \mathbf{BlckPosEnv}$$

- The transition function of the IL'' language is based on the notion of an augmented configuration  $\sigma''$  which is equal to the configuration  $\sigma'$  and it computes the successor configuration from a configuration  $\sigma'' = (\beta, \sigma)$  by extracting the value of the position descriptor  $(pid, bid, bsize, bidx, pc)$  from the component  $\beta$  and applying successively  $bsize$  times the transition function of the language IL' to the configuration  $(\beta, \sigma)$ . In other words, the transition function extracts the information about the position of the flow of control in the CFGB and finds out that the flow of control is currently in the block  $bid$  and the length of  $bid$  is equal  $bsize$ . This means that the flow of control has to make  $bsize$  transitions from block position to block position in order to make one transition to the next block.
- The denotational semantics of an IL'' program is defined as a sequence of tokens printed by that program, i.e. we give the definition of a program semantics function which is a mapping from the set of IL'' programs to the set **ObservableBehavior**,

We proved for all IL' programs  $(P, B)$  and their corresponding IL'' programs  $(P, B)$  that if  $P$  is a well-typed IL program and  $B$  is a well-formed CFGB declaration w.r.t. to  $P$ , then their observable behaviors are equal.

*Bisimulation relation*: We formalized the notion of bisimulation relation  $\mathcal{R}$  between the sets of augmented configurations defined for the language IL''.

$$\mathcal{R} \in \mathbf{BisimulationRelation} ::= \mathcal{P}(\mathbf{Configuration} \times \mathbf{Configuration})$$

*Bisimulation predicate on pairs of IL'' program executions*: We formalized a bisimulation predicate **bisimulation** which takes two IL'' programs and a bisimulation relation and checks if the executions of the programs bisimulate w.r.t. the bisimulation relation.

$$\mathbf{bisimulation} : \mathbf{Program}'' \times \mathbf{Program}'' \times \mathbf{BisimulationRelation} \rightarrow \mathbf{Bool}$$

The formulation of the predicate is akin to bisimulation criteria in the literature on translation validation approach in which they are used by the validators as translation correctness rules.

*Optimization independent translation correctness criterion TCC:* We formalized the translation correctness criterion TCC as a predicate on two IL language programs as follows.

$$\exists \Phi_S \Phi_T B_S B_T \mathcal{R}. \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \text{bisimulation}((S, B_S), (T, B_T), \mathcal{R})$$

*Translation correctness theorem about bisimulation of IL" program executions:* We proved an optimization independent theorem whose statement has the following form:

$$\begin{aligned} &\exists \Phi_S \Phi_T B_S B_T \mathcal{R}. \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \\ &\quad \text{bisimulation}((S, B_S), (T, B_T), \mathcal{R}) \\ \implies & \\ &\text{corrTrans}(S, T) \end{aligned}$$

Thus, the formal framework provided by Layer 4 allows for proving the first step in our aforementioned proof scheme.

The rest of the section is organized as follows. Section 6.2 presents a formalization of the notion of a declaration of a control flow graph with blocks (CFGb) for an IL program. Section 6.3 presents a formalization of the notion of well-formedness of a CFGb declaration w.r.t. an IL program. The formalization of the language IL' and the theorem about the equality of the semantics of the languages IL and IL' are presented in Sections 6.4 and 6.5, respectively. Section 6.6 presents the formalization of the language IL". Section 6.7 presents a formalization of a bisimulation predicate on pairs of IL" program executions. Our optimization independent translation correctness criterion TCC is defined in terms on this notion. The definition of this criterion is presented in Section 6.8. The translation correctness theorem about bisimulation of program executions is presented in Section 6.9.

## 6.2 Block position environments

This section presents formalization of the notion of block position environments. In our work, we use block position environments to declare the structure of a control flow graph with blocks for an IL program. In the following, we list prerequisites which a formal model of the CFGb declaration has to adhere to, and which we took into account setting up the formalization of block position environments.

- A CFGb declaration for a program has to declare a finite set of blocks as a set of *block identifiers* and an allocation of program points of that program to blocks.
- In a CFGb declaration, the allocation of program points to blocks must be provably complete in the following sense:
  1. Each program point has to be allocated to at least one block. In case that a program point is included by a block which not nested in any other blocks, the program point must allocated exactly to that block. Otherwise, the CFGb declaration must allocate the program point to each block, in which it is included, separately. This means that the allocation mapping from blocks to program points must be surjective and therefore the CFGb declaration must declare for each pair  $(pc, bid)$  consisting of a program point  $pc$  and a block identifier  $bid$  a unique identifier (*block position identifier*).
  2. For each block, the CFGb declaration has to declare how many program points that block includes (*the length of the block*).
  3. For each block  $bid$  and program point  $pc$  which is included in  $bid$ , the CFGb declaration has to declare what is the index of  $pc$  within  $bid$  (*block index*).

*Example 6.1.* If the a block with block identifier  $bid$  has a length equal equal one, then it includes exactly one program point whose block index within  $bid$  is equal 0.  $\diamond$

- For the set of blocks, a CFGB declaration for a program has to declare
  - the successor and the predecessor relations between the sets of blocks,
  - the set of program points that are entry points of blocks in the block set,
  - a pair consisting of a program point and a block such that the program point is both the entry point of the program and the entry point of the block.
- In a CFGB declaration for an IL program, the allocation program points of that program to blocks and the successor relation between the block sets have to adhere to the successor relation between the program point sets defined by the set of edges of the CFG that corresponds to the program. The property of adherence must be provable.
- The presence of nested blocks in a CFGB declaration for an IL program means that if the flow of control reaches a program point  $pc$  at two different points of program execution and  $pc$  is included in a block  $b$  that is nested in another block  $b'$ , then it can be within one of two different blocks, i.e. it can be either in  $b$  or in  $b'$ . For this reason, we require that the formal model of the CFGB declaration is defined in such a way that we can prove the following statement for all CFGB declarations and IL programs: If a CFGB declaration fulfills the above completeness and adherence properties w.r.t. an IL program  $P$ , then the following conclusion is derivable from the hypotheses as follows.

1. *Hypothesis:* a configuration  $\sigma_n = (tf_n, af_n, pc_n, b_n, s_n)$  is the result of partial execution of  $P$ ,  $\exists n. M(P, \text{init}(P), n) = \sigma_n$ .
2. *Hypothesis:* program point  $pc_n$  is allocated to a block with identifier  $bid_i$  with a block index  $bidx_i$ .
3. *Hypothesis:* a configuration  $\sigma_{n+1} = (tf_{n+1}, af_{n+1}, pc_{n+1}, b_{n+1}, s_{n+1})$  is the result of program execution transition:  $\text{exec}(P, \sigma_n) = \sigma_{n+1}$ .

*Conclusion:* There exist a unique identifier  $bid_j$  and a unique value  $bidx_j$  such that  $pc_{n+1}$  is allocated as the  $bidx_j$ -th program point to the block  $bid_j$ .

To make this derivation possible, it must be provable within our formal model of the CFGB declaration that the values of  $bid_j$  and  $bidx_j$  can be deterministically computed from  $bid_i$ ,  $bidx_i$ ,  $pc_n$ , and  $pc_{n+1}$  at each point of program execution.

We begin the presentation of our formalization by listing the syntactic sets associated with the notion of block position environments:

- block identifiers **BlckId**,
- block sizes **BlckSize**,
- block position identifiers **BlckPosId**,
- block indexes **BlckIdx**,
- program points **Pc**,
- block position descriptors **BlckPosDescr**,
- block position status **BlckPosStat**,
- block beginning environments **BBEnv**,
- block position descriptor environments **BPEnv**,
- successor block position environments **succBPEnv**,
- buffer types **Buffertype**,
- predecessor block environments **predBEnv**,
- block position environments **BlckPosEnv**;

and defining metavariables ranging over these sets:

- $bid, predbid, succbid$  are ranging over block identifiers **BlckId**,
- $bsize$  is ranging block sizes **BlckSize**,
- $pid$  is ranging over block position identifiers **BlckPosId**,
- $bidx$  is ranging over block indexes **BlckIdx**,
- $pc$  is ranging over program points **Pc**,
- $bpos$  is ranging over block position descriptors **BlckPosDescr**,
- $bposstat$  is ranging over block position status **BlckPosStat**,
- $BB$  is ranging over block beginning environments **BBEnv**,
- $BP$  is ranging over environments of block position descriptors **BPEnv**,
- $succBP$  is ranging over successor block position environments **succBPEnv**,
- $bt$  is ranging over buffer types **Buffertype**,
- $predB$  is ranging over predecessor block environments **predBEnv**,
- $B$  is ranging over block position environments **BlckPosEnv**;

Definition 6.2 gives formation rules for the set of block position environments **BlckPosEnv**.

**Definition 6.2.**

<b>BlckId</b> $\ni bid$	$::= bid_0 \mid \dots \mid bid_n$
<b>BlckPosId</b> $\ni pid$	$::= pid_0 \mid \dots \mid pid_m$
	$pc \in \mathbf{Pc} = \mathbf{Nat}$
	$bsize \in \mathbf{BlckSize} = \mathbf{Nat}$
	$bidx \in \mathbf{BlckIdx} = \mathbf{Nat}$
<b>BlckPosDescr</b> $\ni bpos$	$::= (pid, bid, bsize, bidx, pc)$
	$BB \in \mathbf{BBEnv} = \mathbf{BlckId} \rightsquigarrow \mathbf{BlckPosId}$
	$BP \in \mathbf{BPEnv} = \mathbf{BlckPosId} \rightsquigarrow \mathbf{BlckPosDescr}$
	$succBP \in \mathbf{succBPEnv} = (\mathbf{BlckPosId} \times \mathbf{Pc}) \rightsquigarrow \mathbf{BlckPosId}$
<b>Buffertype</b> $\ni bt$	$::= \mathbf{FTYPE} \mid \mathbf{OTYPE}$
	$predB \in \mathbf{predBEnv} = \mathbf{BlckPosId} \rightsquigarrow \mathcal{P}(\mathbf{BlckId} \times \mathbf{Buffertype})$
<b>BlckPosEnv</b> $\ni B$	$::= (pid_0, BP, BB, succBP, predB)$

The starting point for the definition a block position environment  $B \in \mathbf{BlckPosEnv}$  which declares a CFGFB for an IL program are three finite sets:

1. The set of basic block identifiers **BlckId** comprises an identifier  $bid$  for each block in the node set of the CFGFB.
2. The set of block position identifiers **BlckPosId** comprises a unique identifier  $pid$  for each element of the "includes" relation between the set of blocks **BlckId** and the set of program points of the program and is defined as follows. For each pair  $(bid_i, pc_j)$  consisting of a block identifier  $bid_i \in \mathbf{BlckId}$  and a program point  $pc_j$  such that the block  $bid_i$  includes  $pc_j$ , there is a unique block position identifier  $pid_{i,j} \in \mathbf{BlckPosId}$ . The construction of **BlckPosId** imposes the following two intended logical consequences: Firstly, if a CFGFB declaration contains nested blocks and each block  $bid_i$  includes exactly one program point  $pc_j$ , then there exists a one-to-one correspondence between the set of program points and the power set of **BlckPosId** which maps each  $pc_j$  to the unique singleton  $\{pid_{i,j}\}$ . Secondly, if a CFGFB declaration contains nested blocks and there exists a program point  $pc_k$  and blocks  $bid_i$  and  $bid_j$  such that  $pc_k$  is included in  $bid_i$  and  $bid_i$  is nested in  $bid_j$ , then there exists a one-to-one correspondence between the set of program points and the the power set of **BlckPosId** which maps  $pc_k$  to the unique set  $\{pid_{i,k}, pid_{j,k}\}$ .
3. The set of block position descriptors **BlckPosDescr** comprises a unique tuple  $(pid, bid, bsize, bidx, pc)$  for each block position identifier  $pid \in \mathbf{BlckPosId}$ . Each such tuple describes elab-

orately the "includes" relationship  $(bid, pc)$  between a block  $bid$  and a program point  $pc$ . A tuple  $(pid, bid, bsize, bidx, pc)$  indicates the following:

- the "includes" relationship  $(bid, pc)$  between a block  $bid$  and a program point  $pc$  has a unique identifier  $pid$ ,
- the block  $bid$  includes  $bsize$  program points, i.e. that the length of the block  $bid$  is equal  $bsize$ , and
- $bidx$  is the index of the program point  $pc$  in the block, i.e. that the  $pc$ -th instruction of the instruction list is the  $bidx$ -th instruction of the block  $bid$ .

*Example 6.3.* Figure 6.1 depicts the instruction list of the program  $IL_3$  in Figure A.3, a CFG with basic blocks marked as dashed boxes, and a CFGB for the program  $IL_3$ . The program  $IL_3$  is the result of application of the optimizations CF, DAE, and NI to the program  $IL_0$  depicted in Figures A.1 and 1.4. For example, the chain of these optimizations replaced the 0-th instruction of the program  $IL_0$ ,  $[0]: \_tI\_1 = 0;$ , by the instruction  $[0]: GOTO 1;$ , which emulates nop instruction, cf. Figures A.1, A.2, A.3, . As for each well-formed IL program there exists a unique representation as CFG and for most of CFGs one can give at least one definition of basic blocks, the CFG is furnished with dashed boxes to visualize for which basic block definition is the CFGB in the figure. Figure 6.2 depicts the CFGB from Figure 6.1 and three sets which are the starting point for the declaration of that CFGB: the set of block identifiers **BlckId**, the set of block position identifiers **BlckPosId**, and the set of block position descriptors **BlckPosDescr**.

The first example: The program point 5 is included in the block with the identifier  $b_1$  with the block index equal 1 and the length of  $b_1$  is equal 3. For this reason, the sets **BlckPosId** and **BlckPosDescr** comprise the block position identifier  $p_{1,5}$  and the block position descriptor  $(p_{1,5}, b_1, 3, 1, 5)$ , respectively.

The second example: The program point 12 is included in the block  $b_6$  which is nested in the block  $b_3$ . The block indexes of the program point 12 in the blocks  $b_6$  and  $b_3$  are equal 0 and 1, respectively. The lengths of those blocks are equal 1 and 2 respectively. For this reason, the sets **BlckPosId** and **BlckPosDescr** comprise the block position identifiers  $p_{3,12}$  and  $p_{6,12}$  and the block position descriptors  $(p_{3,12}, b_3, 2, 1, 12)$  and  $(p_{6,12}, b_6, 1, 0, 12)$ , respectively.

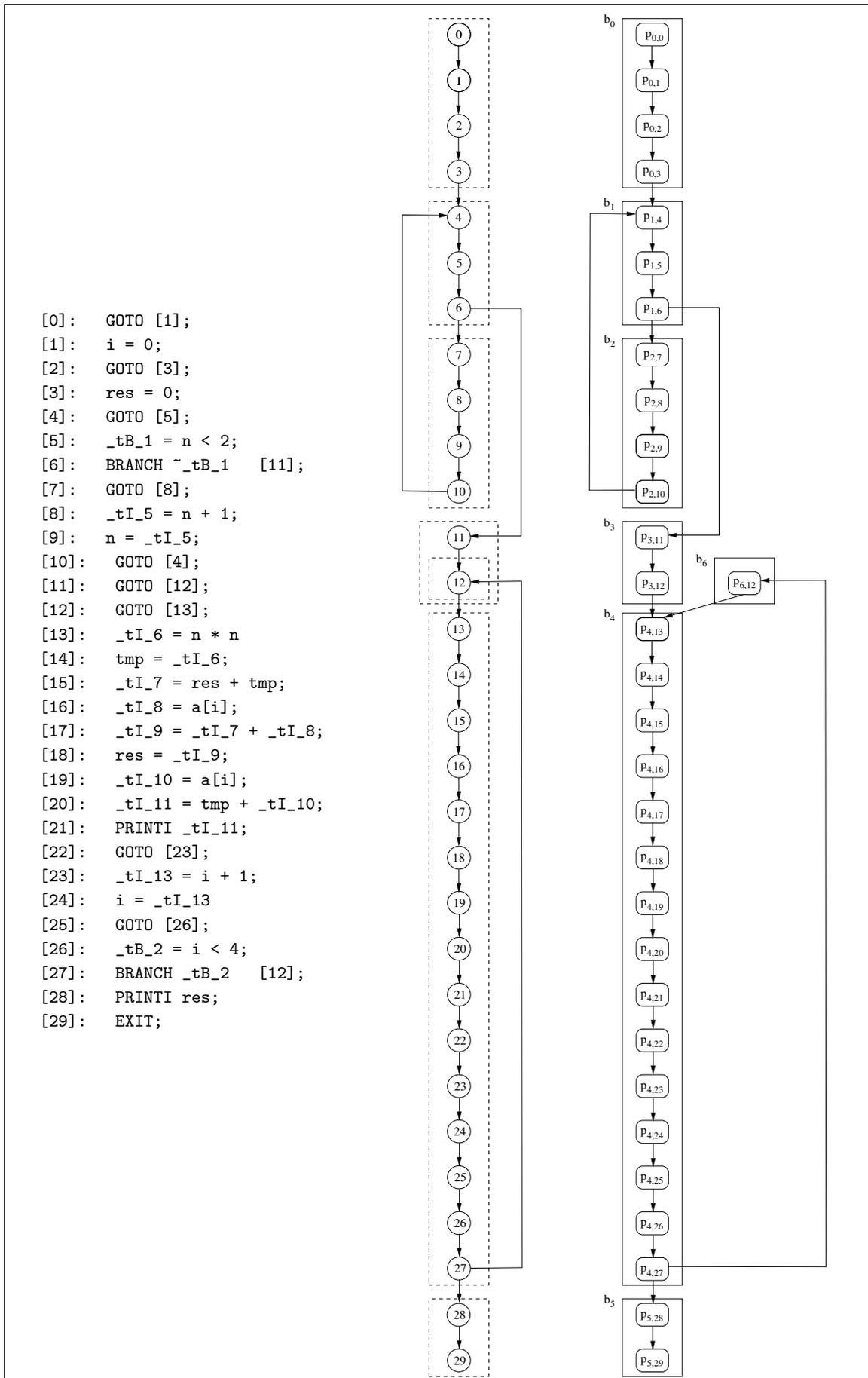
◇

The **BPEnv** set contains partial mappings from block position identifiers to block position descriptors, which we call *block position descriptor environments*. A well-formed block position environment  $BP \in \mathbf{BPEnv}$  is a total function which maps each  $pid \in \mathbf{BlckPosId}$  to a unique block position descriptor,  $BP(pid) = \text{Some}(pid', bid, bsize, bidx, pc)$  with  $pid = pid'$ , and its purpose is to model the "includes" relation between the set of blocks **BlckId** and the set of program points. Thus,  $BP$  can be used to either prove or disprove a statement that a block  $bid$  includes a block position  $pid$ .

*Example 6.4.* This example is the continuation of Example 6.3. Figure 6.3 depicts the CFGB from Figure 6.1 and the diagram of a function  $BP \in \mathbf{BPEnv}$  which is declared for that CFGB. For instance, the program point 5 is included in the block with the identifier  $b_1$  with the block index equal 1 and the length of  $b_1$  is equal 3. This relationship is described by the block position descriptor  $(p_{1,5}, b_1, 3, 1, 5)$ . For this reason,  $BP$  maps the block position identifier  $p_{1,5}$  to the block position descriptor  $(p_{1,5}, b_1, 3, 1, 5)$ .

◇

To declare a CFGB completely, we need to declare its set of edges. We do not, however, declare this set explicitly, for example by defining a successor relation over block pairs as a subset of  $\mathbf{BlckId} \times \mathbf{BlckId}$ , but rather in an implicit way, which is due to the purpose of our formalism,



**Fig. 6.1.** The instruction list of the program  $IL_3$  in Figure A.3, its CFG with a basic block definition marked as dashed boxes, and a CFGB which corresponds to the CFG in the middle of the figure.

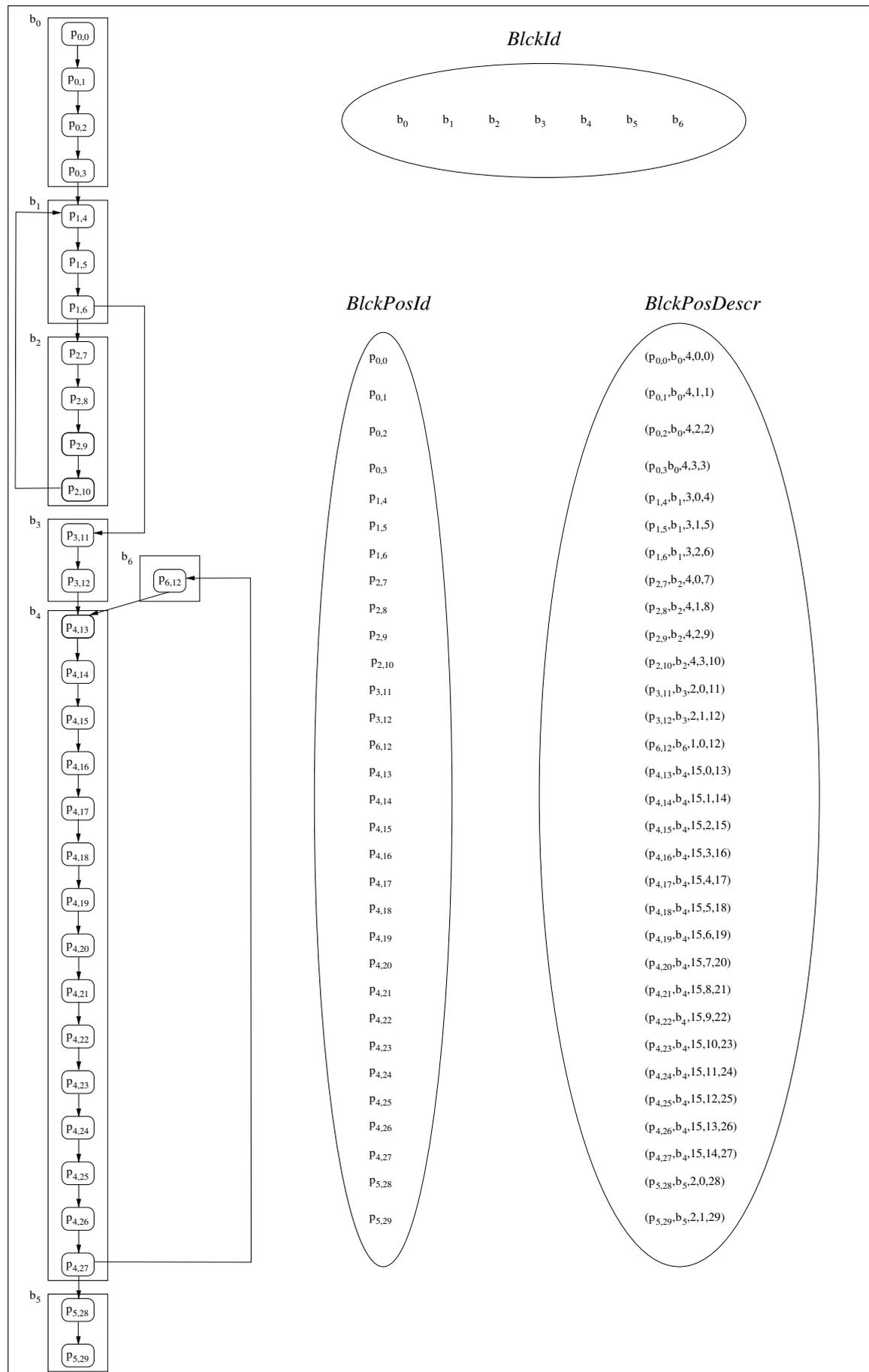
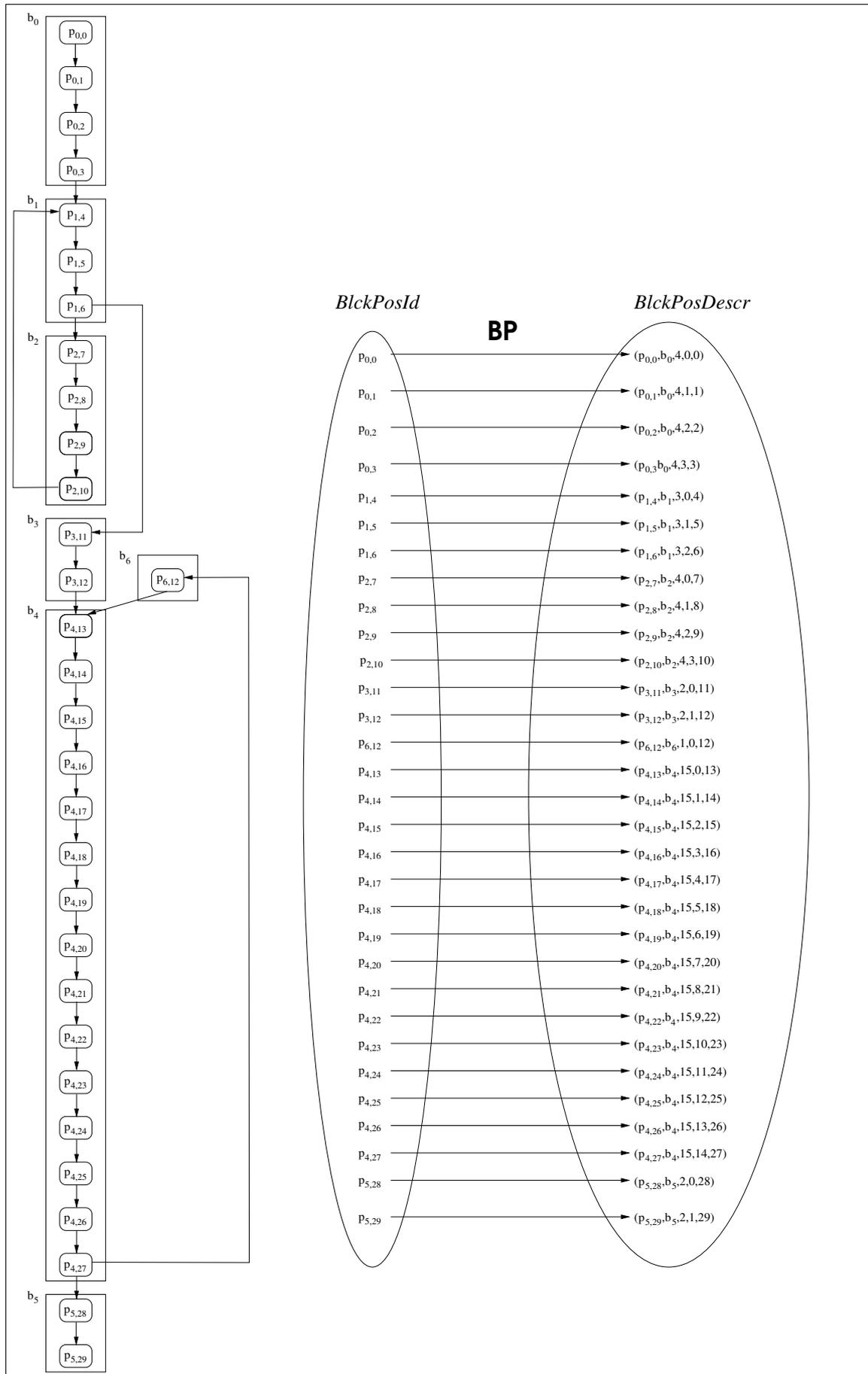


Fig. 6.2. The CFGB from Figure 6.1 and the sets **BlckId**, **BlckPosId**, and **BlckPosDescr**.



**Fig. 6.3.** The CFGB for the instruction list in Figure 6.1 and the declaration of a function  $BP \in \mathbf{BPEnv}$  for that CFGB.

which is to support formal reasoning about the correctness of optimizations. In particular, our framework has to enable one to do the following three things:

1. Expressing formally statements which describe transfers of the flow of control along the edges of a CFGB during partial execution of a program and how the current position of the flow of control in the CFGB has changed after making a transition. There follow some examples of such statements expressed informally:
  - a) "The flow of control has transferred from the block  $bid_i$  and enters the block  $bid_j$ ."
  - b) "The flow of control has transferred from the program point  $pc_j$ , which is included by the block  $bid$ , to the program point  $pc_j$  and remained in that block."
  - c) "The flow of control has transferred from the program point  $pc_i$ , which is included by the block  $bid_k$ , to the program point  $pc_j$ , which is included by another block  $bid_l$ ."
2. Expressing formally that a CFGB declared for a program is well-formed, i.e. it fulfills the aforementioned completeness and adherence properties. Below, Example 6.5 illustrates what does it mean that a concrete CFGB declaration adheres to the CFG that corresponds to a IL program:
3. The formalisation has to provide means which enables one to prove for a concrete CFGB declaration and a concrete IL program that the declaration is well-formed w.r.t. the program.

*Example 6.5.* Let us consider Figure 6.3. The pair of program points (7, 8) is in the set of edges of the CFG in Figure 6.3 and both 7 and 8 are included in the same block  $bid_2$  with the block indexes 0 and 1, respectively, and the length of the block  $bid_2$  is equal 4. Further, the **BkPosId** set comprises the block position identifiers  $p_{2,7}$  and  $p_{2,8}$  for the "includes" relationships  $(b_2, 7)$  and  $(b_2, 8)$ . Then, the edge  $(p_{2,7}, p_{2,8})$  in the CFGB adheres to the edge (7, 8) in the CFG iff the set **BkPosDescr** comprises the descriptors  $(pid_{2,7}, bid_2, 4, 0, 7)$  and  $(pid_{2,8}, bid_2, 4, 1, 8)$ ; and the block position descriptor environment  $BP$  maps  $pid_{2,7}$  and  $pid_{2,8}$  to  $(pid_{2,7}, bid_2, 4, 0, 7)$  and  $(pid_{2,8}, bid_2, 4, 1, 8)$ , respectively.

◇

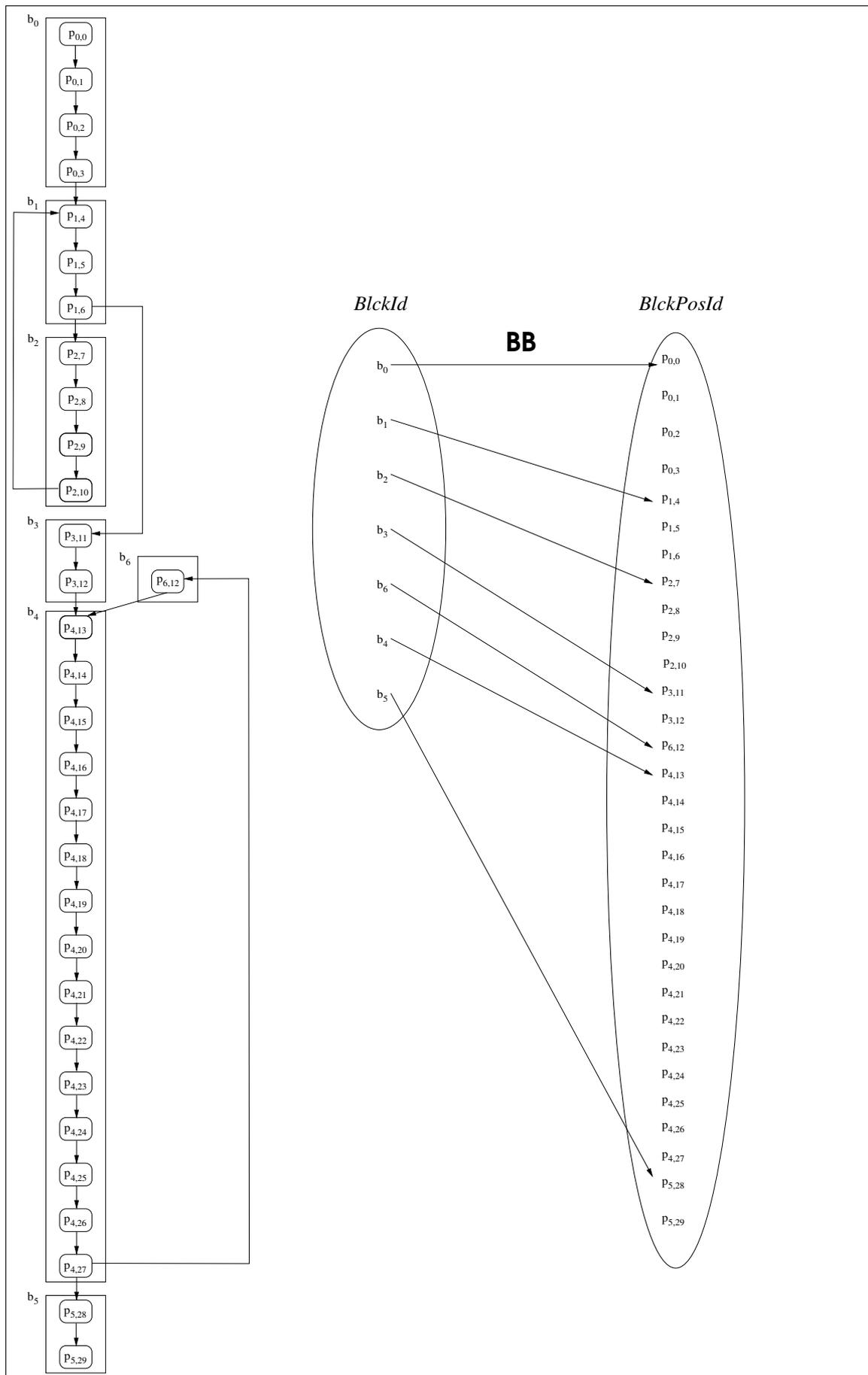
In the following, we explain formation rules for the sets **BBEnv**, **succBPEnv**, **Buffertype**, and **predBPEnv** which implicitly declare the set of edges of a control flow graph with blocks.

The **BBEnv** set contains partial mappings from block identifiers to block position identifiers which we call *block beginning environments*. A well-formed block beginning environment  $BB \in \mathbf{BBEnv}$  maps each block identifier  $bid \in \mathbf{BkId}$  to a well-defined block position identifier  $pid \in \mathbf{BkPosId}$  iff there exists a block length  $bsize$  and a program point  $pc$  such that  $BP(pid) = \text{Some}(pid, bid, bsize, 0, pc)$ . Thus,  $BB$  can be used to either prove or disprove that  $pid$  marks the beginning of the block  $bid$  and that program point  $pc$  is the entry point of the block  $bid$ .

*Example 6.6.* This example is the continuation of Examples 6.3, 6.4. Figure 6.4 depicts the CFGB from Figure 6.1 and the diagram of a function  $BB \in \mathbf{BBEnv}$  which is declared for that CFGB. For instance, the block  $b_2$  includes four block positions and the first block position in that block, i.e. a block position with the block index equal 0, is  $p_{2,7}$ . For this reason, the block beginning environment  $BB$  maps  $b_2$  to  $p_{2,7}$ .

◇

The **succBPEnv** set contains partial mappings from pairs  $(pid, pc)$  consisting of block position identifiers and program points to block position identifiers  $pid'$  which we call *successor block position environments*. The purpose of a mapping  $succBP \in \mathbf{succBPEnv}$  is to declare a successor relation over block position pairs that adheres to the successor relation between the sets of program points of a program. A well-formed successor block position environment



**Fig. 6.4.** The CFGB for the instruction list in Figure 6.1 and the declaration of a function  $BB \in \mathbf{BBEnv}$  for that CFGB.

$\text{succBP}$  maps a pair  $(pid, pc')$  to a block position identifier  $pid'$  iff the corresponding block position descriptor environment  $BP$  maps  $pid$  and  $pid'$  to descriptors  $(pid, bid, bsize, bidx, pc)$  and  $(pid', bid', bsize', bidx', pc')$ , respectively, and  $pc'$  is successor program point of  $pc$ . The following example illustrates this.

*Example 6.7.* This example is the continuation of Examples 6.3, 6.4, and 6.6. Figure 6.5 depicts the CFGB from Figure 6.1 and the diagram of a function  $\text{succBP} \in \mathbf{succBPEnv}$  which is declared for that CFGB. For instance, the CFGB comprises the edge  $(p_{4,27}, p_{6,12})$  which adheres to the edge  $(27, 12)$  in the CFG in Figure 6.1. For this reason, the successor block position environment  $\text{succBP}$  maps the pair  $(p_{4,27}, 12)$  to the block identifier  $p_{6,12}$ .  $\diamond$

The purpose of the set of buffer types **Buffertype** is to model two possible forms of the output buffer component  $b$  in the configuration  $\sigma = (tf, af, pc, b, s)$ . The value **FTYPE** is used as abstraction of the set of values of  $b$  which have the form  $\text{FLUSH}(n)$  and the value **OTYPE** as abstraction of the set of values of  $b$  which have the form  $\text{WRITE}(n, i)$ . The values **FTYPE** and **OTYPE** are used in a CFGB declaration to express that we can statically predict that whenever the flow of control transfers along an edge  $(pc, pc')$  of the corresponding CFG than the current state of the output buffer has a certain form.

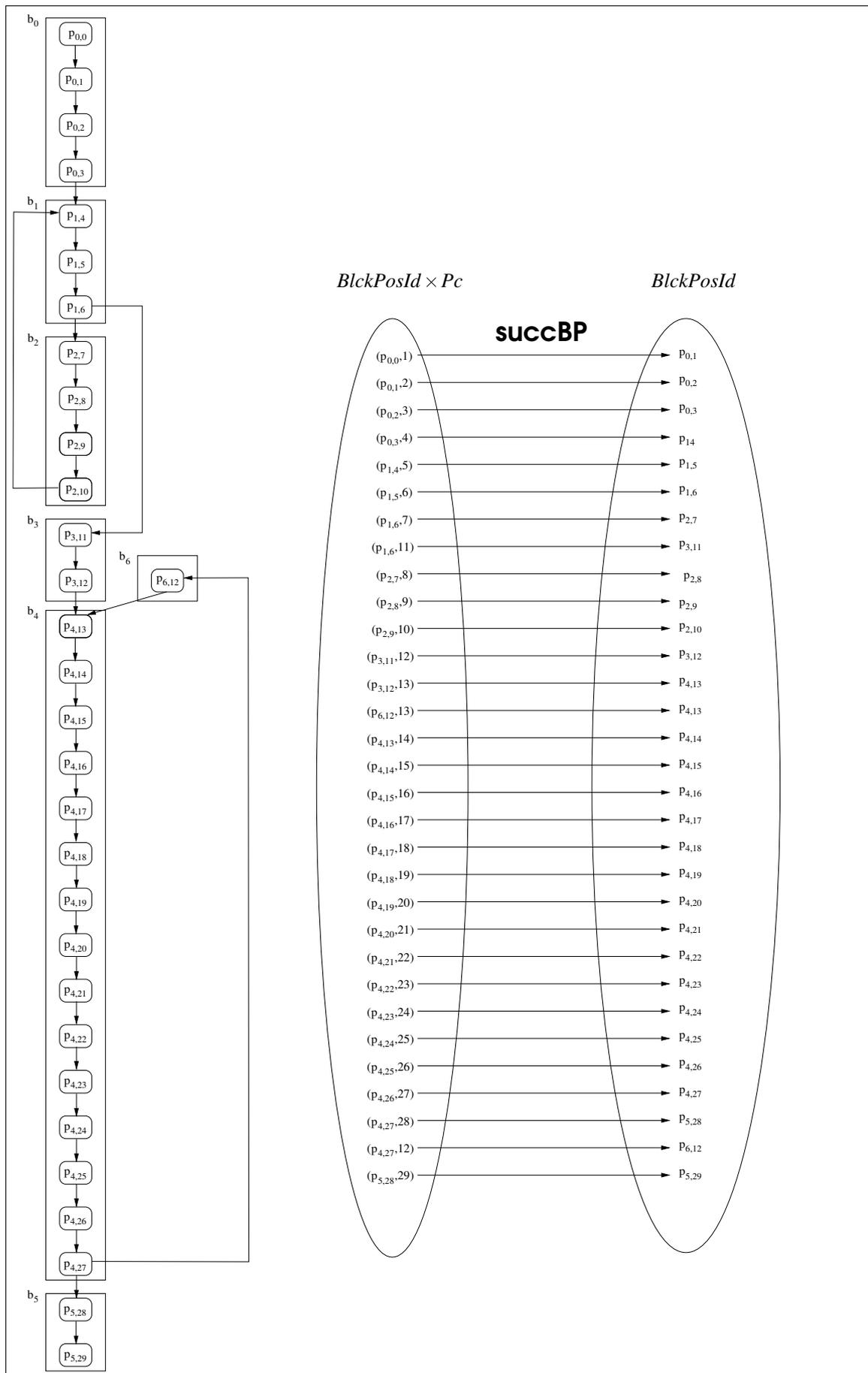
*Example 6.8.* It follows directly from the definition of the operational semantics of the IL language that if the  $pc$ -th instruction of a program is an assignment instruction than, each time the flow of control transfers from the program point  $pc$  to the program point  $pc + 1$ , the state of the output buffer has the form  $\text{FLUSH}(n)$ .  $\diamond$

*Example 6.9.* It follows directly from the definition of the operational semantics of the IL language that if the  $pc$ -th instruction of a program is a printi instruction than, each time the flow of control transfers from the program point  $pc$  to the program point  $pc + 1$ , the state of the output buffer has the form  $\text{WRITE}(n, i)$ .  $\diamond$

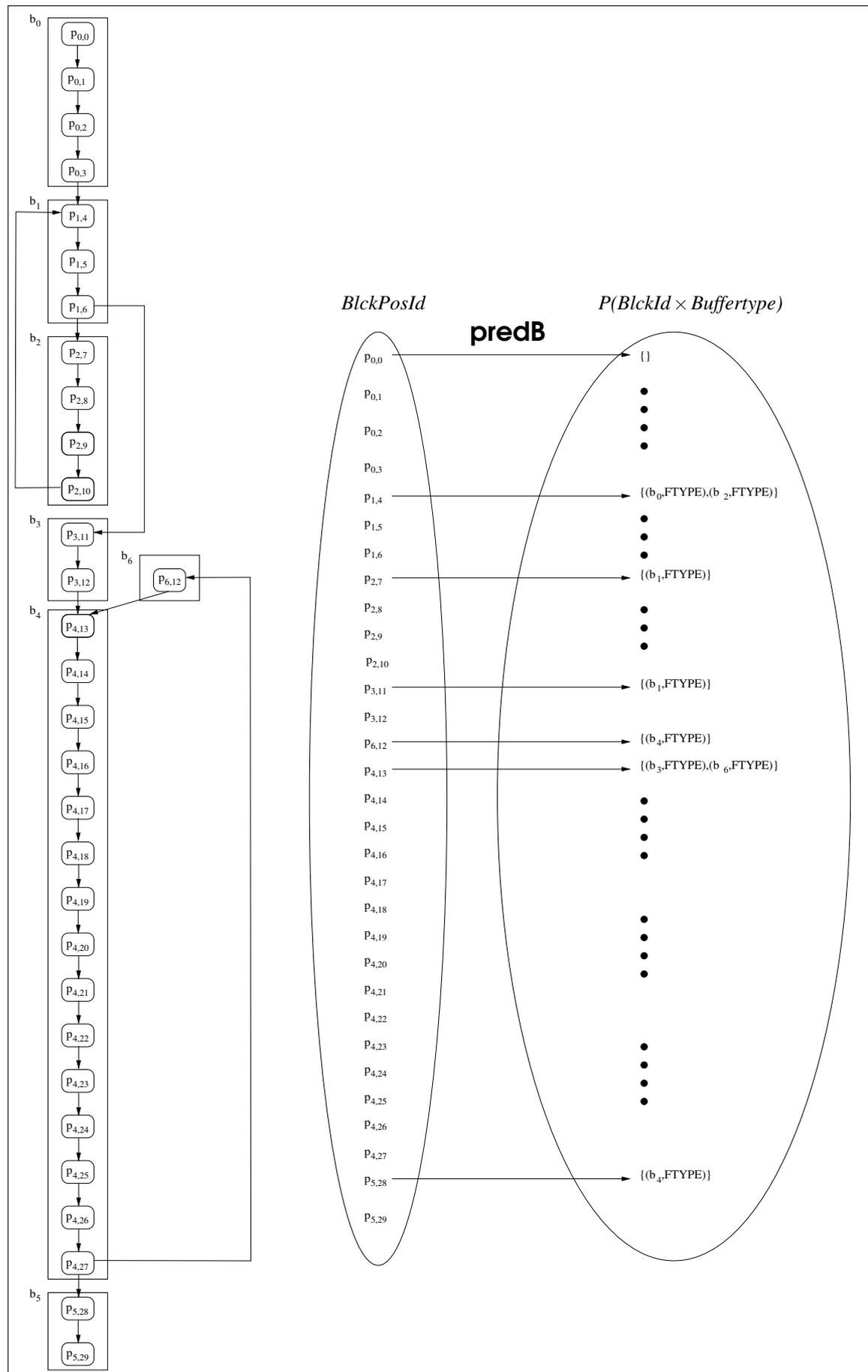
The **predBEnv** set contains partial mappings from block position identifiers  $pid$  to sets of pairs  $(bid, bt)$  consisting of block identifiers and buffertypes. We call these mappings *predecessor block environments*. The purpose of an environment  $\text{predB} \in \mathbf{predBEnv}$  is to declare the set of all predecessor blocks for a block position  $pid$  which has the block index equal 0 in some block  $bid$  (i.e. marks is an entry block position of  $bid$ ) together with the form of the output buffer  $b$  in the current state of program execution  $\sigma$  whenever the flow of control flows into  $bid$  making transition along the edges connecting these blocks with  $bid$ .

*Example 6.10.* This example is the continuation of Examples 6.3, 6.4, 6.6, and 6.7. Figure 6.6 depicts the CFGB from Figure 6.1 and the diagram of a function  $\text{predB} \in \mathbf{predBEnv}$  which is declared for that CFGB. For instance, the predecessor block environment  $\text{predB}$  in Figure 6.6 maps  $p_{4,13}$  to the set  $\{(b_3, \text{FTYPE}), (b_6, \text{FTYPE})\}$  for the following reasons:

1. The 12-th instruction of the instruction list in Figure 6.3 is  $[12]: \text{GOTO } [13]$ ,
2. The program point 12 is allocated to the block positions  $p_{3,12}$  and  $p_{6,12}$ ,
3.  $p_{3,12}$  and  $p_{6,12}$  are the last block positions in the blocks  $b_3$  and  $b_6$ , respectively,
4. The block position  $p_{4,13}$  has the block index 0 in the block  $b_4$ , and
5. The CFGB comprises the edges  $(p_{3,12}, p_{4,13})$  and  $(p_{6,12}, p_{4,13})$ .  $\diamond$



**Fig. 6.5.** The CFGB for the instruction list in Figure 6.1 and the declaration of a function  $succBP \in succBPEnv$  for that CFGB.



**Fig. 6.6.** The CFGB for the instruction list in Figure 6.1 and the declaration of a function  $predB \in predBEnv$  for that CFGB.

In our work, we declare a CFGB for an IL program  $P$  as a block position environment  $B \in \mathbf{BlkPosEnv}$ , where  $B$  is defined as a tuple  $(pid_0, BP, BB, succBP, predB)$  whose components denote the following:

- $pid_0$  denotes a block position identifier of the "include" relationship between a program point 0 which is both the entry point of  $P$  and the entry point of the entry block of the CFGB,
- $BP$  denotes a block position descriptor environment,
- $BB$  denotes a block beginning environment,
- $succBP$  denotes a successor block position environment, and
- $predB$  denotes a predecessor block environment.

If a CFGB declaration  $(pid_0, BP, BB, succBP, predB)$  is well-formed, then the entry block of the CFGB can be determined by looking up in the set of block position descriptors  $BP$  for the descriptor of the block position  $pid_0$ . The result has always the same form,  $BP(pid_0) = \text{Some}(pid_0, bid_0, bsize_0, 0, 0)$  where  $bid_0$  and  $bsize_0$  denote the identifier of the entry block and its length, respectively, as 0-th instruction of a program is always established as the entry point of the program and the entry point of the entry block.

The following section presents the formalization of the notion of well-formedness of a CFGB declaration.

### 6.3 Well-formedness of block position environments

This section presents the formalization of the notion well-formedness of a CFGB declaration w.r.t. a IL program.

As aforementioned, the starting point for declaring of a block position environment  $B$  for an IL program  $P$  are three finite sets:

1. the set of block identifiers  $\mathbf{BlkId}$ ,
2. the set of block position identifiers  $\mathbf{BlkPosId}$ , and
3. the set of block position descriptors  $\mathbf{BlkPosDescr}$

where the sets  $\mathbf{BlkPosId}$  and  $\mathbf{BlkPosDescr}$  must fulfill the completeness property w.r.t. the block set  $\mathbf{BlkId}$  and an IL program.

Definition 6.11 defines an auxiliary function  $pc2pidset$  which we use to check if the sets  $\mathbf{BlkId}$ ,  $\mathbf{BlkPosId}$ , and  $\mathbf{BlkPosDescr}$  defined for a CFGB declaration fulfill the completeness property w.r.t. an IL program.

**Definition 6.11.**

$$pc2pidset : \mathbf{BPEnv} \times \mathbf{Pc} \rightarrow \mathcal{P}(\mathbf{BlkPosId})$$

$$pc2pidset(BP, pc) = \{pid \mid \exists bid \ bsize \ bidx. BP(pid) = \text{Some}(pid, bid, bsize, bidx, pc)\}$$

*Example 6.12.* This example is the continuation of Examples 6.3 and 6.4. Figure 6.3 depicts the CFGB from Figure 6.1 and the diagram of a function  $BP \in \mathbf{BPEnv}$  which is declared for that CFGB. For instance, as the program point 11 is included in only one block  $b_3$  the program point 12 is included in nested blocks  $b_6$  and  $b_3$ , it holds the following for the function  $BP$ :

1.  $pc2pidset(BP, 11) = \{p_{3,11}\}$  and
2.  $pc2pidset(BP, 12) = \{p_{3,12}, p_{6,12}\}$ .

◇

Using the function `pc2pidset`, we can formally express the completeness property of a CFGB declaration  $(pid_0, BP, BB, succBP, predB)$  w.r.t. an IL program  $((vds, instrs), I)$  as follows.

$$\begin{aligned} & \forall pc. 0 \leq pc < \text{length}(instrs) \longrightarrow \text{pc2pidset}(BP, pc) \neq \{\} \wedge \\ & \forall pid. \exists bid \ bsize \ bidx \ pc. BP(pid) = \text{Some}(pid, bid, bsize, bidx, pc) \wedge 0 \leq bidx \wedge bidx < bsize \end{aligned}$$

In the following, we present definitions of predicates which we use to check if a CFGB declaration for a program adheres to the CFG which corresponds to that program.

Definitions 6.13 and 6.15 define two auxiliary predicates `wfB_succpc_in_blk` and `wfB_succpc_next_blk` which we use to check if a pair  $(pid, succpid)$  consisting of two block position identifiers which are in the successor relationship in a CFGB declaration adheres to an edge  $(pc, succpc)$  of a CFG.

Let us consider an edge  $(pc, succpc)$  in the edge set of a CFG, a block  $bid$ , and a block position identifier  $pid$  such that  $bid$  has the length  $bsize$ ; and  $bid$  includes  $pc$ ; and  $pid$  is the unique identifier of the "includes" relationship  $(bid, pc)$ . Then, in a well-formed CFGB declaration, the program point  $succpc$  is also allocated to a block  $bid'$  (according to the completeness property of CFGB) and there are two possibilities for  $bid'$ :

1. The program point  $pc$  is not the last program point included in the block  $bid$ . Consequentially,  $succpc$  must also be included in  $bid$  and it holds  $bid = bid'$ .
2. The program point  $pc$  is the last program point included in the block  $bid$ . Consequentially,  $succpc$  must be the entry point of  $bid'$  and it holds  $bid \neq bid'$ .

The predicate `wfB_succpc_in_blk` formalizes the property of adherence of a block position pair  $(pid, succpid)$  to a CFG edge  $(pc, succpc)$  in the former case. The predicate `wfB_succpc_next_blk` formalizes the property of adherence of a block position pair  $(pid, succpid)$  to a CFG edge  $(pc, succpc)$  in the latter case.

In the definition of `wfB_succpc_in_blk`, the conjuncts  $BP(pid) = \text{Some}(pid, bid, bsize, bidx, pc)$  and  $BP(succpid) = \text{Some}(succpid, bid, bsize, bidx', succpc)$  state that the program points  $pc$  and  $succpc$  are allocated to the block positions  $pid$  and  $succpid$ , respectively, which are included in the same block  $bid$  with the block indexes  $bidx$  and  $bidx'$ , respectively. The conjuncts  $succBP(pid, succpc) = \text{Some}(succpid)$  and  $\text{Suc}(bidx) = bidx'$  states that  $succpid$  is declared in the CFGB declaration  $(pid_0, BP, BB, succBP, predB)$  as the successor block position of  $pid$  w.r.t. the successor program point  $succpc$  and that  $succpid$  is the successor of  $pid$  within the block  $bid$ . The conjunct  $succpid \in \text{pc2pidset}(BP, succpc)$  states that  $succpid$  is one of the block positions to which the program point  $succpc$  is allocated to.

**Definition 6.13.**

$$\begin{aligned} & \text{wfB\_succpc\_in\_blk} : \mathbf{BPEnv} \times \mathbf{succBPEnv} \times \mathbf{BlckPosId} \times \mathbf{BlckPosId} \times \mathbf{Pc} \times \mathbf{Pc} \rightarrow \mathbf{Bool} \\ & \text{wfB\_succpc\_in\_blk}(BP, succBP, pid, succpid, pc, succpc) = \\ & \quad \exists bid \ bsize \ bidx \ bidx'. \\ & \quad \quad BP(pid) = \text{Some}(pid, bid, bsize, bidx, pc) \wedge \\ & \quad \quad BP(succpid) = \text{Some}(succpid, bid, bsize, bidx', succpc) \wedge \\ & \quad \quad succBP(pid, succpc) = \text{Some}(succpid) \wedge \\ & \quad \quad succpid \in \text{pc2pidset}(BP, succpc) \wedge \\ & \quad \quad \text{Suc}(bidx) = bidx' \end{aligned}$$

*Example 6.14.* This example is the continuation of Examples 6.3, 6.4, 6.7, and 6.12. Figures 6.3 and 6.5 depicts the CFGB from Figure 6.1 and the diagrams of the function  $BP \in \mathbf{BPEnv}$  and  $succBP \in \mathbf{succBPEnv}$  which are declared for that CFGB. For instance, using the definitions of

those functions, we can prove that the block position pair  $(p_{3,11}, p_{3,12})$  adheres to the CFG edge  $(11, 12)$ :

$$\text{wfB\_succpc\_in\_blk}(BP, \text{succBP}, p_{3,11}, p_{3,12}, 11, 12)$$

*Proof.*

$$\text{wfB\_succpc\_in\_blk}(BP, \text{succBP}, p_{3,11}, p_{3,12}, 11, 12)$$

$$\Leftarrow [\text{by the definition of wfB\_succpc\_in\_blk}]$$

$$\exists \text{bid bsize bidx bidx}'.$$

$$BP(p_{3,11}) = \text{Some}(p_{3,11}, \text{bid}, \text{bsize}, \text{bidx}, 11) \wedge$$

$$BP(p_{3,12}) = \text{Some}(p_{3,12}, \text{bid}, \text{bsize}, \text{bidx}', 12) \wedge$$

$$\text{succBP}(p_{3,11}, 12) = \text{Some}(p_{3,12}) \wedge$$

$$p_{3,12} \in \text{pc2pidset}(BP, p_{3,12}) \wedge$$

$$\text{Suc}(\text{bidx}) = \text{bidx}'$$

$$\Leftarrow [\text{by the existential introduction rule}]$$

$$BP(p_{3,11}) = \text{Some}(p_{3,11}, b_3, 2, 0, 11) \wedge$$

$$BP(p_{3,12}) = \text{Some}(p_{3,12}, b_3, 2, 1, 12) \wedge$$

$$\text{succBP}(p_{3,11}, 12) = \text{Some}(p_{3,12}) \wedge$$

$$p_{3,12} \in \text{pc2pidset}(BP, p_{3,12}) \wedge$$

$$\text{Suc}(0) = 1$$

$$\Leftarrow [\text{by the definition of pc2pidset and BP}]$$

$$BP(p_{3,11}) = \text{Some}(p_{3,11}, b_3, 2, 0, 11) \wedge$$

$$BP(p_{3,12}) = \text{Some}(p_{3,12}, b_3, 2, 1, 12) \wedge$$

$$\text{succBP}(p_{3,11}, 12) = \text{Some}(p_{3,12}) \wedge$$

$$p_{3,12} \in \{p_{3,12}, p_{6,12}\} \wedge$$

$$\text{Suc}(0) = 1$$

This holds by the definitions of  $BP$  and  $\text{succBP}$ ; and by the definition of 1 in HOL. □

◇

In the definition of  $\text{wfB\_succpc\_next\_blk}$ , the conjunct  $BP(\text{succpid}) = \text{Some}(\text{succpid}, \text{bid}', \text{bsize}', 0, \text{succpc})$  states the program point  $\text{succpc}$  is allocated to the block positions  $\text{succpid}$  which has the block index equal 0 in the block  $\text{bid}'$ , i.e.  $\text{succpc}$  is the entry point of  $\text{bid}'$ . The conjunct  $\text{succBP}(\text{pid}, \text{succpc}) = \text{Some}(\text{succpid})$  states that  $\text{succpid}$  is declared in the CFGB declaration  $(\text{pid}_0, BP, BB, \text{succBP}, \text{predB})$  as the successor block position of  $\text{pid}$  w.r.t. the successor program point  $\text{succpc}$ . The conjunct  $\text{succpid} \in \text{pc2pidset}(BP, \text{succpc})$  states that  $\text{succpid}$  is one of the block positions to which the program point  $\text{succpc}$  is allocated to.

**Definition 6.15.**

$$\text{wfB\_succpc\_next\_blk} : \mathbf{BPEnv} \times \mathbf{succBPEnv} \times \mathbf{BlkPosId} \times \mathbf{BlkPosId} \times \mathbf{Pc} \times \mathbf{Pc} \rightarrow \mathbf{Bool}$$

$$\text{wfB\_succpc\_next\_blk}(BP, \text{succBP}, \text{pid}, \text{succpid}, \text{pc}, \text{succpc}) =$$

$$\exists \text{bid}' \text{bsize}'.$$

$$BP(\text{succpid}) = \text{Some}(\text{succpid}, \text{bid}', \text{bsize}', 0, \text{succpc}) \wedge$$

$$\text{succBP}(\text{pid}, \text{succpc}) = \text{Some}(\text{succpid}) \wedge$$

$$\text{succpid} \in \text{pc2pidset}(BP, \text{succpc})$$

*Example 6.16.* This example is the continuation of Examples 6.3, 6.4, 6.7, and 6.12. Figures 6.3 and 6.5 depicts the CFGB from Figure 6.1 and the diagrams of the function  $BP \in \mathbf{BPEnv}$  and  $succBP \in \mathbf{succBPEnv}$  which are declared for that CFGB. For instance, using the definitions of those functions, we can prove that the block position pair  $(p_{3,12}, p_{4,13})$  adheres to the CFG edge  $(12, 13)$ :

$$\text{wfB\_succpc\_next\_blk}(BP, succBP, p_{3,12}, p_{4,13}, 12, 13)$$

*Proof.*

$$\text{wfB\_succpc\_next\_blk}(BP, succBP, p_{3,12}, p_{4,13}, 12, 13)$$

$$\Leftarrow [\text{by the definition of wfB\_succpc\_next\_blk}]$$

$$\exists bid' bsize'$$

$$BP(p_{4,13}) = \text{Some}(p_{4,13}, bid', bsize', 0, 13) \wedge$$

$$succBP(p_{3,12}, 13) = \text{Some}(p_{4,13}) \wedge$$

$$p_{4,13} \in \text{pc2pidset}(BP, 13)$$

$$\Leftarrow [\text{by the existential introduction rule}]$$

$$BP(p_{4,13}) = \text{Some}(p_{4,13}, bid', bsize', 0, 13) \wedge$$

$$succBP(p_{3,12}, 13) = \text{Some}(p_{4,13}) \wedge$$

$$p_{4,13} \in \text{pc2pidset}(BP, 13)$$

$$\Leftarrow [\text{by the definition of pc2pidset and BP}]$$

$$BP(p_{4,13}) = \text{Some}(p_{4,13}, bid', bsize', 0, 13) \wedge$$

$$succBP(p_{3,12}, 13) = \text{Some}(p_{4,13}) \wedge$$

$$p_{4,13} \in \{p_{4,13}\}$$

This holds by the definitions of  $BP$  and  $succBP$ .

□  
◇

Definition 6.17 defines a predicate  $\text{wfBinstr}$  which takes a CFGB declaration  $(pid_0, BP, BB, succBP, predB)$ , a program point  $pc$ , and an instruction  $instr$  as input and checks if the set of block positions to whom  $pc$  is allocated,  $\text{pc2pidset}(BP, pc)$ , adheres to  $pc$  w.r.t. the operational semantics of the instruction  $instr$ . The property of adherence of  $\text{pc2pidset}(BP, pc)$  to  $pc$  is defined as follows.

The program point  $pc$  is interpreted as a node of a CFG that corresponds to an IL program. For this reason, the predicate has an auxiliary argument, the  $pc$ -th instruction of the program  $instr$ , which is needed to determine the set of CFG edges  $(pc, succpc)$  in which  $pc$  is in the predecessor relationship with  $succpc$ . The edge set is determined in accordance with the operational semantics of  $instr$ . For each edge  $(pc, succpc)$  in the set, the CFGB declaration must declare a block position edge  $(pid, succpid)$  such that  $pc$  is allocated to the block position  $pid$ , i.e.  $pid \in \text{pc2pidset}(BP, pc)$ , and the edge  $(pid, succpid)$  adheres to the edge  $(pc, succpc)$ .

**Definition 6.17.**

$$\begin{aligned}
& \mathbf{wfBinstr} : \mathbf{BlckPosEnv} \times \mathbf{Pc} \times \mathbf{Instruction} \rightarrow \mathbf{Bool} \\
& \mathbf{wfBinstr}((pid_0, BP, BB, succBP, predB), pc, lv:=e) = \\
& \quad \forall pid \in \mathbf{pc2pidset}(BP, pc). \\
& \quad \quad \exists bid \ bsize \ bidx. \\
& \quad \quad \quad BP(pid) = \mathbf{Some}(pid, bid, bsize, bidx, pc) \wedge \\
& \quad \quad \quad \mathbf{if} \ \mathbf{Suc}(bidx) < bsize \\
& \quad \quad \quad \mathbf{then} \ \exists succpid. \ \mathbf{wfB\_succpc\_in\_blk}(BP, succBP, pid, succpid, pc, \mathbf{Suc}(pc)) \\
& \quad \quad \quad \mathbf{else} \ \exists succpid. \ \mathbf{wfB\_succpc\_next\_blk}(BP, succBP, pid, succpid, pc, \mathbf{Suc}(pc)) \\
& \\
& \mathbf{wfBinstr}((pid_0, BP, BB, succBP, predB), pc, \mathbf{printi}(e)) = \\
& \quad \forall pid \in \mathbf{pc2pidset}(BP, pc). \\
& \quad \quad \exists bid \ bsize \ bidx. \\
& \quad \quad \quad BP(pid) = \mathbf{Some}(pid, bid, bsize, bidx, pc) \wedge \\
& \quad \quad \quad \mathbf{if} \ \mathbf{Suc}(bidx) < bsize \\
& \quad \quad \quad \mathbf{then} \ \exists succpid. \ \mathbf{wfB\_succpc\_in\_blk}(BP, succBP, pid, succpid, pc, \mathbf{Suc}(pc)) \\
& \quad \quad \quad \mathbf{else} \ \exists succpid. \ \mathbf{wfB\_succpc\_next\_blk}(BP, succBP, pid, succpid, pc, \mathbf{Suc}(pc)) \\
& \\
& \mathbf{wfBinstr}((pid_0, BP, BB, succBP, predB), pc, \mathbf{branch}(e, dst)) = \\
& \quad \forall pid \in \mathbf{pc2pidset}(BP, pc). \\
& \quad \quad \exists bid \ bsize \ bidx. \\
& \quad \quad \quad BP(pid) = \mathbf{Some}(pid, bid, bsize, bidx, pc) \wedge \\
& \quad \quad \quad \mathbf{if} \ \mathbf{Suc}(bidx) < bsize \\
& \quad \quad \quad \mathbf{then} \ \exists succpid. \ \mathbf{wfB\_succpc\_in\_blk}(BP, succBP, pid, succpid, pc, \mathbf{Suc}(pc)) \wedge \\
& \quad \quad \quad \quad \exists succpid. \ \mathbf{wfB\_succpc\_in\_blk}(BP, succBP, pid, succpid, pc, dst) \\
& \quad \quad \quad \mathbf{else} \ \exists succpid. \ \mathbf{wfB\_succpc\_next\_blk}(BP, succBP, pid, succpid, pc, \mathbf{Suc}(pc)) \wedge \\
& \quad \quad \quad \quad \exists succpid. \ \mathbf{wfB\_succpc\_next\_blk}(BP, succBP, pid, succpid, pc, dst) \\
& \\
& \mathbf{wfBinstr}((pid_0, BP, BB, succBP, predB), pc, \mathbf{goto}(dst)) = \\
& \quad \forall pid \in \mathbf{pc2pidset}(BP, pc). \\
& \quad \quad \exists pid' \ bid \ bsize \ bidx \ pc'. \\
& \quad \quad \quad BP(pid) = \mathbf{Some}(pid', bid, bsize, bidx, pc') \wedge \\
& \quad \quad \quad pc = pc' \wedge \\
& \quad \quad \quad \mathbf{if} \ \mathbf{Suc}(bidx) < bsize \\
& \quad \quad \quad \mathbf{then} \ \exists succpid. \ \mathbf{wfB\_succpc\_in\_blk}(BP, succBP, pid, succpid, dst) \\
& \quad \quad \quad \mathbf{else} \ \exists succpid. \ \mathbf{wfB\_succpc\_next\_blk}(BP, succBP, pid, succpid, dst) \\
& \\
& \mathbf{wfBinstr}((pid_0, BP, BB, succBP, predB), pc, \mathbf{exit}) = \\
& \quad \forall pid \in \mathbf{pc2pidset}(BP, pc). \\
& \quad \quad \exists pid' \ bid \ bsize \ bidx \ pc'. \\
& \quad \quad \quad BP(pid) = \mathbf{Some}(pid', bid, bsize, bidx, pc') \wedge \\
& \quad \quad \quad \mathbf{Suc}(bidx) = bsize
\end{aligned}$$

Definition 6.18 defines a predicate  $\mathbf{wfBinstrs}$  which checks if a CFGB declaration  $(pid_0, BP, BB, succBP, predB)$  is complete w.r.t. to an instruction list  $instrs$  and the successor relation over block position pairs, which is declared by  $succBP$ , adheres to the set of node edges of the CFG which corresponds to  $instrs$ . The former property is defined as follows.  $(pid_0, BP, BB, succBP, predB)$  is complete w.r.t.  $instrs$  iff each program point  $pc$  of  $instrs$  is allocated to at least one block position, i.e. the set  $\mathbf{pc2pidset}(BP, pc)$  must not be empty for each  $pc$ . The latter property is defined as follows.  $(pid_0, BP, BB, succBP, predB)$  adheres to the set of node edges of the CFG which corresponds to  $instrs$  iff it holds for each program point of  $instrs$ ,  $pc$ , that the set of block

positions  $\text{pc2pidset}(BP, pc)$  adheres to  $pc$  w.r.t. the operational semantics of the  $pc$ -th instruction of  $instrs$ .

**Definition 6.18.**

$$\begin{aligned} \text{wfBinstrs} &: \mathbf{BlckPosEnv} \times \mathbf{InstructionList} \rightarrow \mathbf{Bool} \\ \text{wfBinstrs}((pid_0, BP, BB, succBP, predB), instrs) &= \\ &\forall pc. 0 \leq pc < \text{length}(instrs) \longrightarrow \text{pc2pidset}(BP, pc) \neq \{\} \wedge \\ &\quad \text{wfBinstr}((pid_0, BP, BB, succBP, predB), pc, instrs!pc) \end{aligned}$$

Definition 6.19 defines a predicate  $\text{wfB}$  which checks if a CFGB declaration  $(pid_0, BP, BB, succBP, predB)$  is well-formed w.r.t. an IL program  $((vds, instrs), I)$ . The declaration  $(pid_0, BP, BB, succBP, predB)$  is well-formed w.r.t.  $(pid_0, BP, BB, succBP, predB)$  iff it fulfills the following requirements:

- $BP$  is an injective mapping from block positions to block position descriptors, i.e. we require the following:
  - For each block position  $pid$  there exists a block well-formed descriptor  $(pid, bid, bsize, bidx, pc)$  which describes the "includes" relationship between a block  $bid$  and a program point  $pc$  with a valid block index  $bidx$  and
  - if the descriptor's block index component  $bidx$  is equal 0, then the block position  $pid$  is declared by the mapping  $BB$  as the entry of the block  $bid$ .
- For each block  $bid$  there exists a block position  $pid$  which is the entry of  $bid$ .
- The program point 0 is allocated to the block position  $pid_0$  which is the entry of the entry block of the CFGB.
- The CFGB declaration is both complete w.r.t. to the instruction list  $instrs$  and the successor relation over block position pairs, which is declared by  $succBP$ , adheres to the set of node edges of the CFG which corresponds to  $instrs$ .

**Definition 6.19.**

$$\begin{aligned} \text{wfB} &: \mathbf{Program} \times \mathbf{BlckPosEnv} \rightarrow \mathbf{Bool} \\ \text{wfB}(((vds, instrs), I), (pid_0, BP, BB, succBP, predB)) &= \\ &\forall pid. \exists bid \ bsize \ bidx \ pc. \\ &\quad BP(pid) = \mathbf{Some}(pid, bid, bsize, bidx, pc) \wedge \\ &\quad 0 \leq bidx \wedge \\ &\quad bidx < bsize \wedge \\ &\quad bidx = 0 \longrightarrow BB(bid) = \mathbf{Some}(pid) \\ &\wedge \\ &\forall bid. \exists pid \ bid \ bsize \ pc. \\ &\quad BB(bid) = \mathbf{Some}(pid) \wedge \\ &\quad BP(pid) = \mathbf{Some}(pid, bid, bsize, 0, pc) \\ &\wedge \\ &\exists bid_0 \ bsize_0. BP(pid_0) = \mathbf{Some}(pid_0, bid_0, bsize_0, 0, 0) \wedge \\ &\wedge \\ &\text{wfBinstrs}((pid_0, BP, BB, succBP, predB), instrs) \end{aligned}$$

## 6.4 Formalization of the language IL'

This section presents the formalization of an intermediate language IL' which can be seen as a language of CFGB declarations whose operational semantics is defined by means of block-position-wise transitions of the flow of control.

The section consists of two parts and is organized as follows. The first part of the section presents the abstract syntax of the IL' language. The second part presents the semantics definition of IL'.

### 6.4.1 Abstract syntax of IL'

We begin our presentation of the abstract syntax definition by introducing a syntactic set of IL' programs, **Program'**, and metavariables ranging over IL' programs:  $P'$ ,  $S'$ , and  $T'$ .

Definition 6.20 defines formation rule for the syntactic set **Program'**. An IL' program is tuple  $(P, B)$  consisting of an IL program  $P$  and a CFGB declaration  $B$ .

#### Definition 6.20.

$$\mathbf{Program}' \ni P', S', T' ::= (P, B)$$

### 6.4.2 Semantics of the IL' language

This section presents the definition of the semantics of the IL' language. We begin our presentation by introducing sets and metavariables that are associated with the definition.

For the purpose of the operational semantics definition, we introduce the following syntactic sets:

- numbers of transitions from block position to block position made by the flow of control ("small step" numbers) **SS**,
- numbers of transitions from block to block made by the flow of control ("block step" numbers) **BS**,
- numbers of transitions which were made by the flow of control until entering into the current block ("until" numbers) **Until**, and
- augmented configurations **Configuration'**;

and metavariables ranging over these sets:

- $ss$  ranges over small step numbers **SS**,
- $bs$  ranges over numbers of block steps **BS**,
- $until$  and  $ss$  range over until numbers **Until**
- $\sigma'$  ranges over augmented configurations **Configuration'**.

Definition 6.21 defines the set of augmented configurations **Configuration'**. An augmented configuration  $\sigma'$  is a tuple  $(ss, bs, until, bpos, bid, \sigma)$  consisting of

- a small step number  $ss$ ,
- a block step number  $bs$ ,
- an until number  $until$ ,
- a block position status  $bposstat$ ,
- a predecessor block identifier  $prebid$ , and
- a configuration  $\sigma$ .

The meaning of the components  $ss$ ,  $bs$ ,  $until$ ,  $bposstat$ ,  $predbid$ , and  $\sigma$  will be explained below in the part of the section which presents the definition of the operational semantics of IL'.

**Definition 6.21.**

$ \begin{aligned} ss &\in \mathbf{SS} = \mathbf{Nat} \\ bs &\in \mathbf{BS} = \mathbf{Nat} \\ until &\in \mathbf{Until} = \mathbf{Nat} \\ \mathbf{BlckPosStat} \ni bposstat &::= \mathbf{EXITBLCK} \mid \\ &\quad \mathbf{EXCBLCK} \mid \\ &\quad \mathbf{REStBLCK}(bsize, bidx) \mid \\ &\quad \mathbf{NORMBLCK}(bpos) \\ \mathbf{Configuration}' \ni \sigma' &::= (ss, bs, until, bposstat, predbid, \sigma) \end{aligned} $
--

Definition 6.22 defines a transition function  $\text{exec}'$

$$\text{exec}' : \mathbf{Program}' \times \mathbf{Configuration}' \rightarrow \mathbf{Configuration}'$$

which computes a successor configuration  $\text{exec}'(P', \sigma')$  for an IL' program  $P'$  and an augmented configuration  $\sigma'$ .  $\text{exec}'$  is a wrapper function which takes into account that an IL' program is a tuple of the form  $(P, B)$  which consists of an IL program  $P$  and a CFGB declaration  $B$ ; and that the  $\sigma$  component in an augmented configuration  $\sigma' = (ss, bs, until, bposstat, predbid, \sigma)$  is an element of the **Configuration** set which is used by the definition of the operational semantics of the IL language. Independently of the status of program execution, the successor of an augmented configuration  $\sigma' = (ss, bs, until, bposstat, predbid, \sigma)$  has always the same form

$$\text{exec}'((P, B), (ss, bs, until, bposstat, predbid, \sigma)) = (\text{Suc } ss, bs', until', bposstat', predbid', \text{exec}(P, \sigma))$$

where the values of the components  $bs'$ ,  $until'$ ,  $bposstat'$ ,  $predbid'$  are functions of  $\sigma'$  and the successor configuration  $\text{exec}(P, \sigma)$ . In the following, we explain the meaning of the components of  $\sigma'$  and how their values are computed by the function  $\text{exec}'$  during execution of  $(P, B)$ .

*The component  $\sigma$ :* The most important component in  $\sigma'$  is the configuration  $\sigma \in \mathbf{Configuration}$ . The components of  $\sigma$  are interpreted in the same way as explained in Section 4.2.

*The components  $ss$ ,  $bs$ , and  $until$ :* The components  $ss$ ,  $bs$ , and  $until$  serve as step counters and are defined as follows.

- The small step number  $ss$  is incremented by each application of the transition function  $\text{exec}'$ . Thus, the value of the  $ss$  component in an augmented configuration  $\sigma'$  is equal  $n$  iff  $\sigma'$  is the result of  $n$  successive applications of  $\text{exec}'$  to an initial augmented configuration for the program  $(P, B)$ . Later on, we present a function  $\text{init}'$ , which computes an initial augmented configuration for an IL' program.
- The block step number  $bs$  is incremented whenever the flow of control leaves the current block and transfers to another block. Thus, the value of the  $bs$  component in an augmented configuration  $\sigma'$  is equal  $n$  iff the flow of control transferred through  $n$  blocks during partial execution of  $(P, B)$  that produced  $\sigma'$ .
- Given that the flow of control is currently at one of the block positions included in a block, then the value of the component  $until$  indicates how many transitions were made by the flow of control before transferring to the first block position of that block. During the whole execution of the program  $(P, B)$ , the following invariant holds:

$$until \leq ss < (until + bsize) \quad \wedge \quad ss = until + bidx,$$

where  $until$  and  $ss$  denote the values of the components  $until$  and  $ss$ , respectively, and  $bsize$  denotes the length of a block which the flow of control is currently transferring through.

*The component  $predbid$ :* Given that the flow of control is currently at one of the block positions included in a block, the value of the component  $predbid$  indicates that the flow of control was transferred to that block from the block  $predbid$ .

*The component  $bposstat$ :* In an augmented configuration  $\sigma' = (ss, bs, until, bposstat, predbid, (tf, af, pc, b, s))$ , the block status component  $bposstat$  describes a block position in the CFG, which the flow of control is currently at, and its value depends of one of four modes of execution of the program:

The normal mode: A program is executed in the normal mode iff the values of the components  $tf$  and  $af$  in  $\sigma'$  are equal NT and  $AB_{ok}$ , respectively, and the value of the component  $bposstat$  has the form  $NORMBLCK(pid, bid, bsize, bidx, pc)$ . The meaning of the value is that the flow of control is currently at the program point  $pc$  which is allocated to a block position  $pid$  which is included in a block  $bid$  and that the includes relationship between  $pid$  and  $bid$  is described by a block position descriptor  $(pid, bid, bsize, bidx, pc) \in \mathbf{BlkPosDescr}$  such that  $BP = \mathbf{Some}(pid, bid, bsize, bidx, pc)$ . The value of  $predbid$  denotes a block  $predbid$  which is a predecessor of  $bid$ , i.e. that the flow of control, which is currently being in the block  $bid$ , transferred to  $bid$  from  $predbid$ . There is only one possibility for how computing of the successor augmented configuration  $\sigma'$  can result in switching of the execution mode into the normal mode: The program is already executed in the normal mode, i.e.  $\sigma'$  has the form

$$(ss, bs, until, NORMBLCK(pid, bid, bsize, bidx, pc), predbid, (NT, AB_{ok}, pc, b, s)),$$

and the values of the components  $tf$  and  $af$  in the successor configuration  $\mathit{exec}(P, (NT, AB_{ok}, pc, b, s))$  remain unchanged. In this case, the transition function  $\mathit{exec}'$  computes a new value of the component  $bposstat$  and adjusts the values of remaining components according to the result of computation:

If the block position  $pid$  is not the last block position included in the block  $bid$ , i.e. it holds  $\mathit{Suc}(bidx) < bsize$ , then the flow of control merely transfers to a next block position  $pid'$  included in the block  $bid$  with the block index  $bidx'$  equal  $\mathit{Suc}(bidx)$ . This means that the value of the components  $bposstat$ ,  $predbid$ ,  $ss$ ,  $bs$ , and  $until$  are computed as follows.

*$bposstat$ :* The function looks up for the successor of the block position  $pid$ ,  $pid'$ , and its descriptor  $(pid', bid', bsize', bidx', pc')$  and sets the value of the block position status component  $bposstat$  to the value  $NORMBLCK(pid', bid', bsize', bidx', pc')$ .

*$predbid$ :* The value of the predecessor block component remains unchanged as the flow of control does not leave the block  $bid$ .

*$ss$ ,  $bs$ , and  $until$ :* Only the value of the small step number component is incremented as the flow of control does not leave the block  $bid$ .

If the block position  $pid$  is the last block position included in the block  $bid$ , i.e. it holds  $\mathit{Suc}(bidx) = bsize$ , then the flow of control transfers to a next block position  $pid'$  included in a successor block  $bid'$  with the block index  $bidx'$  equal 0. This means that the value of the components  $bposstat$ ,  $predbid$ ,  $ss$ ,  $bs$ , and  $until$  are computed in the following way:

*$bposstat$ :* The function looks up for the successor of the block position  $pid$ ,  $pid'$ , and its descriptor  $(pid', bid', bsize', bidx', pc')$  and sets the value of the block position status component  $bposstat$  to the value  $NORMBLCK(pid', bid', bsize', bidx', pc')$ .

*$predbid$ :* The value of the predecessor block component is set to  $bid$  as the flow of control leaves the block  $bid$  transfers into the successor block  $bid'$ .

*$ss$ ,  $bs$ , and  $until$ :* The small step number and block step number components are incremented to indicate that the flow of control transferred to both the next block position and the next block. The value of the component  $until$  is adjusted according to the length of the block  $bid$

in order to maintain the aforementioned invariant: If the CFGB declaration  $B$  is well-formed, then it holds

$$bidx' = 0 \wedge 0 < bsize \wedge 0 < bsize'.$$

This implies

$$\begin{aligned} & until \leq ss < until + bsize \wedge ss = until + bidx \\ \implies & (until + bsize) \leq \text{Suc}(ss) < (until + bsize) + bsize' \wedge \text{Suc}(ss) = (until + bsize) + bidx'. \end{aligned}$$

The exit mode: In this mode, the values of the components  $bposstat$  and  $pc$  in  $\sigma'$  denote that the last instruction which was executed during partial execution that produced  $\sigma'$  was an exit instruction and that this instruction is the  $pc$ -th instruction of the program. Further, the value `EXITBLCK` denotes that the flow of control is currently in a virtual block which has the length equal 1. There are two possibilities for how computing of the successor augmented configuration can result in switching of the mode of execution into the exit mode:

The first possibility is that the program is already executed in the exit mode: As aforementioned in Section 4.2, if the value of the termination flag component  $tf$  in a configuration  $\sigma$  is equal `T`, then value of the successor configuration  $\text{exec}(P, \sigma)$  is equal  $\sigma$ . In other words, the transition function  $\text{exec}$  "freezes" the values of other components, if it holds  $tf = \text{T}$ . For this reason, the transition function  $\text{exec}'$ , which is a wrapper function, also freezes the values  $bposstat$  and  $predbid$  in the successor configuration of  $(ss, bs, until, \text{EXITBLCK}, predbid, \sigma)$  and computes new values of the step number components  $ss$ ,  $bs$ , and  $until$  according to the length of the virtual block.

The second possibility is that the program is executed in the normal mode, i.e. the values of the components  $t$  and  $af$  are equal `NT` and `ABok`, respectively, the flow of control is at a block position  $pid$  described by a block position descriptor  $(pid, bid, bsize, bidx, pc)$ , the  $pc$ -th instruction of the program is an exit instruction, and  $pid$  is the last block position included in the block  $bid$ , i.e. it holds  $\text{Suc}(bidx) = bsize$ . In this case, it follows from the definition of the operational semantics of the IL language that executing the exit instruction results in setting the values of the components  $tf$  and  $af$  in the successor configuration of the  $\sigma$  to `T` and `ABok`, respectively. The function  $\text{exec}'$  interpretes these values as making transition by the flow of control transfers from the block  $bid$  to the virtual exit block and sets the values of the components  $bposstat$  and  $predbid$  to `EXITBLCK` and  $bid$ , respectively. The new values of the components  $ss$ ,  $bs$ , and  $until$  are computed according to the length of the block  $bid$ .

The emulation mode: A program is executed in the emulation mode iff the values of the components  $tf$  and  $af$  are equal `NT` and `AB`, respectively, and the value of  $bposstat$  has the form `RESTBLCK`( $bid, bsize, bidx$ ). In this mode, the values of the components  $bposstat$  and  $pc$  in  $\sigma'$  denote that the  $pc$ -th instruction was the last instruction that was executed by the partial execution that produced  $\sigma'$  and that execution of that instruction resulted in raising an array-index-out-bounds exception. Further, the values denote that there exists a block position  $pid_{pc}$  and a block index  $bidx_{pc}$  such that  $pid_{pc}$  is included in  $bid$  with the block index  $bidx_{pc}$  and that  $pid_{pc}$  is not the last block position in  $bid$ , i.e. it holds  $\text{Suc}(bidx_{pc}) < bsize$ , where  $bsize$  is the length of  $bid$ . Additionally, the value `RESTBLCK`( $bid, bsize, bidx$ ) denotes that the flow of control is currently at one of virtual block positions that are included in a virtual rest block which is defined for the block  $bid$ , its length  $bsize$ , and one of its block indexes  $bidx_{pc}$ . The set of virtual block positions which are included in this block is defined as follows.

$$\{(bid, bsize, bidx) \mid bidx_{pc} < bidx \wedge bidx < bsize\}$$

There are two possibilities for how computing of the successor augmented configuration can result in switching of the mode of execution into the emulation mode:

The first possibility is that the program is already executed in this mode: If the flow of control is currently at a virtual block position  $(bid, bsize, bidx)$  which is not the last one in the virtual block  $bid$ , i.e. if  $\text{Suc}(bidx < bsize)$  holds, then the flow of control transfers to the next virtual block position  $(bid, bsize, \text{Suc } bidx)$ , the values of the step number components are updated correspondingly and execution proceeds in the emulation mode since it follows from the definition of the operational semantics of the IL language that the values of the components  $tf$  and  $af$  in the successor configuration  $\text{exec}(P, \sigma)$  have the same values as in  $\sigma$ . The second possibility is that the program is executed in the normal mode, the flow of control is at a block position described by the descriptor  $(pid_{pc}, bid, bsize, bidx_{pc}, pc)$ , the block position  $pid_{pc}$  is not the last block position in the block  $bid$ , and execution of the  $pc$ -th instruction raises an array-index-of-bounds exception. In this case, it follows from the definition of the operational semantics of the IL language that the values of the components  $tf$  and  $af$  are equal NT and AB, respectively. Thus, the step number components are updated correspondingly, the flow of control transfers to a virtual block position  $(bid, bsize, \text{Suc } bidx_{pc})$ , which is the first block position of a virtual rest block defined for the values  $bid$ ,  $bsize$ , and  $bidx_{pc}$ . The resulting successor configuration of  $\sigma'$  has the form

$$(\text{Suc } ss, bs, \text{until}, \text{RESTBLCK}(bsize, \text{Suc } bidx_{pc}), \text{predbid}, \text{exec}(P, \sigma))$$

which means that execution switched to the emulation mode. Emulation of execution of the rest of a block  $bid$  means that the next  $bsize - (bidx + 1)$  successive applications of the transition function to this configuration results in  $bsize - (bidx + 1)$  successive increments of the values  $\text{Suc}(ss)$  and  $\text{Suc}(bidx_{pc})$  only. The rest of the components will remain unchanged.

The exception mode: A program is executed in the exception mode iff the values of the components  $tf$ ,  $af$ , and  $bposstat$  in  $\sigma'$  are equal NT, AB, and EXCBLCK, respectively. In this mode, the values of the components  $bposstat$  and  $pc$  in  $\sigma'$  denote that the  $pc$ -th instruction was the last instruction that was executed by the partial execution that produced  $\sigma'$  and that execution of that instruction resulted in raising an array-index-out-of-bounds exception. Further, the value EXCBLCK denotes that the flow of control is currently at a virtual block position of a virtual exception block which has the length equal 1, i.e. it includes only that block position. There are two possibilities for how computing of the successor augmented configuration can result in switching of the mode of execution into the exception mode:

The first possibility is that the program is already executed in this mode: If the flow of control is currently at the virtual block position in the virtual exception block, then the flow of control remains at that block position after making transition to the successor configuration of  $\sigma'$ , the values of the step number components are updated according to the length of the virtual block, and execution proceeds in the exception mode since it follows from the definition of the operational semantics of the IL language that the values of the components  $tf$  and  $af$  in the successor configuration  $\text{exec}(P, \sigma)$  have the same values as in  $\sigma$ .

The second possibility is that the program is executed in the emulation mode and that the flow of control is currently at the last virtual position of a virtual rest block defined for a block  $bid$  that has the length  $bsize$ , i.e. the values of the components  $tf$  and  $af$  are equal NT and AB, respectively, and the value of the  $bposstat$  component has the form  $\text{RESTBLCK}(bid, bsize, bidx)$  with  $\text{Suc}(bidx) = bsize$ . The function  $\text{exec}'$  interprets the values of the components in  $\sigma'$  as if there were a virtual edge connecting the virtual block position  $(bid, bsize, bidx)$  and the virtual block position in the virtual exception block and the flow of control transfers along that edge into the virtual exception block. Transition of the flow into the exception block is realised as setting the values of the components  $bposstat$  and  $predbid$  to EXCBLCK and  $bid$ , respectively, updating the values of the step number components according to the length of the block  $Vbid$  (not the length of the virtual rest block!), and switching of the mode of execution into the exception mode as it follows from the definition of the operational semantics of the IL language

that the values of the components  $tf$  and  $af$  in the successor configuration  $\text{exec}(P, \sigma)$  are equal NT and AB respectively, and thus the same as corresponding values of the components in  $\sigma$ .

The fact that the values of step number components are updated according to the length of the block  $Vbid$  and not the length of the virtual rest block is an important part of the definition of the operational semantics of the IL': Whenever the flow of control transfers from the last block position of a virtual rest block defined for a block  $bid$ , which has the length  $bsize$ , into the virtual exception block, the values of the step number components denote that the flow of control made  $bsize$  transitions before leaving  $bid$ .

**Definition 6.22.**

$$\begin{array}{l}
\text{exec}' : \text{Program}' \times \text{Configuration}' \rightarrow \text{Configuration}' \\
\text{exec}'((P, B), (ss, bs, until, \text{EXITBLCK}, \text{predbid}, \sigma)) = \\
\quad (\text{Suc } ss, \text{Suc } bs, \text{Suc } until, \text{EXITBLCK}, \text{predbid}, \text{exec}(P, \sigma)) \\
\text{exec}'((P, B), (ss, bs, until, \text{EXCBLCK}, \text{predbid}, \sigma)) = \\
\quad (\text{Suc } ss, \text{Suc } bs, \text{Suc } until, \text{EXCBLCK}, \text{predbid}, \text{exec}(P, \sigma)) \\
\text{exec}'((P, B), (ss, bs, until, \text{RESTBLCK}(bsize, bidx), \text{predbid}, \sigma)) = \\
\quad \begin{cases} (\text{Suc } ss, bs, until, \text{RESTBLCK}(bsize, \text{Suc } bidx), \text{predbid}, \text{exec}(P, \sigma)) & \text{if } \text{Suc}(bidx) < bsize, \\ (\text{Suc } ss, \text{Suc } bs, until + bsize, \text{EXCBLCK}, \text{predbid}, \text{exec}(P, \sigma)) & \text{otherwise} \end{cases} \\
\text{exec}'((P, B), (ss, bs, until, \text{NORMBLCK}(pid, bid, bsize, bidx, pc), \text{predbid}, \sigma)) = \\
\quad \text{let} \\
\quad \quad (tf', af', pc', b', s') = \text{exec}(P, \sigma); \\
\quad \quad \text{Some}(pid') = \text{succBP}(pid, pc'); \\
\quad \quad \text{Some}(pid', bid', bsize', bidx', pc') = \text{BP}(pid') \\
\quad \text{in} \\
\quad \quad \begin{cases} (\text{Suc } ss, bs, until, \text{NORMBLCK}(pid', bid', bsize', bidx', pc'), \text{predbid}, \text{exec}(P, \sigma)) & \text{if } \text{Suc}(bidx) < bsize \wedge af' = \text{AB}_{\text{OK}}, \\ (\text{Suc } ss, bs, until, \text{RESTBLCK}(bsize, \text{Suc } bidx), \text{predbid}, \text{exec}(P, \sigma)) & \text{if } \text{Suc}(bidx) < bsize \wedge af' = \text{AB}, \\ (\text{Suc } ss, \text{Suc } bs, until + bsize, \text{EXIT}, bid, \text{exec}(P, \sigma)) & \text{if } \text{Suc}(bidx) = bsize \wedge af' = \text{AB}_{\text{OK}} \wedge tf' = \text{T}, \\ (\text{Suc } ss, \text{Suc } bs, until + bsize, \text{NORMBLCK}(pid', bid', bsize', bidx', pc'), bid, \text{exec}(P, \sigma)) & \text{if } \text{Suc}(bidx) = bsize \wedge af' = \text{AB}_{\text{OK}} \wedge tf' = \text{NT}, \\ (\text{Suc } ss, \text{Suc } bs, until + bsize, \text{EXCBLCK}, bid, \text{exec}(P, \sigma)) & \text{otherwise} \end{cases} \\
\quad \text{end}
\end{array}$$

Definition 6.23 defines a function  $\text{init}'$  which computes an initial augmented configuration for an IL' program  $P' = (P, B)$  where the CFGB declaration has the form  $(pid_0, BP, BB, \text{succBP}, \text{predB})$ . Given that the CFGB declaration  $B$  is well-formed, then the values of the components of an initial configuration  $\sigma'_0 = \text{init}'(P, (pid_0, BP, BB, \text{succBP}, \text{predB}))$  are defined as follows.

- The values of all three step number components  $ss$ ,  $bs$ , and  $until$  are set to 0,
- The block position status  $bposstat$  is set the value  $\text{NORMBLCK}(pid_0, bid_0, bsize_0, bidx_0, pc_0)$  with  $\text{Some}(pid_0, bid_0, bsize_0, bidx_0, pc_0) = \text{BP}(pid_0)$ . The value of  $bposstat$  denotes that the flow of control is at the first block position of the entry block  $bid_0$ .
- The value of the predecessor block component  $\text{predbid}$  is set to  $bid_0$  and it denotes that in our setting the predecessor block of the entry block is predefined as entry block itself.
- The value of the  $\sigma$  component in an initial augmented configuration for an IL' program  $(P, B)$  is defined as initial configuration for an IL program  $P$ ,  $\text{init}(P)$ .

**Definition 6.23.**

$$\begin{array}{l}
\mathit{init}' : \mathbf{Program}' \rightarrow \mathbf{Configuration}' \\
\mathit{init}'(P, B) = \\
\quad \mathbf{let} \\
\quad \quad (pid_0, BP, BB, succBP, predB) = B \\
\quad \mathbf{in} \\
\quad \quad \left\{ \begin{array}{ll}
(0, 0, 0, \mathbf{NORMBLCK}(pid_0, bid_0, bsize_0, bidx_0, pc_0), bid_0, \mathit{init}(P)) & \mathbf{if} \ \mathbf{Some}(pid_0, bid_0, bsize_0, bidx_0, pc_0) = BP(pid_0), \\
\mathbf{arbitrary} & \mathbf{otherwise}
\end{array} \right. \\
\quad \mathbf{end}
\end{array}$$

Definition 6.24 defines a function  $M'$  which models a simple machine performing partial executions of IL' programs. The function  $M'$  takes an IL' program  $P'$ , an augmented configuration  $\sigma'$ , and a small step number  $ss$ ; and computes an augmented configuration,  $M'(P', \sigma', ss)$ , which is the result of  $ss$  successive applications of the transition function  $\mathit{exec}'$  to the configuration  $\sigma'$ . We use this function to formalize results of partial executions of IL' programs, e.g.  $M'(P', \mathit{init}'(P'), n)$ .

**Definition 6.24.**

$$\begin{array}{l}
M' : \mathbf{Program}' \times \mathbf{Configuration}' \times \mathbf{Nat} \rightarrow \mathbf{Configuration}' \\
M'(P', \sigma', 0) = \sigma' \\
M'(P', \sigma', \mathbf{Suc}(n)) = \mathit{exec}'(P', M'(P', \sigma', n))
\end{array}$$

For the definition of the program semantics of the IL' language, we need to formalize the notions of emission of an augmented configuration  $\sigma' \in \mathbf{Configuration}'$  and observable behavior of an IL' program.

Definition 6.25 defines a function  $\mathit{emission}'$  which computes emission of an augmented configuration  $\sigma'$ .  $\mathit{emission}'$  is a wrapper function which computes emission of an augmented configuration  $\sigma' = (ss, bs, \mathit{until}, \mathit{bposstat}, \mathit{predbid}, \sigma)$  as emission of its  $\sigma$  component,  $\mathit{emission}(\sigma)$ .

**Definition 6.25.**

$$\begin{array}{l}
\mathit{emission}' : \mathbf{Configuration}' \rightarrow \mathbf{Emission} \\
\mathit{emission}'(ss, bs, \mathit{until}, \mathit{bposstat}, \mathit{predbid}, \sigma) = \mathit{emission}(\sigma)
\end{array}$$

Definition 6.26 defines a function  $\mathit{Sem}'$  which computes observable behavior of an IL' program during its execution by the machine  $M'$ . An informal idea of the algorithm computing observable behavior of an IL' programs  $P'$  as follows. We let the machine  $M'$  execute the program  $P'$  forever by applying the transition function  $\mathit{exec}'$  successively in an infinite loop. In doing so, we generate an infinite trace of augmented configurations  $t$  where each element of the trace  $t(n)$  is the result of  $n$  applications of the transition function  $\mathit{exec}$  applied successively to initial augmented configuration  $t(n) = M'(P', \mathit{init}'(P'), n)$ . Each trace element  $t(i+1)$  carries information about changes of the observable behavior of  $P'$  during transition from the trace element  $t(i)$  to  $t(i+1)$ . This information is encoded in the state of the output buffer  $b$  in the  $\sigma$  component in  $t(i+1)$ . The algorithm builds an output sequence of tokens from the trace  $t$  in three steps:

1. First step inspects iteratively all elements of  $t$  in the order of their producing,  $t(0)$ ,  $t(1)$ , ..., and computes an intermediate sequence of emissions of  $P'$  by applying the function  $\mathit{emission}'$  to the elements of  $t$ .

2. Second step filters out NOEMIT emissions from the intermediate sequence.
3. Third step builds output sequence of tokens by stripping EMIT constructors from elements of the sequence which results from the second step.

**Definition 6.26.**

$\text{Sem}' : \text{Program}' \rightarrow \text{ObservableBehavior}$   
 $\text{Sem}'(P') = \{tok \mid \exists n. \text{emission}'(M'(P', n)) = \text{EMIT}(tok)\}$

The semantics of an IL' program  $P'$ ,  $\text{Sem}'(P')$ , is defined as observable behavior of  $P'$  during execution of  $P'$  by the machine  $M'$ .

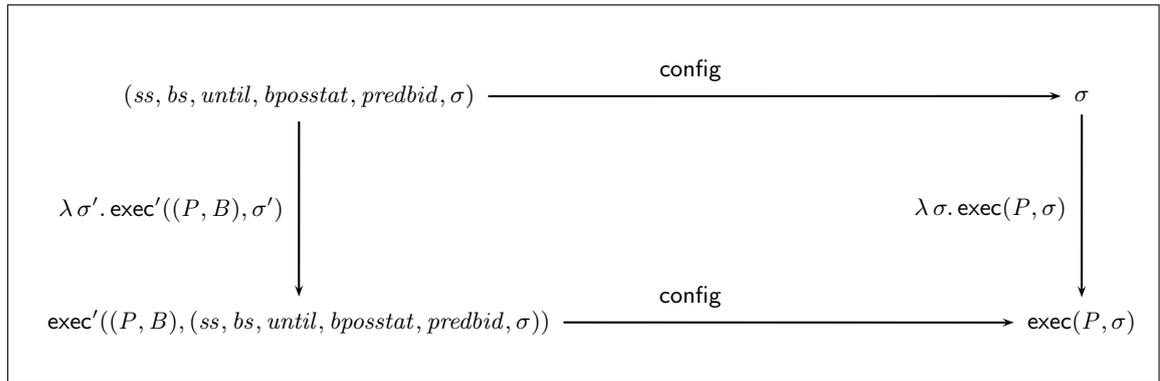
**6.5 Equality of the semantics of the languages IL and IL'**

Definition 6.27 defines a projection function  $\text{config}$  which takes an augmented configuration  $\sigma' \in \text{Configuration}'$  and projects to its configuration component  $\sigma \in \text{Configuration}$ .

**Definition 6.27.**

$\text{config} : \text{Configuration}' \rightarrow \text{Configuration}$   
 $\text{config}(ss, bs, \text{until}, \text{bposstat}, \text{prebid}, \sigma) = \sigma$

Below, we present Lemma 6.28 which states for an IL' program  $(P, B)$  and its corresponding IL program  $P$  that a transition function  $\lambda \sigma'. \text{exec}'((P, B), \sigma')$  maps the  $\sigma$  component in its argument, which is an augmented configuration  $\sigma' \in \text{Configuration}'$ , to the same value as a corresponding transition function  $\lambda \sigma. \text{exec}(P, \sigma)$ . Informally, the lemma states that  $\text{exec}'$  is a proper wrapper function w.r.t. the function  $\text{exec}$  and that the diagram in Figure 6.7 commutes.

**Fig. 6.7.**

**Lemma 6.28.** (*Commutativity of  $\text{exec}$ ,  $\text{exec}'$ , and  $\text{config}$* )

$$\text{exec}(P, \text{config}(\sigma')) = \text{config}(\text{exec}'((P, B), \sigma'))$$

□

Lemma 6.29 is a statement about two machine functions  $\lambda n. M'((P, B), \text{init}'((P, B), n))$  and  $\lambda n. M(P, \text{init}(P), n)$  defined for an IL' program  $(P, B)$  and its corresponding IL program  $P$ , respectively, which states for all transition numbers  $n$  that a partial execution of the program  $(P, B)$  which makes  $n$  transitions from block position to block position in  $B$  produces an augmented configuration  $\sigma'$  whose  $\sigma$  component is equal to a configuration which is the result of partial execution of  $P$  which makes equal number of transitions from program point to program point in  $P$ . Informally, the lemma states that  $M'$  is a proper wrapper function w.r.t. the function  $M$  and that the diagram in Figure 6.8 commutates.

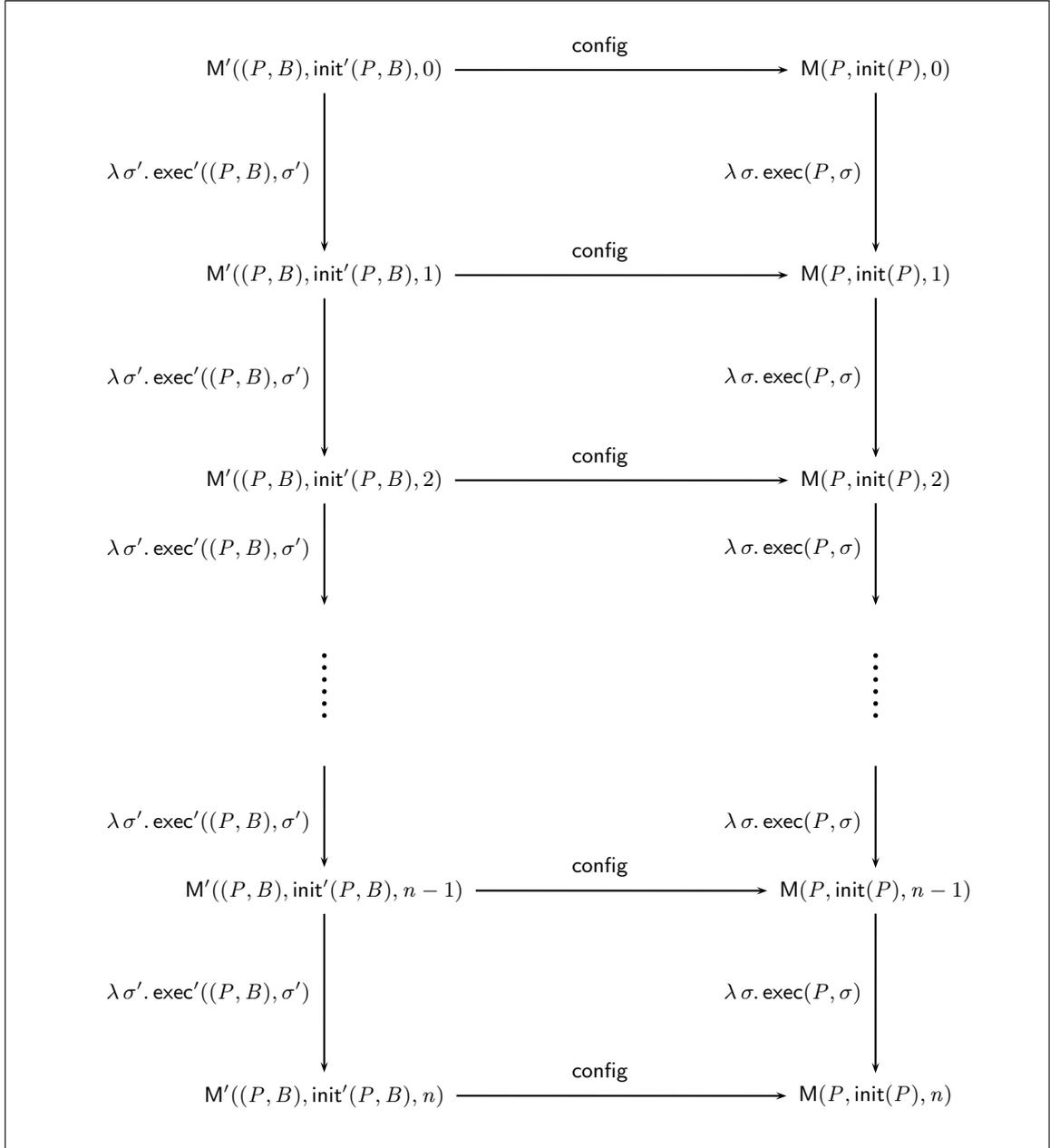


Fig. 6.8.

**Lemma 6.29.** (*Commutativity of M, M', and config*)

$$\text{wfB}(P, B) \implies \text{M}(P, \text{init}(P), n) = \text{config}(\text{M}'((P, B), \text{init}'(P, B), n))$$

*Proof.* The proof of this lemma follows by induction over the number  $n$ .

*The induction begin:*

For the induction begin, one has to show the following:

$$\text{wfB}(P, B) \implies \text{M}(P, \text{init}(P), 0) = \text{config}(\text{M}'((P, B), \text{init}'(P, B), 0))$$

By the definitions of M and M', this can be simplified to

$$\text{wfB}(P, B) \implies \text{init}(P) = \text{config}(\text{init}'(P, B))$$

By the definition of  $B$ , there exist  $\text{pid}_0$ ,  $BP$ ,  $BB$ ,  $\text{succBP}$ , and  $\text{predB}$  such that  $B = (\text{pid}_0, BP, BB, \text{succBP}, \text{predB})$ . Therefore, by application of the existential elimination rule, the following statement has to be shown for arbitrary but fixed  $\text{pid}_0$ ,  $BP$ ,  $BB$ ,  $\text{succBP}$ , and  $\text{predB}$ :

$$\begin{aligned} & \text{wfB}(P, (\text{pid}_0, BP, BB, \text{succBP}, \text{predB})) \\ & \implies \\ & \text{init}(P) = \text{config}(\text{init}'(P, (\text{pid}_0, BP, BB, \text{succBP}, \text{predB}))) \end{aligned}$$

From  $\text{wfB}(P, (\text{pid}_0, BP, BB, \text{succBP}, \text{predB}))$  follows

$$\exists \text{bid}_0 \text{ bsize}_0. BP(\text{pid}_0) = \text{Some}(\text{pid}_0, \text{bid}_0, \text{bsize}_0, 0, 0)$$

by the definition of  $\text{wfB}$ . Therefore, by application of the existential elimination rule, the following statement has to be shown for arbitrary but fixed  $\text{bid}_0$  and  $\text{bsize}_0$ :

$$\begin{aligned} & BP(\text{pid}_0) = \text{Some}(\text{pid}_0, \text{bid}_0, \text{bsize}_0, 0, 0) \\ & \implies \\ & \text{init}(P) = \text{config}(\text{init}'(P, (\text{pid}_0, BP, BB, \text{succBP}, \text{predB}))) \end{aligned}$$

Unfolding the definition of  $\text{init}'$  and simplifying the conclusion and the premise yields:

$$\begin{aligned} & BP(\text{pid}_0) = \text{Some}(\text{pid}_0, \text{bid}_0, \text{bsize}_0, 0, 0) \\ & \implies \\ & \text{init}(P) = \text{config}(0, 0, 0, \text{NORMBLCK}(\text{pid}_0, \text{bid}_0, \text{bsize}_0, 0, 0), \text{bid}_0, \text{init}(P)) \end{aligned}$$

This holds by the definition of  $\text{config}$ .

*The induction step:*

For the induction step, the following statement has to shown:

$$\begin{aligned} & \text{wfB}(P, B) \wedge \text{M}(P, \text{init}(P), n-1) = \text{config}(\text{M}'((P, B), \text{init}'(P, B), n-1)) \\ & \implies \\ & \text{M}(P, \text{init}(P), n) = \text{config}(\text{M}'((P, B), \text{init}'(P, B), n)) \end{aligned}$$

To show this, one has to derive the conclusion from the premise in a forward manner:

$$\begin{aligned}
& M(P, \text{init}(P), n - 1) = \text{config}(M'((P, B), \text{init}'(P, B), n - 1)) \\
& \implies [\text{by HOL rule } x=y \implies f(x)=f(y)] \\
& (\lambda \sigma'. \text{exec}(P, \sigma'))(M(P, \text{init}(P), n - 1)) \\
& = \\
& (\lambda \sigma'. \text{exec}(P, \sigma'))(\text{config}(M'((P, B), \text{init}'(P, B), n - 1))) \\
& \implies \\
& \text{exec}(P, M(P, \text{init}(P), n - 1)) = \text{exec}(P, \text{config}(M'((P, B), \text{init}'(P, B), n - 1))) \\
& \implies [\text{by the definition of } M \text{ and application of Lemma 6.28}] \\
& M(P, \text{init}(P), n) = \text{config}(\text{exec}'((P, B), M'((P, B), \text{init}'(P, B), n - 1))) \\
& \implies [\text{by the definition of } M'] \\
& M(P, \text{init}(P), n) = \text{config}(M'((P, B), \text{init}'(P, B), n))
\end{aligned}$$

□

Theorem 6.30 states that observable behaviors of an IL' program  $(P, B)$  and a corresponding IL program  $P$  are equal.

**Theorem 6.30.** (*Equality of the program semantics of the languages IL and IL'*)

$$\text{wfB}(P, B) \implies \text{Sem}(P) = \text{Sem}'(P, B)$$

□

The proof of Theorem 6.30 is straightforward as its statement is a direct consequence of Lemma 6.29.

## 6.6 Formalization of the language IL''

This section presents the formalization of an intermediate language IL'' which can be seen as a language of control flow graphs with blocks with the operational semantics defined in terms of block-wise transfers of the flow of control. The formalization of the IL'' language is based on the formalization of the IL' language: The IL'' programs have the same abstract syntax as the IL' language programs and the definition of their operational semantics is based on the definition of the operational semantics for the language IL' which is defined in terms of block-position-wise transfers of the flow of control.

The rest of the section is organized as follows. The first part of the section presents the abstract syntax of the IL'' language. The second part presents the semantics definition of IL''.

### 6.6.1 Abstract syntax

We begin our presentation of the abstract syntax of the language IL'' by introducing a syntactic set of IL'' programs **Program''**; and metavariables  $P''$ ,  $S''$ , and  $T''$  ranging over IL'' programs **Program''**.

Definition 6.31 defines a formation rule for the syntactic set **Program''** which defines the abstract syntax of the language IL''. An IL'' program is tuple  $(P, B)$  consisting of an IL program  $P$  and a block environment  $B$ . Thus, the languages IL' and IL'' have the same abstract syntax.

**Definition 6.31.**

$$\mathbf{Program''} \ni P'', S'', T'' ::= P'$$

### 6.6.2 Semantics of the language IL''

For the purpose of the semantics definition of the IL'', we introduce a syntactic set of augmented configurations **Configuration''**, which is equal to the syntactic set of augmented configurations **Configuration'**, and a metavariable  $\sigma''$  ranging over augmented configurations **Configuration''**. Definition 6.32 gives formation rule of the set **Configuration''**.

**Definition 6.32.**

$$\mathbf{Configuration''} \ni \sigma'' ::= \sigma'$$

Definition 6.33 defines two auxiliary functions **ss** and **blksize** for the definition of the operational semantics of IL''. The function **ss** projects an augmented configuration  $\sigma'$  to its small step number component  $ss$ . The function **blksize** takes an augmented configuration  $\sigma'$  as input and computes the length of a block which includes a block position which described by the value of its block position status component  $bposstat$ .

**Definition 6.33.**

$$\begin{aligned} \mathbf{ss} : \mathbf{Configuration''} &\rightarrow \mathbf{SS} \\ \mathbf{ss}(ss, bs, until, bposstat, predpid, \sigma) &= ss \\ \\ \mathbf{blksize} : \mathbf{Configuration''} &\rightarrow \mathbf{BlkSize} \\ \mathbf{blksize}(ss, bs, until, \mathbf{EXITBLCK}, predpid, \sigma) &= 1 \\ \mathbf{blksize}(ss, bs, until, \mathbf{EXCBLCK}, predpid, \sigma) &= 1 \\ \mathbf{blksize}(ss, bs, until, \mathbf{RESTBLCK}(bid, bsize, bidx), predpid, \sigma) &= bsize \\ \mathbf{blksize}(ss, bs, until, \mathbf{NORMBLCK}(pid, bid, bsize, bidx, pc), predpid, \sigma) &= bsize \end{aligned}$$

Definition 6.34 defines a function **init''** which computes an initial configuration  $\sigma_0'' \in \mathbf{Configuration''}$  for an IL'' program  $P''$ . As the sets **Program'** and **Program''**; and **Configuration'** and **Configuration''** are equal, respectively, the initial augmented configurations for IL'' programs are defined in the same way as their counterparts for IL' programs.

**Definition 6.34.**

$$\begin{aligned} \mathbf{init''} : \mathbf{Program''} &\rightarrow \mathbf{Configuration''} \\ \mathbf{init''}(P'') &= \mathbf{init'}(P'') \end{aligned}$$

Definition 6.35 defines a transition function **exec''** which computes a successor configuration  $\mathbf{exec}(P'', \sigma'')$  for an IL'' program  $P''$  and an augmented configuration  $\sigma''$ . The function **exec''**

assumes that  $\sigma''$  is the result of partial execution of  $P''$  and that the flow of control is at the entry block position  $pid$  of a block  $bid$ , i.e. the block index of  $pid$  is equal 0, and computes the successor in three steps as follows.

1. First step computes the number of successive applications of the transition function  $exec'$  which were performed by the partial execution. This number is recorded in the  $ss$  component of  $\sigma''$ .
2. Second step computes the length of the block  $bid$  including  $pid$ ,  $bsize$ .
3. Third step uses the results delivered the first and second steps: According to the value of the small step number  $ss$ ,  $\sigma''$  is the result of  $ss$  successive applications of the transition function  $exec'$  to the initial augmented configuration  $init''(P'')$ ,  $M'(P'', init''(P''), ss)$ . Thus, the result of execution of all instructions at program points allocated to block positions included by  $bid$  must be a configuration that is the result of  $bsize$  successive application of  $exec'$  to  $M'(P'', init''(P''), ss)$ . This configuration is equal  $M'(P'', init''(P''), ss + bsize)$ .

**Definition 6.35.**

```

exec'' : Program'' × Configuration'' → Configuration''
exec''( $P''$ ,  $\sigma''$ ) =
  let
     $ss = ss(\sigma'')$ 
     $bsize = blcksize(\sigma'')$ 
  in
     $M'(P'', init''(P''), ss + bsize)$ 
  end

```

Definition 6.36 defines a function  $M''$  which models a simple machine performing partial executions of IL'' programs. The function  $M''$  takes an IL'' program  $P''$ , an augmented configuration  $\sigma''$ , and a block step number  $bs$ ; and computes an augmented configuration,  $M''(P'', \sigma'', bs)$ , which is the result of  $bs$  successive applications of the transition function  $exec''$  to the configuration  $\sigma''$ . We use this function to formalize results of partial executions of IL'' programs, e.g.  $M''(P'', init''(P''), n)$ .

**Definition 6.36.**

```

M'' : Program'' × Configuration'' × Nat → Configuration''
M''( $P''$ ,  $\sigma''$ , 0) =  $\sigma''$ 
M''( $P''$ ,  $\sigma''$ ,  $Suc(n)$ ) =  $exec''(P'', M''(P'', \sigma'', n))$ 

```

Definition 6.37 defines a function  $emission''$  which formalizes the notion of emission sets for the IL'' language. A set of emissions of an IL'' program  $P''$  is build w.r.t. two points of partial execution of  $P''$  by the machine  $M'$ ,  $ss_i$  and  $ss_j$  with  $ss_i \leq ss_j$ , and comprises all changes of observable behavior which happen after producing the configuration  $M'(P'', init''(P''), ss_i)$  and before producing the configuration  $M'(P'', init''(P''), ss_j)$  by the machine  $M'$ . We use this function to compute the set of changes of observable behavior of an IL'' program while the flow of control is making transition from a block position which is the entry of a block to a block position which is the entry of a successor block of that block.

**Definition 6.37.**

$$\begin{aligned} \text{emission}'' : \mathbf{Program}'' \times \mathbf{Configuration}'' \times \mathbf{Configuration}'' &\rightarrow \mathbf{ObservableBehavior} \\ \text{emission}''(P'', \sigma_i'', \sigma_j'') &= \{tok \mid \exists i. \text{ss}(\sigma_i'') \leq i \wedge i < \text{ss}(\sigma_j'') \wedge \\ &\quad \text{emission}'(M'(P'', \text{init}'(P'')), i) = \text{Some}(tok)\} \end{aligned}$$

Definition 6.38 defines a function  $\text{Sem}''$  which computes observable behavior of an IL'' program during its execution by the machine  $M''$ . An informal idea of the algorithm computing observable behavior of an IL'' program  $P''$  is as follows. We compute the observable behavior of  $P''$  in three steps. First, we let the machine  $M''$  execute the program  $P''$  forever by applying the transition function  $\text{exec}''$  successively in an infinite loop. In doing so, we generate an infinite trace of augmented configurations  $t$  where each element of the trace is the result of  $n$  successive transitions starting from initial augmented configuration  $\text{init}''(P'')$ ,  $t(n) = M''(P'', \text{init}''(P''), n)$ . Second, for each pair of successive trace elements  $(t(i), t(i+1))$ , we compute the set of tokens emitted by  $P''$  during making transition from  $t(i)$  to  $t(i+1)$ ,  $\text{emission}''(P'', t(i), t(i+1))$ . The second step yields a (possibly infinite) sequence of sets of tokens. The third step returns the result as the union of all sets in the sequence.

**Definition 6.38.**

$$\begin{aligned} \text{Sem}'' : \mathbf{Program}'' &\rightarrow \mathbf{ObservableBehavior} \\ \text{Sem}''(P'') &= \\ \text{let} & \\ \quad \sigma_0'' &= \text{init}''(P'') \\ \text{in} & \\ \quad \{tok \mid \exists i. tok \in \text{emission}''(P'', M''(P'', \sigma_0'', i), \text{exec}''(P'', M''(P'', \sigma_0'', i)))\} & \\ \text{end} & \end{aligned}$$

Theorem 6.39 states that if an IL program  $P$  is well-typed w.r.t. to a program type  $\Phi$  and a CFGB declaration  $B$  is well-formed w.r.t.  $P$ , then observable behaviors of an IL' program  $(P, B)$  and an IL'' program  $(P, B)$  are equal. Informally, the theorem states that if the CFGB declaration  $B$  is well-formed w.r.t. the program  $P$ , then it does not make any difference to the resulting observable behavior of the tuple  $(P, B)$  whether it is interpreted as an IL' program and executed by the machine  $M'$ , which transfers the flow of control from block position to block position, or it is interpreted as an IL'' program and executed by the machine  $M''$ , which transfers the flow of control from block entry to block entry.

**Theorem 6.39.** (*Equality of the semantics of IL' and IL''*)

$$\text{wtp}(P, \Phi) \wedge \text{wfB}(P, B) \implies \text{Sem}'(P, B) = \text{Sem}''(P, B)$$

□

## 6.7 Bisimulation predicate on pairs of IL'' program executions

This section presents formalizations of the notions of bisimulation relation and bisimulation predicate.

We begin the presentation of our formalization by introducing a set of bisimulation relations **BisimulationRelation** and a metavariable  $\mathcal{R}$  ranging over bisimulation relations.

Definition 6.40 gives the formation rule for the set of bisimulation relations **BisimulationRelation**. A bisimulation relation between the sets of augmented configurations **Configuration''** is a subset of **Configuration''**  $\times$  **Configuration''**.

**Definition 6.40.**

$$\mathcal{R} \in \mathbf{BisimulationRelation} = \mathcal{P}(\mathbf{Configuration''} \times \mathbf{Configuration''})$$

Definition 6.41 defines a bisimulation predicate **bisimulation** on executions of two IL" programs w.r.t. a bisimulation relation. Executions of two IL" programs  $S''$  and  $T''$  bisimulate w.r.t. a bisimulation relation  $\mathcal{R}$  iff

1. a pair  $(\mathit{init''}(S''), \mathit{init''}(T''))$  consisting of initial augmented configurations for  $S''$  and  $T''$  is in the relation  $\mathcal{R}$ ; and
2.  $\mathcal{R}$  is closed under the operator

$$\lambda(\sigma_S'', \sigma_T''). (\mathit{exec''}(S'', \sigma_S''), \mathit{exec''}(T'', \sigma_T''));$$

and

3. it holds for each configuration pair  $(\sigma_S'', \sigma_T'')$  in  $\mathcal{R}$  that the sets of changes of observable behaviors of  $S''$  and  $T''$  resulting from applying the above operator to that pair are equal.

**Definition 6.41.**

$$\mathbf{bisimulation} : \mathbf{Program''} \times \mathbf{Program''} \times \mathbf{BisimulationRelation} \rightarrow \mathbf{Bool}$$

$$\mathbf{bisimulation}(S'', T'', \mathcal{R}) =$$

$$(\mathit{init''}(S''), \mathit{init''}(T'')) \in \mathcal{R} \wedge$$

$$\forall (\sigma_S'', \sigma_T'') \in \mathcal{R}. (\mathit{exec''}(S'', \sigma_S''), \mathit{exec''}(T'', \sigma_T'')) \in \mathcal{R} \wedge$$

$$\mathit{emission''}(S'', \sigma_S'', \mathit{exec''}(S'', \sigma_S'')) = \mathit{emission''}(T'', \sigma_T'', \mathit{exec''}(T'', \sigma_T''))$$

We call the function **bisimulation** a bisimulation predicate on executions of two IL" programs as the right hand side of its definition implies the statement

$$\forall n. (\mathit{M''}(S'', n), \mathit{M''}(T'', n)) \in \mathcal{R} \wedge$$

$$\mathit{emission''}(S'', \mathit{init''}(S''), \mathit{M''}(S'', n)) = \mathit{emission''}(T'', \mathit{init''}(T''), \mathit{M''}(T'', n))$$

which says for all partial executions of  $S''$  and  $T''$  that if they make the same number of transitions, then observable behaviors of  $S''$  and  $T''$  during those executions are equal and they produce two augmented configurations  $\sigma_S''$  and  $\sigma_T''$  with  $(\sigma_S'', \sigma_T'') \in \mathcal{R}$ .

Theorem 6.42 says that if two partial executions of two IL" programs  $S''$  and  $T''$  bisimulate w.r.t. a bisimulation relation  $\mathcal{R}$ , then observable behaviors of  $S''$  and  $T''$  are equal.

**Theorem 6.42.** (*Bisimulation of executions of two IL" programs implies equality of their observable behaviors*)

$$\mathbf{bisimulation}(S'', T'', \mathcal{R}) \implies \mathbf{Sem''}(S'') = \mathbf{Sem''}(T'')$$

□

## 6.8 Optimization independent translation correctness criterion

In the beginning of Chapter 6, we mentioned that the main purpose of Layer 4 in our implementation of the SVF is to provide a formal definition of an optimization independent translation correctness criterion on two IL programs which expresses a sufficient condition of the translation correctness predicate provided by the translation contract.

In Sections 5.2, 6.3, and 6.7, we formalized the well-typedness predicate `wtp` on IL programs and program types, the well-formedness predicate `wfB` on CFGB declarations and IL programs, and the bisimulation predicate `bisimulation` on IL<sup>''</sup> programs and bisimulation relations, respectively. Now, with the definitions of those predicates at hand, we can give the definition of an optimization independent translation correctness criterion on two IL programs and a bisimulation relation which is independent of any optimizations performed by our compiler.

### Definition 6.43.

$$\begin{aligned} \text{TCC} &: \text{Program} \times \text{Program} \times \text{BisimulationRelation} \rightarrow \text{Bool} \\ \text{TCC}(S, T, \mathcal{R}) &= \exists \Phi_S \Phi_T B_S B_T. \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \\ &\quad \text{bisimulation}((S, B_S), (T, B_T), \mathcal{R}) \end{aligned}$$

## 6.9 Optimization independent translation correctness theorem

This section presents the main theorem provided by Layer 4 of our SVF which says that optimization independent translation correctness criterion `TCC` constitutes sufficient condition of the translation correctness predicate `corrTrans`.

Theorem 6.44 is a statement about two IL programs  $S$  and  $T$  and it says that if there exists a bisimulation relation such that  $S$  and  $T$  fulfill the optimization independent translation correctness criterion `TCC` w.r.t. that relation, then they fulfill the translation correctness predicate `corrTrans`.

**Theorem 6.44.** (*Optimization independent translation correctness theorem*)

$$\exists \mathcal{R}. \text{TCC}(S, T, \mathcal{R}) \implies \text{corrTrans}(S, T)$$

*Proof.* By the definition of the criterion TCC, we have to show

$$\begin{aligned} & \exists \mathcal{R} \Phi_S \Phi_T B_S B_S. \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \\ & \quad \text{bisimulation}((S, B_S), (T, B_T), \mathcal{R}) \\ \implies & \\ & \text{corrTrans}(S, T) \end{aligned}$$

By application of the existential elimination rule, we have to show the following statement for arbitrary but fixed program types  $\Phi_S$  and  $\Phi_T$ ; CFGB declarations  $B_S$  and  $B_T$ ; and a bisimulation relation  $\mathcal{R}$ :

$$\begin{aligned} & \text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge \\ & \quad \text{bisimulation}((S, B_S), (T, B_T), \mathcal{R}) \\ \implies & \\ & \text{corrTrans}(S, T) \end{aligned}$$

By Theorem 6.30, the following holds

$$\text{Sem}(S) = \text{Sem}'(S, B_S) \quad \text{and} \quad \text{Sem}(T) = \text{Sem}'(T, B_T).$$

By application of Theorem 6.39, from

$$\text{wtp}(S, \Phi_S) \wedge \text{wtp}(T, \Phi_T) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T)$$

follows

$$\text{Sem}'(S, B_S) = \text{Sem}''(S, B_S) \wedge \text{Sem}'(T, B_T) = \text{Sem}''(T, B_T).$$

By application of Theorem 6.42, from

$$\text{bisimulation}((S, B_S), (T, B_T), \mathcal{R})$$

follows

$$\text{Sem}''(S, B_S) = \text{Sem}''(T, B_T).$$

By application of standard HOL rules, from

$$\begin{aligned} & \text{Sem}''(S, B_S) = \text{Sem}''(T, B_T) \wedge \\ & \text{Sem}(S) = \text{Sem}'(S, B_S) \wedge \text{Sem}'(S, B_S) = \text{Sem}''(S, B_S) \\ & \text{Sem}(T) = \text{Sem}'(T, B_T) \wedge \text{Sem}'(T, B_T) = \text{Sem}''(T, B_T) \end{aligned}$$

follows

$$\text{Sem}(S) = \text{Sem}(T).$$

This is equivalent to the conclusion  $\text{corrTrans}(S, T)$  by the definition of the predicate  $\text{corrTrans}$ .  $\square$



## Chapter 7

---

### Translation correctness criteria for particular optimizations

This chapter presents the content of Layer 5 of our implementation of the SVF. In Sections 3.5, 3.6 and 6, we motivated that Layer 5 in our implementation of the SVF is to provide, for each optimization  $O$  in the chain of optimizations performed by the compiler, the following:

- the formalization of an optimization correctness criterion  $TCC_O$ , whose definition is specific to the definition of the optimization  $O$ ,
- the proof of a  $TCC_O$  criterion correctness theorem whose statement says that the specification of the  $TCC_O$  criterion is correct and has the following form:

*If two programs are the source and the target programs of the transformation  $T$  and they fulfill the optimization correctness criterion  $TCC_O$ , then the programs fulfill the optimization independent translation correctness criterion  $TCC$  provided by Layer 4.*

- the proof of an optimization correctness theorem whose statement is specific to the optimization  $O$  and has the following form:

*If two programs are the source and the target programs of the transformation  $T$  and they fulfill the optimization correctness criterion  $TCC_O$ , then the programs fulfill the translation correctness predicate  $\text{corrTrans}$  provided by the translation contract layer.*

In our implementation of the SVF, the statement of the optimization correctness theorem is the result of conjoining of Theorem 6.44 provided by Layer 4 and the above  $TCC_O$  criterion correctness theorem. The proof of this theorem is the ultimate result provided by this layer as the theorem is directly applicable in translation certificates generated by our compiler front-end. As mentioned in Section 6, the proof of this theorem is conducted in two steps: The first step is specific to the optimization  $O$  and applies the  $TCC_O$  criterion correctness theorem. The second step is optimization independent and makes use of the optimization independent translation correctness theorem. The idea behind the formal framework provided by Layers 5 and 4 is that it is structured in two parts comprising formalizations and theorems which are needed to conduct the first and the second step of the proof, respectively, cf. Section 3.5.

The rest of this chapter is organized as follows:

- Section 7.1 presents the implementation of the SVF for constant folding optimizations (CF). In particular, this section presents the formalization of an optimization correctness criterion  $TCC_{CF}$  which is an instance of the above criterion  $TCC_O$  and Theorem 7.27 which is the optimization correctness theorem in this section.
- Section 7.2 presents the implementation of the SVF for dead assignment elimination optimizations (DAE). In particular, this section present the formalization of an optimization correctness criterion  $TCC_{DAE}$  which is an instance of the above criterion  $TCC_O$  and Theorem 7.55 which is the optimization correctness theorem in this section.

- Section 7.3 presents the implementation of the SVF for nop insertion optimizations (NI). In particular, this section presents the formalization of an optimization correctness criterion  $TCC_{NI}$  which is an instance of the above criterion  $TCC_O$  and Theorem 7.72 which is the optimization correctness theorem in this section.
- Section 7.4 presents our implementation of the SVF for redundant assignment insertion optimizations (RAI). In particular, this section explains how the formal framework provided by the SVF for DAE optimizations can directly be reused in proofs of the RAI correctness generated by our compiler.
- Section 7.5 presents our implementation of the SVF for redundant assignment elimination optimizations (RAE). In particular, this section presents the formalization of an optimization correctness criterion  $TCC_{RAE}$  which is an instance of the above criterion  $TCC_O$  and Theorem 7.97 which is the optimization correctness theorem in this section.

## 7.1 SVF for CF optimizations

This section presents formalization of an optimization correctness criterion for CF optimizations,  $TCC_{CF}$ , and a corresponding optimization correctness theorem which is directly applied in translation certificates generated by our compiler front-end.

As aforementioned in Section 1.5.1, our implementation of the CF procedure is standard [1, 3, 119]. In the following, we give a general description of this procedure. The description only provides details which we need for the purpose of the explanation of the SVF for CF optimizations.

In general, the CF is performed by our compiler front-end in two steps:

The first step: The compiler performs a constant propagation analysis (CPA) on the input program. The result of this analysis,  $\mathcal{A}_{CPA}$ , is a mapping from program points to invariants which are partial mappings from program variables to constants. If, during the CPA, the compiler determines that whenever the flow of control reaches a program point  $pc$  the value of a variable  $v$  is invariantly equal to a constant  $i$ , then  $\mathcal{A}_{CPA}$  maps  $pc$  to  $inv$  and  $inv$  maps  $v$  to  $i$ .

The second step: The compiler takes the source program  $S$  and the CPA result  $\mathcal{A}_{CPA}$  as input and transforms  $S$  in two steps:

Constant propagation: For each program point  $pc$  and each expression  $e$  in the  $pc$ -th instruction and each variable  $v$  in  $e$ , the compiler checks if  $\mathcal{A}_{CPA}$  declares  $v$  as constant, i.e. if there exist an invariant  $inv$  and a constant value  $i$  such that  $\mathcal{A}_{CPA}$  maps  $pc$  to  $inv$  and  $inv$  maps  $v$  to  $i$ . If this holds, the compiler replaces  $v$  by  $i$ .

Constant folding: For each program point  $pc$  and each expression  $e$  in the  $pc$ -th instruction, the compiler checks if all operands in  $e$  are constant values and  $e$  can be evaluated at compile time. If this holds, the compiler evaluates  $e$  to a constant value  $i_e$  and replaces  $e$  by  $i_e$ .

It follows from the above description that the CF optimization performed by the compiler, which is a part of the FTV system described in this thesis, is structure preserving, i.e. does not modify the set of edges and nodes of the CFG of the source program. Therefore, the formalization of the optimization correctness criterion  $TCC_{CF}$  makes the following assumptions about the CFGs and the CFGB declarations involved in the CF optimization:

- The sets of nodes and edges of the CFGs of the source and the target programs are identical.
- The CFGB declarations for the source and the target programs of the CF are identical.
- The corresponding program points relation between program points of the source and the target programs is defined as identity, i.e. as a one-to-one correspondence between program points of the source and the target programs.

- For both the source and the target program, the allocation relation between program points and block positions is a one-to-one correspondence.
- The "includes" relation between blocks and block positions is a one-to-one correspondence.
- The CFGB declaration used to formalize the criteria  $TCC_{CF}$  comprises no nested blocks.

As aforementioned in Chapter 3, the SVF for CF optimizations also provides formalizations of further notions which are needed to formulate the  $TCC_{CF}$  criterion and to conduct the proof the optimization correctness theorem. These notions are as follows.

- The definition of  $TCC_{CF}$  is based on the definition of a function `cf_transrel_instr` which computes a translation relation over instruction pairs for CF optimizations. This function is an instance of the translation relation predicate  $TCC_0$  in Section 3.6.
- The proof of the optimization correctness theorem adheres to the proof scheme described in Chapter 3. The first step of this proof scheme is based on the notion of bisimulation and a bisimulation relation. Therefore, the SVF presented in this section provides the definition of a function `bisimrelCF` which computes a bisimulation relation  $\mathcal{R}_{CF}$  as a function of the source and the target programs and the CF result involved in a CF optimization. This function is an instance of the function `bisimrel0` in Section 3.6.

The rest of this section is organized as follows. Section 7.1.1 presents formation rules for the set of CPA results. Section 7.1.2 presents the definition of a function `bisimrelCF` which computes the bisimulation relation  $\mathcal{R}_{CF}$ . Section 7.1.3 presents the definitions of the function `cf_transrel_instr` and the criterion  $TCC_{CF}$ . Section 7.1.4 presents the optimization correctness theorem for CF optimizations.

### 7.1.1 Abstract syntax of CPA results

This section presents the definition of the abstract syntax of the CPA results. We begin the presentation by listing syntactic sets associated with this notion:

- instruction numbers **InstructionNr**,
- constant value environments **ConstantValueEnv**,
- invariant environments **InvariantEnv**;

and defining metavariables ranging over these sets:

- $pc$  is ranging over instruction numbers **InstructionNr**,
- $inv$  is ranging over constant value environments **ConstantValueEnv**, and
- $\mathcal{A}_{CPA}$  is ranging over invariant environments **InvariantEnv**.

Definition 7.1 gives formation rules for the set of CPA results **InvariantEnv**. A CPA result  $\mathcal{A}_{CPA}$  is a partial mapping from the set of instruction numbers **InstructionNr** to the set of invariants **ConstantValueEnv**. We call CPA results *invariant environments* and if a CPA result  $\mathcal{A}_{CPA}$  is well-formed, then it is total. An invariant  $inv$  is a partial mapping from program variables to values. The purpose of a CPA result is to model facts about the values of certain program variables during execution of a program which were determined by the compiler during the CPA of that program. Informally, the meaning of a CPA result  $\mathcal{A}_{CPA}$  is as follows. Given that the compiler performs the CPA on an IL program  $P$ , the result of this analysis is  $\mathcal{A}_{CPA}$ , then  $\mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge inv(v) = \text{Some}(val)$  iff the compiler determines that the value of a variable  $v$  is always equal  $val$  each time the flow of control transfers to the program point  $pc$ .

**Definition 7.1.**

$pc$	$\in$	<b>InstructionNr</b>	$=$	<b>Pc</b>
$inv$	$\in$	<b>ConstantValueEnv</b>	$=$	<b>Variable</b> $\rightsquigarrow$ <b>Value</b>
$\mathcal{A}_{CPA}$	$\in$	<b>InvariantEnv</b>	$=$	<b>InstructionNr</b> $\rightsquigarrow$ <b>ConstantValueEnv</b>

**7.1.2 Bisimulation relation for the CF optimization**

This section presents the definition of a function  $\text{bisimrel}_{CF}$  which computes a bisimulation relation for two IL<sup>''</sup> programs, which are the source and the target programs of a concrete CF optimization, and a CPA result.

Informally, our notion of a bisimulation relation  $\mathcal{R}_{CF}$ , which is a function of a concrete CF optimization, can be characterized as follows.

- By Definition 6.40,  $\mathcal{R}_{CF}$  has to be a subset of

$$\mathbf{Configuration''} \times \mathbf{Configuration''}.$$

- Being a function of a concrete CF optimization means that  $\mathcal{R}_{CF}$  is a function of the source and the target programs of that optimization,  $S$  and  $T$ , and a result of the CPA that was performed prior to that optimization,  $\mathcal{A}_{CPA}$ .
- As the CF optimization is a structure preserving optimization, i.e. it does not modify the sets of nodes and edges of the CFG of a program, the CFGB declarations  $B_S$  and  $B_T$  for  $S$  and  $T$  are also identical. For this reason, it must hold for those programs that they have the same observable behaviors, if it holds for two arbitrary partial executions of  $(S, B_S)$  and  $(T, B_T)$  of the same length, then they produce equal augmented configurations  $\sigma_S''$  and  $\sigma_T''$ . Therefore,  $\mathcal{R}_{CF}$  has to comprise pairs  $(\sigma_S'', \sigma_T'')$  consisting of augmented configurations which are equal

$$\forall (\sigma_S'', \sigma_T'') \in \mathcal{R}_{CF}. \sigma_S'' = \sigma_T''.$$

- As the proof of a statement saying that if  $(T, B_T)$  is a correct CF optimization of  $(S, B_S)$  implies  $\text{bisimulation}((S, B_S), (T, B_T), \mathcal{R}_{CF})$  has to be conducted by induction on the length of partial execution of  $(S, B_S)$ , we have to strengthen the induction invariant

$$M''((S, B_S), \text{init}(S, B_S), n) = M''((T, B_T), \text{init}(T, B_T), n)$$

which follows from the above requirement by an additional invariant saying that the augmented configuration  $M''((S, B_S), \text{init}(S, B_S), n)$  conforms with the CPA result  $\mathcal{A}_{CPA}$ . The notion of conformance is organized hierarchically: At first, we define the notion of conformance of a state  $s$  to an invariant  $inv$ . Then, we give the definition of the notion of conformance of an augmented configuration  $\sigma''$  to a CPA result  $\mathcal{A}_{CPA}$  which is based on the previous notion.

We begin the presentation of the definition of  $\mathcal{R}_{CF}$  with the formalization of the notion of conformance.

Definition 7.2 defines a predicate  $\text{confinv}$  on invariants and state which checks if a state  $s$  conforms with an invariant  $inv$ . An invariant  $inv$  is a mapping which models all informations about invariant variables for a program point which were determined by the compiler during the CPA and it is always defined for a program point  $pc$ . If the compiler determines during the analysis that whenever the flow of control transfers to a program point  $pc$  the value of a variable  $v$  in the context of a state of computation  $s$  is always equal  $val$ , then the invariant  $inv$  defined for  $pc$  comprises a binding  $v \mapsto val$ . For this reason, an invariant  $inv$  conforms to a state  $s$  iff all bindings which are in  $inv$  defined properly, i.e. if  $inv$  maps a variable  $v$  to a constant value  $val$ , then  $s$  also maps  $v$  to  $val$ .

**Definition 7.2.**

**confinv** : **ConstantValueEnv** × **State** → **Bool**  
**confinv**(*inv*, *s*) =  $\forall v \in \text{dom}(\textit{inv}). \textit{inv}(v) = s(v)$

Definition 7.3 defines a predicate **confcpare**s on a CPA result  $\mathcal{A}_{CPA}$  and an augmented configuration  $\sigma''$  which checks if  $\sigma''$  conforms with  $\mathcal{A}_{CPA}$ . A CPA result  $\mathcal{A}_{CPA}$  is a mapping which models all informations about invariant variables for all program points which were determined by the compiler during the CPA. A CPA result  $\mathcal{A}_{CPA}$  conforms to an augmented configuration  $\sigma''$  iff  $\mathcal{A}_{CPA}$  maps its program counter component *pc* to a well-defined invariant *inv* that conforms to its state component *s*.

**Definition 7.3.**

**confcpare**s : **InvariantEnv** × **Configuration''** → **Bool**  
**confcpare**s( $\mathcal{A}_{CPA}, (ss, bs, \textit{until}, \textit{bposstat}, \textit{predbid}, (\textit{tf}, \textit{af}, \textit{pc}, b, s))$ ) =  

$$\begin{cases} \text{confinv}(\textit{inv}, s) & \text{if } \mathcal{A}_{CPA}(\textit{pc}) = \text{Some}(\textit{inv}), \\ \text{False} & \text{otherwise} \end{cases}$$

With the definition of the conformance predicate at hand, we can give the definition of the bisimulation relation  $\mathcal{R}_{CF}$ .

As aforementioned, our main aim is to define a bisimulation relation over pairs of augmented configurations  $(\sigma''_S, \sigma''_T)$  which supports inductive reasoning about pairs of executions of IL' programs, i.e. we want to define a function **bisimrel**<sub>CF</sub> which takes an IL program *S*, its CF optimization *T*, a CFGB declaration<sup>1</sup> *B*, and a CPA result  $\mathcal{A}_{CPA}$  and computes a set of pairs consisting of augmented configurations such that

$$\forall n. (M''((S, B), \text{init}(S, B), n), M''((T, B), \text{init}(T, B), n)) \in \mathcal{R}_{CF})$$

By the definition of  $M''$ , we know that each augmented configuration of the form

$$M''((P, B), \sigma''_0, n) = (ss, n, ss, \textit{bposstat}, \textit{predbid}, \sigma)$$

is the result of *n* block-wise transfers of the flow of control in the CFGB which is declared by *B* for the program *P*. Further, we know from Definitions 6.36, 6.35 6.24, and 6.22 that

- the machine function  $M''$  in this equation uses the operational semantics of the language IL' to compute the result of partial execution  $M''((P, B), \sigma''_0, n)$ ,
- the operational semantics of the IL' language is defined by means of transfers of the flow of control from block position to block position in that CFGB,
- transfers of the flow are computed by the transition function **exec'**, and
- computing the successor configuration by the function **exec'** can result in switching of the mode of execution into the one of four modes: the normal mode, the exit mode, the emulation mode, and the exception mode.

In Section 6.4.2, we explained that computing the successor configuration by the function **exec'** can result in switching of the mode of execution into the emulation mode only, if the flow of control is within a block at a block position which is not the last one in that block and executing an

<sup>1</sup> As we previously mentioned, in the case of the CF optimization, respective CFGB declarations for *S* and *T*,  $B_S$  and  $B_T$ , are equal. So, we need to declare only one CFGB for both *S* and *T*.

instruction at a program point which is allocated to that block position results in an exception. In other words, if the flow of control leaves a block by transferring to a block position which is the first one in another block, then this transfer always results in switching into one of three modes: either the normal mode or the exception mode or the exit mode.

As we know what syntactic forms the augmented configurations have in those modes, we can define our bisimulation relation  $\mathcal{R}_{CF}$  as a union of three sets which are defined for a respective mode of execution and each of those sets fulfills the following properties:

- For each configuration pair  $(\sigma''_S, \sigma''_T)$  in the set, it holds  $\sigma''_S = \sigma''_T$ .
- If the set is defined for a particular mode of execution, then it holds for each configuration pair  $(\sigma''_S, \sigma''_T)$  in this set that  $\sigma''_S$  has the syntactic form which corresponds to that mode of execution, cf. Section 6.4.2 for the syntactic forms of augmented configurations in the respective modes of execution.
- For each configuration pair  $(\sigma''_S, \sigma''_T)$  in the set, it holds that  $\sigma''_S$  and  $\sigma''_T$  are results of partial executions of IL" programs  $(S, B)$  and  $(T, B)$  by the machine  $M''$  and that these executions which has the same length

$$\exists n. (M''((S, B), \text{init}(S, B), n), M''((T, B), \text{init}(T, B), n)) = (\sigma''_S, \sigma''_T))$$

- If the set is build for the normal mode of execution, then it additionally holds for each pair  $(\sigma''_S, \sigma''_T)$  in this set that  $\sigma''_S$  conforms with the CPA result  $\mathcal{A}_{CPA}$ .

In the following, we specify these three sets by giving the definitions of respective predicates which check if a configuration pair  $(\sigma''_S, \sigma''_T)$  in the respective set defined w.r.t. IL" programs  $(S, B)$  and  $(T, B)$ ; and a CPA result  $\mathcal{A}_{CPA}$ .

Definition 7.4 defines a function `bisimrel_CF_normblk` which checks if a configuration pair  $(\sigma''_S, \sigma''_T)$  is in a subset of the bisimulation relation  $\mathcal{R}_{CF}$  which is defined for the normal mode of execution and w.r.t. two IL" programs,  $(S, B)$  and  $(T, B)$ , and a CPA result  $\mathcal{A}_{CPA}$ .

**Definition 7.4.**

```

bisimrel_CF_normblk : Program × Program × BkPosEnv × InvariantEnv
                    × Configuration'' × Configuration'' → Bool

bisimrel_CF_normblk(S, T, (pid0, BP, BB, succBP, predB),  $\mathcal{A}_{CPA}$ ,  $\sigma''_S, \sigma''_T$ ) =
  ∃ bs ss pid bid pc predbid b s bt set.
     $\sigma''_S = \sigma''_T \wedge$ 
     $\sigma''_S = (ss, bs, ss, \text{NORMBLCK}(pid, bid, 1, 0, pc), predbid, (\text{NT}, \text{AB}_{\text{ok}}, pc, b, s)) \wedge$ 
     $M''((S, B), bs) = (ss, bs, ss, \text{NORMBLCK}(pid, bid, 1, 0, pc), predbid, (\text{NT}, \text{AB}_{\text{ok}}, pc, b, s)) \wedge$ 
     $M''((T, B), bs) = (ss, bs, ss, \text{NORMBLCK}(pid, bid, 1, 0, pc), predbid, (\text{NT}, \text{AB}_{\text{ok}}, pc, b, s)) \wedge$ 
     $BB(bid) = \text{Some}(pid) \wedge$ 
     $BP(pid) = \text{Some}(pid, bid, 1, 0, pc) \wedge$ 
     $predB(pid) = \text{Some}(set) \wedge$ 
     $(predbid, bt) \in set \wedge$ 
     $\text{confbuffer}(bt, b) \wedge$ 
     $\text{confcpares}(\mathcal{A}_{CPA}, (ss, bs, ss, \text{NORMBLCK}(pid, bid, 1, 0, pc), predbid, (\text{NT}, \text{AB}_{\text{ok}}, pc, b, s)))$ 

```

Definition 7.5 defines a function `bisimrel_CF_excbk` which checks if a configuration pair  $(\sigma''_S, \sigma''_T)$  is in a subset of the bisimulation relation  $\mathcal{R}_{CF}$  which is defined for the exception mode of execution and w.r.t. two IL" programs,  $(S, B)$  and  $(T, B)$ .

**Definition 7.5.**

$$\begin{aligned} \text{bisimrel\_CF\_excbck} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \\ &\quad \times \mathbf{Configuration} \times \mathbf{Configuration}'' \rightarrow \mathbf{Bool} \\ \text{bisimrel\_CF\_excbck}(S, T, (pid_0, BP, BB, succBP, predB), \sigma_S'', \sigma_T'') = \\ &\quad \exists bs \ ss \ pc \ predbid \ n \ s. \\ &\quad \sigma_S'' = \sigma_T'' \wedge \\ &\quad \sigma_S'' = (ss, bs, ss, \mathbf{EXCBLCK}, predbid, (\mathbf{NT}, \mathbf{AB}, pc, \mathbf{FLUSH}(n), s)) \wedge \\ &\quad M''((S, B), bs) = (ss, bs, ss, \mathbf{EXCBLCK}, predbid, (\mathbf{NT}, \mathbf{AB}, pc, \mathbf{FLUSH}(n), s)) \wedge \\ &\quad M''((T, B), bs) = (ss, bs, ss, \mathbf{EXCBLCK}, predbid, (\mathbf{NT}, \mathbf{AB}, pc, \mathbf{FLUSH}(n), s)) \end{aligned}$$

Definition 7.6 defines a function `bisimrel_CF_exitbck` which checks if a configuration pair  $(\sigma_S'', \sigma_T'')$  is in a subset of the bisimulation relation  $\mathcal{R}_{CF}$  which is defined for the exit mode of execution and w.r.t. two IL'' programs,  $(S, B)$  and  $(T, B)$ , and a CPA result  $\mathcal{A}_{CPA}$ .

**Definition 7.6.**

$$\begin{aligned} \text{bisimrel\_CF\_exitbck} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \\ &\quad \times \mathbf{Configuration}'' \times \mathbf{Configuration}'' \rightarrow \mathbf{Bool} \\ \text{bisimrel\_CF\_exitbck}(S, T, (pid_0, BP, BB, succBP, predB), \sigma_S'', \sigma_T'') = \\ &\quad \exists bs \ ss \ pc \ predbid \ n \ s. \\ &\quad \sigma_S'' = \sigma_T'' \wedge \\ &\quad \sigma_S'' = (ss, bs, ss, \mathbf{EXITBLCK}, predbid, (\mathbf{T}, \mathbf{AB}_{ok}, pc, \mathbf{FLUSH}(n), s)) \wedge \\ &\quad M''((S, B), bs) = (ss, bs, ss, \mathbf{EXITBLCK}, predbid, (\mathbf{T}, \mathbf{AB}_{ok}, pc, \mathbf{FLUSH}(n), s)) \wedge \\ &\quad M''((T, B), bs) = (ss, bs, ss, \mathbf{EXITBLCK}, predbid, (\mathbf{T}, \mathbf{AB}_{ok}, pc, \mathbf{FLUSH}(n), s)) \end{aligned}$$

Definition 7.7 defines a function `bisimrelCF` which checks if a configuration pair  $(\sigma_S'', \sigma_T'')$  is in the bisimulation relation  $\mathcal{R}_{CF}$  which is defined for two IL'' programs,  $(S, B)$  and  $(T, B)$ , and a CPA result  $\mathcal{A}_{CPA}$ .

**Definition 7.7.**

$$\begin{aligned} \text{bisimrel}_{CF} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \rightarrow \mathbf{BisimulationRelation} \\ \text{bisimrel}_{CF}(S, T, B, \mathcal{A}_{CPA}) = &\{ (\sigma_S'', \sigma_T'') \mid \text{bisimrel\_CF\_normbck}(S, T, B, \mathcal{A}_{CPA}, \sigma_S'', \sigma_T'') \vee \\ &\text{bisimrel\_CF\_excbck}(S, T, B, \sigma_S'', \sigma_T'') \vee \\ &\text{bisimrel\_CF\_exitbck}(S, T, B, \sigma_S'', \sigma_T'') \} \end{aligned}$$
**7.1.3 Optimization correctness criterion for the CF optimization**

This section presents the formalization of a translation relation predicate  $\text{TCC}_{CF}$  on two IL'' programs,  $S$  and  $T$ , and a CPA result  $\mathcal{A}_{CPA}$ , which we call optimization correctness criterion for the CF optimization. The parameters of  $\text{TCC}_{CF}$  denote source and target programs of a concrete CF optimization, and a result of the CP analysis that was performed by the compiler prior to that optimization, respectively, and  $\text{TCC}_{CF}$  checks if  $T$  is a correct CF optimization of  $S$  w.r.t.  $\mathcal{A}_{CPA}$ . In our implementation, the  $\text{TCC}_{CF}$  predicate defines an instance of the optimization correctness criterion  $\text{TCC}_0$  that was described in the overview of Layer 5 in Section 3.6

We begin the presentation by listing sets associated with the definition of  $\text{TCC}_{CF}$ :

- relations over l-value pairs `LValueTransRel_CF`,
- relations over expression pairs `ExpressionTransRel_CF`, and

- relations over instruction pairs **InstrTransRel\_CF**.

Definition 7.8 gives formation rules for those sets.

**Definition 7.8.**

$$\begin{aligned}
\mathbf{LValueTransRel\_CF} &= \mathcal{P}(\mathbf{LValue} \times \mathbf{LValue}) \\
\mathbf{ExpressionTransRel\_CF} &= \mathcal{P}(\mathbf{Expression} \times \mathbf{Expression}) \\
\mathbf{InstrTransRel\_CF} &= \mathcal{P}(\mathbf{Instruction} \times \mathbf{Instruction})
\end{aligned}$$

The rest of this section is organized as follows: The first part of this section presents the definition of a function `cf_transrel_expr` which computes a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$ . The second part of this section presents the definition of a function `cf_transrel_lv` which computes a CF optimization relation over l-value pairs  $(lv, lv')$  for a source and a target programs of a concrete CF optimization and a result of the CPA which was performed prior to that optimization. The definition of `cf_transrel_lv` uses the definition of `cf_transrel_expr`. The third part of this second presents the definition of a function `cf_transrel_instr` which computes a CF optimization relation over instruction pairs  $(instr, instr')$  for a source and a target programs of a concrete CF optimization and a result of the CPA which was performed prior to that optimization. The definition of `cf_transrel_instr` uses the definitions of `cf_transrel_lv` and `cf_transrel_expr`. The last part this section presents the definition of the optimization correctness criterion  $\mathbf{TCC}_{CF}$ .

Definition 7.9 defines a function `cf_transrel_expr_operand` which computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is an operand, i.e.  $e$  has the syntactic form  $o$ .

**Definition 7.9.**

$$\begin{aligned}
&\mathbf{cf\_transrel\_expr\_operand} : \mathbf{ConstantValueEnv} \rightarrow \mathbf{ExpressionTransRel\_CF} \\
&\mathbf{cf\_transrel\_expr\_operand}(inv) = \\
&\quad \{(e, e') \mid \exists i. (e, e') = (i, i) \vee \\
&\quad \quad \exists b. (e, e') = (b, b) \vee \\
&\quad \quad \exists v. (e, e') = (v, v) \wedge inv(v) = \mathbf{None} \vee \\
&\quad \quad \exists v i. (e, e') = (v, i) \wedge inv(v) = \mathbf{Some}(i) \vee \\
&\quad \quad \exists v b. (e, e') = (v, b) \wedge inv(v) = \mathbf{Some}(b) \vee \\
&\quad \quad \exists a i. (e, e') = (a[i], a[i]) \vee \\
&\quad \quad \exists a v. (e, e') = (a[v], a[v]) \wedge inv(v) = \mathbf{None} \vee \\
&\quad \quad \exists a v i. (e, e') = (a[v], a[i]) \wedge inv(v) = \mathbf{Some}(i) \}
\end{aligned}$$

Definition 7.10 defines a function `cf_transrel_expr_unmin` which computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a unary operand prefixed by the unary operator  $-$ , i.e.  $e$  has the syntactic form  $-o$ .

**Definition 7.10.**

$\text{cf\_transrel\_expr\_unmin} : \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF}$   
 $\text{cf\_transrel\_expr\_unmin}(inv) = \{(e, e') \mid$   
 $\exists i_1 i_2 i_3. (e, e') = (-i_1, i_2) \wedge -i_1 = i_2 \vee$   
 $\exists v_1 i_2. (e, e') = (-v, -v) \wedge inv(v) = \text{None} \vee$   
 $\exists v i_1 i_2. (e, e') = (-v, i_2) \wedge inv(v) = \text{Some}(i_1) \wedge -i_1 = i_2 \vee$   
 $\exists a i. (e, e') = (-a[i], -a[i]) \vee$   
 $\exists a v. (e, e') = (-a[v], -a[v]) \wedge inv(v) = \text{None} \vee$   
 $\exists a v i. (e, e') = (-a[v], -a[i]) \wedge inv(v) = \text{Some}(i) \vee$

Definition 7.11 defines a function `cf_transrel_expr_plus` which computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $+$ , i.e.  $e$  has the syntactic form  $o + o$ .

**Definition 7.11.**

$\text{cf\_transrel\_expr\_plus} : \mathbf{ConstantValueEnv} \rightarrow \mathbf{ExpressionTransRel\_CF}$ $\text{cf\_transrel\_expr\_plus}(inv) = \{(e, e') \mid$ $\exists i_1 i_2 i_3. (e, e') = (i_1 + i_2, i_3) \wedge i_1 + i_2 = i_3 \vee$ $\exists v_1 i_2. (e, e') = (v_1 + i_2, v_1 + i_2) \wedge inv(v_1) = \mathbf{None} \vee$ $\exists v_1 i_1 i_2 i_3. (e, e') = (v_1 + i_2, i_3) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge i_1 + i_2 = i_3 \vee$ $\exists a i_1 i_2. (e, e') = (a[i_1] + i_2, a[i_1] + i_2) \vee$ $\exists a v_1 i_2. (e, e') = (a[v_1] + i_2, a[v_1] + i_2) \wedge inv(v_1) = \mathbf{None} \vee$ $\exists a v_1 i_1 i_2. (e, e') = (a[v_1] + i_2, a[i_1] + i_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \vee$ $\exists v_1 i_2. (e, e') = (i_1 + v_2, i_1 + v_2) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists v_1 i_2 i_3. (e, e') = (i_1 + v_2, i_3) \wedge inv(v_2) = \mathbf{Some}(i_2) \wedge i_1 + i_2 = i_3 \vee$ $\exists v_1 v_2. (e, e') = (v_1 + v_2, v_1 + v_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$ $\exists v_1 v_2 i_2. (e, e') = (v_1 + v_2, v_1 + i_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists v_1 v_2 i_1. (e, e') = (v_1 + v_2, i_1 + v_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists v_1 v_2 i_1 i_2 i_3. (e, e') = (v_1 + v_2, i_3) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \wedge i_1 + i_2 = i_3 \vee$ $\exists a i_1 v_2. (e, e') = (a[i_1] + v_2, a[i_1] + v_2) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a i_1 v_2 i_2. (e, e') = (a[i_1] + v_2, a[i_1] + i_2) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a v_1 v_2. (e, e') = (a[v_1] + v_2, a[v_1] + v_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a v_1 v_2 i_2. (e, e') = (a[v_1] + v_2, a[v_1] + i_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a v_1 i_1 v_2. (e, e') = (a[v_1] + v_2, a[i_1] + v_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a v_1 i_1 v_2 i_2. (e, e') = (a[v_1] + v_2, a[i_1] + i_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a i_1 i_2. (e, e') = (i_1 + a[i_2], i_1 + a[i_2]) \vee$ $\exists a v_1 i_2. (e, e') = (v_1 + a[i_2], v_1 + a[i_2]) \wedge inv(v_1) = \mathbf{None} \vee$ $\exists a v_1 i_1 i_2. (e, e') = (v_1 + a[i_2], i_1 + a[i_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \vee$ $\exists a_1 i_1 a_2 i_2. (e, e') = (a_1[i_1] + a_2[i_2], a_1[i_1] + a_2[i_2]) \vee$ $\exists a_1 v_1 a_2 i_2. (e, e') = (a_1[v_1] + a_2[i_2], a_1[v_1] + a_2[i_2]) \wedge inv(v_1) = \mathbf{None} \vee$ $\exists a_1 v_1 i_1 a_2 i_2. (e, e') = (a_1[v_1] + a_2[i_2], a_1[i_1] + a_2[i_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \vee$ $\exists a i_1 v_2. (e, e') = (i_1 + a[v_2], i_1 + a[v_2]) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a i_1 v_2 i_2. (e, e') = (i_1 + a[v_2], i_1 + a[i_2]) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a v_1 v_2. (e, e') = (v_1 + a[v_2], v_1 + a[v_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a v_1 i_1 v_2. (e, e') = (v_1 + a[v_2], i_1 + a[v_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a v_1 v_2 i_2. (e, e') = (v_1 + a[v_2], v_1 + a[i_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a v_1 i_1 v_2 i_2. (e, e') = (v_1 + a[v_2], i_1 + a[i_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a_1 i_1 a_2 v_2. (e, e') = (a_1[i_1] + a_2[v_2], a_1[i_1] + a_2[v_2]) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[i_1] + a_2[v_2], a_1[i_1] + a_2[i_2]) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a_1 v_1 a_2 v_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[v_1] + a_2[v_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a_1 v_1 i_1 a_2 v_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[i_1] + a_2[v_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$ $\exists a_1 v_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[v_1] + a_2[i_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$ $\exists a_1 v_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[i_1] + a_2[i_2]) \wedge$ $inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \} $
--

Definition 7.12 defines a function `cf_transrel_expr` which computes a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that it holds for each expression pair  $(e, e')$  in that relation that its right component  $e'$  is a result of constant folding performed w.r.t. the invariant  $inv$ . An optimization relation `cf_transrel_expr(inv)` is defined as a union of optimization relations over expression pairs  $(e, e')$  such that each of those relations is defined for a respective syntactic form of the expressions  $e$  in the pairs  $(e, e')$ . The definitions of those relations build for expressions having the syntactic forms  $o$ ,  $-o$ , and  $o + o$  are defined by Definitions 7.9, 7.10, and 7.11, respectively. For the brevity, we moved the definitions of functions computing the sets for the remaining the syntactic forms of expression  $e$  in Chapter B. These functions are as follows.

1. The function `cf_transrel_expr_binmin` is defined by Definition B.5 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $-$ , i.e.  $e$  has the syntactic form  $o - o$ .
2. The function `cf_transrel_expr_mult` is defined by Definition B.6 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $*$ , i.e.  $e$  has the syntactic form  $o * o$ .
3. The function `cf_transrel_expr_and` is defined by Definition B.7 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $\wedge$ , i.e.  $e$  has the syntactic form  $o \wedge o$ .
4. The function `cf_transrel_expr_not` is defined by Definition B.2 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a unary operand prefixed by the unary operator  $\neg$ , i.e.  $e$  has the syntactic form  $\neg o$ .
5. The function `cf_transrel_expr_or` is defined by Definition B.8 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $\vee$ , i.e.  $e$  has the syntactic form  $o \vee o$ .
6. The function `cf_transrel_expr_eq` is defined by Definition B.9 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $=$ , i.e.  $e$  has the syntactic form  $o = o$ .
7. The function `cf_transrel_expr_neq` is defined by Definition B.10 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $\neq$ , i.e.  $e$  has the syntactic form  $o \neq o$ .
8. The function `cf_transrel_expr_lt` is defined by Definition B.11 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $<$ , i.e.  $e$  has the syntactic form  $o < o$ .
9. The function `cf_transrel_expr_le` is defined by Definition B.12 and computes a subset of a CF optimization relation over expression pairs  $(e, e')$  w.r.t. an invariant  $inv$  such that each expression  $e$  in the pair  $(e, e')$  is a binary expression consisting of two operands joined by a binary operator  $\leq$ , i.e.  $e$  has the syntactic form  $o \leq o$ .

**Definition 7.12.**

$\text{cf\_transrel\_expr} : \mathbf{ConstantValueEnv} \rightarrow \mathbf{ExpressionTransRel\_CF}$   
 $\text{cf\_transrel\_expr}(inv) = \{(e, e') \mid (e, e') \in \text{cf\_transrel\_expr\_operand}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_plus}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_binmin}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_mult}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_unmin}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_and}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_not}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_or}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_eq}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_neq}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_lt}(inv) \vee$   
 $(e, e') \in \text{cf\_transrel\_expr\_le}(inv) \}$

Now, we present the second part of the formalization of the optimization relation predicate  $\text{TCC}_{\text{CF}}$ , the definition of a function  $\text{cf\_transrel\_lv}$  which computes a CF optimization relation over l-value pairs  $(lv, lv')$  for two IL" programs and a CPA result.

The function  $\text{cf\_transrel\_lv}$  is used later on by a function  $\text{cf\_transrel\_instr\_assign}$  which computes a subset of a CF optimization relation over instruction pairs  $(instr, instr')$  in which the left component of each instruction pair  $(instr, instr')$  is an assignment, i.e.  $instr$  has the syntactic form  $lval := e$ .

The definitions of functions computing relations over l-value pairs make the following context assumptions:

1. The compiler performed the CPA on an IL program  $S$ . The result of this analysis is  $\mathcal{A}_{\text{CPA}}$ .
2. The compiler performed the CF optimization on  $S$ . The result of this optimization is a target program  $T$ .
3. The compiler generated a CFGB declaration  $B$  which is identical for both  $S$  and  $T$ , and all blocks in the CFGB declared by  $B$  have the length equal one.
4. There exists a program point  $pc$  which is allocated to a block position  $pid$  which is included in a block  $bid$ .
5. The  $pc$ -th instruction of  $S$  is an assignment, i.e. the  $pc$ -th instruction of  $S$  has the syntactic form  $lval := e$ .
6. The  $pc$ -th instruction of  $T$  is an assignment, i.e. the  $pc$ -th instruction of  $T$  has the syntactic form  $lval' := e'$ .
7. There exists a well-defined invariant  $inv$  such that  $\mathcal{A}_{\text{CPA}}(pc) = \text{Some}(inv)$  and the expression pair  $(e, e')$  is in the optimization relation  $\text{cf\_transrel\_expr}(inv)$ .

Definition 7.13 defines a function  $\text{cf\_transrel\_lv\_case1}$  which makes the above context assumptions 1. through 7. and computes a subset of a CF optimization relation over l-value pairs  $(lval, lval')$  w.r.t. an invariant  $inv$ , a program point  $pc$ , and an expression pair  $(e, e')$  such that the following holds for that relation:

1. In each l-value pair  $(lval, lval')$  in this relation, the left and the right components,  $lval$  and  $lval'$ , have both the syntactic form  $v$ .
2. The  $pc$ -th instruction of the source program  $S$  has the syntactic form  $v := e$ .
3. The  $pc$ -th instruction of the target program  $T$  has the syntactic form  $v := i$ , i.e.  $e$  has either the form  $i$  or it could be folded completely during the CF optimization.
4. The expression pair  $(e, i)$  is in the translation relation  $\text{cf\_transrel\_expr}(inv)$ .

**Definition 7.13.**

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case1} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \times \\
& \qquad \qquad \qquad \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case1}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set. \\
& \qquad pc' = \text{Suc}(pc) \wedge \\
& \qquad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \qquad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \qquad BB(bid') = \text{Some}(pid') \wedge \\
& \qquad predB(pid') = \text{Some}(set) \wedge \\
& \qquad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \qquad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \qquad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \qquad e' = i \wedge \\
& \qquad (e, i) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \qquad \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s(v \mapsto i)) \}
\end{aligned}$$

Definition 7.14 defines a function `cf_transrel_lv_case2` which makes the above context assumptions 1. through 7. and computes a subset of a CF optimization relation over l-value pairs  $(lval, lval')$  w.r.t. an invariant  $inv$ , a program point  $pc$ , and an expression pair  $(e, e')$  such that the following holds for that relation:

1. In each l-value pair  $(lval, lval')$  in this relation, the left and the right components,  $lval$  and  $lval'$ , have both the syntactic form  $v$ ,
2. the  $pc$ -th instruction of the source program  $S$  has the syntactic form  $v := e$ , and
3. the  $pc$ -th instruction of the target program  $T$  has the syntactic form  $v := b$ , i.e.  $e$  has either the form  $b$  or it could be folded completely during the CF optimization.
4. The expression pair  $(e, b)$  is in the translation relation `cf_transrel_expr(inv)`.

**Definition 7.14.**

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case2} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \times \\
& \qquad \qquad \qquad \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case2}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set. \\
& \qquad pc' = \text{Suc}(pc) \wedge \\
& \qquad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \qquad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \qquad BB(bid') = \text{Some}(pid') \wedge \\
& \qquad predB(pid') = \text{Some}(set) \wedge \\
& \qquad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \qquad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \qquad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \qquad e' = b \wedge \\
& \qquad (e, b) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \qquad \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s(v \mapsto b)) \}
\end{aligned}$$

Definition 7.15 defines a function `cf_transrel_lv_case3` which makes the above context assumptions 1. through 7. and computes a subset of a CF optimization relation over l-value pairs  $(lval, lval')$  w.r.t. an invariant  $inv$ , a program point  $pc$ , and an expression pair  $(e, e')$  such that the following holds for that relation:

1. In each l-value pair  $(lval, lval')$  in this relation, the left and the right components,  $lval$  and  $lval'$ , have both the syntactic form  $v$ ,
2. the  $pc$ -th instruction of the source program  $S$  has the syntactic form  $v:=e$ ,
3. the  $pc$ -th instruction of the target program  $T$  has the syntactic form  $v:=v'$ , i.e.  $e$  also has the form  $v'$  and it could not be folded during the CF optimization.
4. The expression pair  $(e, v)$  is in the translation relation  $cf\_transrel\_expr(inv)$ .

**Definition 7.15.**

$$\begin{aligned}
& cf\_transrel\_lv\_case3 : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \times \\
& \qquad \qquad \qquad \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& cf\_transrel\_lv\_case3((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set. \\
& \qquad pc' = \mathbf{Suc}(pc) \wedge \\
& \qquad succBP(pid, pc') = \mathbf{Some}(pid') \wedge \\
& \qquad BP(pid') = \mathbf{Some}(pid', bid', 1, 0, pc') \wedge \\
& \qquad BB(bid') = \mathbf{Some}(pid') \wedge \\
& \qquad predB(pid') = \mathbf{Some}(set) \wedge \\
& \qquad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \qquad \mathcal{A}_{CPA}(pc) = \mathbf{Some}(inv) \wedge \\
& \qquad \mathcal{A}_{CPA}(pc') = \mathbf{Some}(inv') \wedge \\
& \qquad inv'(v) = \mathbf{None} \wedge \\
& \qquad e' = v' \wedge \\
& \qquad (e, v') \in cf\_transrel\_expr(inv) \wedge \\
& \qquad confcpares(inv, s) \longrightarrow confcpares(inv', s) \}
\end{aligned}$$

Definition 7.16 defines a function  $cf\_transrel\_lv\_case19$  which makes the above context assumptions 1. through 7. and computes a subset of a CF optimization relation over l-value pairs  $(lval, lval')$  w.r.t. an invariant  $inv$ , a program point  $pc$ , and an expression pair  $(e, e')$  such that the following holds for that relation:

1. In each l-value pair  $(lval, lval')$  in this relation, the left and the right components have the syntactic forms  $a[v]$  and  $a[i]$ , respectively.
2. the  $pc$ -th instruction of the source program  $S$  has the syntactic form  $a[v]:=e$ ,
3. the  $pc$ -th instruction of the target program  $T$  has the syntactic form  $a[i]:=e'$ .
4. There exists an invariant  $inv$  such that  $\mathcal{A}_{CPA}(pc) = \mathbf{Some}(inv)$  and  $inv(a) = \mathbf{None}$  and  $inv(v) = \mathbf{Some}(i)$ .

**Definition 7.16.**

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case19} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \times \\
& \quad \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case19}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(a[v], a[i]) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv(a) = \text{None} \wedge \\
& \quad \quad inv(v) = \text{Some}(i) \wedge \\
& \quad \quad \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s) \}
\end{aligned}$$

Definition 7.17 defines a function `cf_transrel_lv` which makes the above context assumptions 1. through 7. and computes a CF optimization relation over l-value pairs  $(lval, lval')$  w.r.t. an invariant  $inv$ , a program point  $pc$ , and an expression pair  $(e, e')$  such that it holds for each l-value pair  $(lval, lval')$  in that relation that its right component  $lval'$  is a result of constant folding performed w.r.t.  $lval$ ,  $pc$ , and  $(e, e')$ . An optimization relation  $\text{cf\_transrel\_lv}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is defined as a union of nineteen optimization relations over l-value pairs  $(lval, lval')$  defined for respective syntactic forms of the expression  $e'$  and the l-value  $lval'$ .

For the brevity, we moved a part of the definitions of functions computing those relations in Chapter B. These definitions are as follows.

1. The definition of the relation  $\text{cf\_transrel\_lv\_case1}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  was described in Definition 7.13.
2. The definition of the relation  $\text{cf\_transrel\_lv\_case2}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  was described in Definition 7.14.
3. The definition of the relation  $\text{cf\_transrel\_lv\_case3}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  was described in Definition 7.15.
4. The definition of the relation  $\text{cf\_transrel\_lv\_case4}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.17
5. The definition of the relation  $\text{cf\_transrel\_lv\_case5}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.18
6. The definition of the relation  $\text{cf\_transrel\_lv\_case6}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.19
7. The definition of the relation  $\text{cf\_transrel\_lv\_case7}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.20
8. The definition of the relation  $\text{cf\_transrel\_lv\_case8}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.21
9. The definition of the relation  $\text{cf\_transrel\_lv\_case9}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.22
10. The definition of the relation  $\text{cf\_transrel\_lv\_case10}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.23
11. The definition of the relation  $\text{cf\_transrel\_lv\_case11}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.24

12. The definition of the relation  $\text{cf\_transrel\_lv\_case12}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.25
13. The definition of the relation  $\text{cf\_transrel\_lv\_case13}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.26
14. The definition of the relation  $\text{cf\_transrel\_lv\_case14}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.27
15. The definition of the relation  $\text{cf\_transrel\_lv\_case15}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.28
16. The definition of the relation  $\text{cf\_transrel\_lv\_case16}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.29
17. The definition of the relation  $\text{cf\_transrel\_lv\_case17}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.30
18. The definition of the relation  $\text{cf\_transrel\_lv\_case18}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  is given in Definition B.31
19. The definition of the relation  $\text{cf\_transrel\_lv\_case19}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e')$  was described in Definition 7.16.

**Definition 7.17.**

$$\begin{aligned} & \text{cf\_transrel\_lv} : \mathbf{BlkPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlkPosId} \times \mathbf{BlkId} \times \mathbf{State} \times \\ & \quad \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\ \text{cf\_transrel\_lv}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = & \\ & \{ (lw, lw') \mid (lw, lw') \in \text{cf\_transrel\_lv\_case1}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case2}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case3}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case4}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case5}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case6}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case7}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case8}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case9}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case10}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case11}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case12}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case13}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case14}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case15}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case16}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case17}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case18}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\ & \quad (lw, lw') \in \text{cf\_transrel\_lv\_case19}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \} \end{aligned}$$

The third part of the formalization of the optimization relation predicate  $\text{TCC}_{CF}$  is the definition of a function  $\text{cf\_transrel\_instr}$  which makes the same context assumptions about the source and the target programs, the CFGB declaration  $B$ , and a CPA result as the function  $\text{cf\_transrel\_lv}$  and computes a CF optimization relation over instruction pairs  $(instr, instr')$  w.r.t. program point  $pc$ , and a block position  $pid$ .

Definition 7.18 computes a subset of a CF optimization relation over instruction pairs  $\text{cf\_transrel\_instr\_assign}(B, \mathcal{A}_{CPA}, pc, pid, bid, s)$  such that it holds for each instruction pair  $(instr, instr')$  in this subset that

- both  $instr$  and  $instr'$  are  $pc$ -th instructions in a source and the target programs in the context,



**Definition 7.19.**

$$\begin{aligned}
& \text{cf\_transrel\_instr\_printi} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \hspace{15em} \rightarrow \mathbf{InstrTransRel\_CF} \\
& \text{cf\_transrel\_instr\_printi}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s) = \\
& \quad \{(\text{print}(e), \text{print}(e')) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set. \\
& \hspace{10em} \mathcal{A}_{CPA}!pc = \text{Some}(inv) \wedge \\
& \hspace{10em} (e, e') \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \hspace{10em} pc' = \text{Suc}(pc) \wedge \\
& \hspace{10em} succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \hspace{10em} BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \hspace{10em} BB(bid') = \text{Some}(pid') \wedge \\
& \hspace{10em} predB(pid') = \text{Some}(set) \wedge \\
& \hspace{10em} (bid, \mathbf{OFTYPE}) \in set \wedge \\
& \hspace{10em} \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \hspace{10em} \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s) \}
\end{aligned}$$

Definition 7.20 computes a subset of a CF optimization relation over instruction pairs  $\text{cf\_transrel\_instr\_branch}(B, \mathcal{A}_{CPA}, pc, pid, bid, s)$  such that it holds for each instruction pair  $(instr, instr')$  in this subset that

- both  $instr$  and  $instr'$  are  $pc$ -th instructions in a source and the target programs in the context,
- both  $instr$  and  $instr'$  are branch instructions, i.e. each pair has the syntactic form  $(\text{branch}(e, dst), \text{branch}(e', dst))$ ,
- the CFG of the source program comprises two edges  $(pc, pc + 1)$  and  $(pc, dst)$ ,
- the program points  $pc$ ,  $pc + 1$ , and  $dst$  are allocated to block positions  $pid$ ,  $pid'$ , and  $pid''$ , respectively, that are included in blocks  $bid$ ,  $bid'$ , and  $bid''$ , respectively,
- the CFGB declaration  $(pid_0, BP, BB, succBP, predB)$  declares  $bid'$  and  $bid''$  to be the successor blocks of  $bid$  and the block edges  $(bid, bid')$  and  $(bid, bid'')$  to be labeled with the buffertype  $\mathbf{FTYPE}$ ,
- the CPA result  $\mathcal{A}_{CPA}$  maps the program points  $pc$ ,  $pc + 1$ , and  $dst$  into well-defined invariants  $inv$ ,  $inv'$ , and  $inv''$ ,
- the expression pair  $(e, e')$  is in the CF optimization relation over expression pairs  $\text{cf\_transrel\_expr}(inv)$ , and
- if the IL" program  $(S, B)$  is executed and the flow of control transfers to the block position  $pid$  and the state component  $s$  in the current configuration conforms with the invariant  $inv$ , then the state component in the successor configuration remains unchanged and conforms with the invariants either  $inv'$  or  $inv''$ , which is dependent on whether the flow of control transfers to  $pid'$  or  $pid''$ , respectively.

**Definition 7.20.**

$$\begin{aligned}
& \text{cf\_transrel\_instr\_branch} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \hspace{15em} \rightarrow \mathbf{InstrTransRel\_CF} \\
& \text{cf\_transrel\_instr\_branch}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s) = \\
& \quad \{(\text{branch}(e, dst), \text{print}(e', dst)) \mid \exists inv\ inv'\ inv''\ pc'\ pid'\ pid''\ bid'\ bid''\ set'\ set'' . \\
& \hspace{10em} \mathcal{A}_{CPA}!pc = \text{Some}(inv) \wedge \\
& \hspace{10em} (e, e') \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \hspace{10em} pc' = \text{Suc}(pc) \wedge \\
& \hspace{10em} succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \hspace{10em} BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \hspace{10em} BB(bid') = \text{Some}(pid') \wedge \\
& \hspace{10em} predB(pid') = \text{Some}(set') \wedge \\
& \hspace{10em} (bid, \mathbf{FTYPE}) \in set' \wedge \\
& \hspace{10em} \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \hspace{10em} \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s) \wedge \\
& \hspace{10em} succBP(pid, dst) = \text{Some}(pid'') \wedge \\
& \hspace{10em} BP(pid'') = \text{Some}(pid'', bid'', 1, 0, dst) \wedge \\
& \hspace{10em} BB(bid'') = \text{Some}(pid'') \wedge \\
& \hspace{10em} predB(pid'') = \text{Some}(set'') \wedge \\
& \hspace{10em} (bid, \mathbf{FTYPE}) \in set'' \wedge \\
& \hspace{10em} \mathcal{A}_{CPA}(dst) = \text{Some}(inv'') \wedge \\
& \hspace{10em} \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv'', s) \}
\end{aligned}$$

Definition 7.21 computes a subset of a CF optimization relation over instruction pairs  $\text{cf\_transrel\_instr\_goto}(B, \mathcal{A}_{CPA}, pc, pid, bid, s)$  such that it holds for each instruction pair  $(instr, instr')$  in this subset that

- both  $instr$  and  $instr'$  are  $pc$ -th instructions in a source and the target programs in the context,
- both  $instr$  and  $instr'$  are goto instructions, i.e. each pair has the syntactic form  $(\text{goto}(dst), \text{goto}(dst))$ ,
- the CFG of the source program comprises an edge  $(pc, dst)$ ,
- the program points  $pc$  and  $dst$  are allocated to block positions  $pid$  and  $pid'$ , respectively, that are included in blocks  $bid$  and  $bid'$ , respectively,
- the CFGB declaration declares  $bid$  as the predecessor block of  $bid'$  and the edge  $(bid, bid')$  is labeled with the buffertype  $\mathbf{FTYPE}$ ,
- the CPA result  $\mathcal{A}_{CPA}$  maps the program points  $pc$  and  $dst$  into well-defined invariants  $inv$  and  $inv'$ ,
- if the IL" program  $(S, B)$  is executed and the flow of control transfers to the block position  $pid$  and the state component  $s$  in the current configuration conforms with the invariant  $inv$ , then the state component in the successor configuration remains unchanged and conforms with the invariant  $inv'$ .

**Definition 7.21.**

$$\begin{aligned}
& \text{cf\_transrel\_instr\_goto} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \hspace{15em} \rightarrow \mathbf{InstrTransRel\_CF} \\
& \text{cf\_transrel\_instr\_goto}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s) = \\
& \quad \{(\text{goto}(dst), \text{goto}(dst)) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ pc'\ set. \\
& \hspace{10em} \mathcal{A}_{CPA}!pc = \mathbf{Some}(inv) \wedge \\
& \hspace{10em} succBP(pid, dst) = \mathbf{Some}(pid') \wedge \\
& \hspace{10em} BP(pid') = \mathbf{Some}(pid', bid', 1, 0, dst) \wedge \\
& \hspace{10em} BB(bid') = \mathbf{Some}(pid') \wedge \\
& \hspace{10em} predB(pid') = \mathbf{Some}(set') \wedge \\
& \hspace{10em} (bid, \mathbf{FTYPE}) \in set' \wedge \\
& \hspace{10em} \mathcal{A}_{CPA}(dst) = \mathbf{Some}(inv') \wedge \\
& \hspace{10em} \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s) \}
\end{aligned}$$

Definition 7.22 defines a function `cf_transrel_instr` which makes the same context assumptions as the functions `cf_transrel_instr_assign`, `cf_transrel_instr_printi`, `cf_transrel_instr_branch`, and `cf_transrel_instr_goto` and computes a CF optimization relation over instruction pairs `cf_transrel_instr( $\mathcal{A}_{CPA}, pc, pid, bid, s$ )` in which each instruction pair  $(instr, instr')$  fulfills the following property:

- Both  $instr$  and  $instr'$  are  $pc$ -th instructions in a source and the target programs in the context.
- The program point  $pc$  is allocated to a block position  $pid$  is included in a blocks  $bid$  which has the length equal one.

Further, it holds that the relation `cf_transrel_instr( $\mathcal{A}_{CPA}, pc, pid, bid, s$ )` is a union of five disjoint sets which are defined as follows.

1. The first set comprises assignment instruction pairs which are in the relation `cf_transrel_instr_assign( $\mathcal{A}_{CPA}, pc, pid, bid, s$ )`.
2. The second set comprises printi instruction pairs which are in the relation `cf_transrel_instr_printi( $\mathcal{A}_{CPA}, pc, pid, bid, s$ )`.
3. The third set comprises branch instruction pairs which are in the relation `cf_transrel_instr_branch( $\mathcal{A}_{CPA}, pc, pid, bid, s$ )`.
4. The fourth set comprises goto instruction pairs which are in the relation `cf_transrel_instr_goto( $\mathcal{A}_{CPA}, pc, pid, bid, s$ )`.
5. The fifth set is equal  $\{(\mathbf{exit}, \mathbf{exit})\}$ .

**Definition 7.22.**

$$\begin{aligned}
& \text{cf\_transrel\_instr} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \hspace{15em} \rightarrow \mathbf{InstrTransRel\_CF} \\
& \text{cf\_transrel\_instr}(B, \mathcal{A}_{CPA}, pc, pid, bid, s) = \text{cf\_transrel\_instr\_assign}(B, \mathcal{A}_{CPA}, pc, pid, bid, s) \cup \\
& \hspace{10em} \text{cf\_transrel\_instr\_printi}(B, \mathcal{A}_{CPA}, pc, pid, bid, s) \cup \\
& \hspace{10em} \text{cf\_transrel\_instr\_branch}(B, \mathcal{A}_{CPA}, pc, pid, bid, s) \cup \\
& \hspace{10em} \text{cf\_transrel\_instr\_goto}(B, \mathcal{A}_{CPA}, pc, pid, bid, s) \cup \\
& \hspace{10em} \{(\mathbf{exit}, \mathbf{exit})\}
\end{aligned}$$

Finally, with the definitions of the conformance predicate `confcpares` and the function `cf_transrel_instr`, we can give the definition of the optimization correctness criterion  $\text{TCC}_{CF}$  on two IL programs,  $S$  and  $T$ , a CFGB declaration  $B$ , and a CPA result  $\mathcal{A}_{CPA}$  which formalizes what does it mean that  $T$  is a correct CF optimization of  $S$  w.r.t.  $B$  and  $\mathcal{A}_{CPA}$ .

In the beginning of the presentation of the definition of the function `cf_transrel_instr`, we explained that this function makes the following context assumptions: The compiler takes an IL program  $S$  and performs the CPA on it. The result of this analysis is  $\mathcal{A}_{CPA}$ . The result of the CF optimization is an IL program  $T$ . Additionally, the compiler generates a CFGB declaration  $B$  for both  $S$  and  $T$  such that the allocation mappings from the program points of those programs to block positions are identical. The definition of our optimization correctness criterion  $\text{TCC}_{CF}$  makes these assumptions explicit.

The definition of  $\text{TCC}_{CF}$  consists of two conjuncts: The first conjunct denotes a well-formedness criterion for the entry block of  $B$  and the input of an IL program w.r.t. a CPA result. The second conjunct denotes an optimization correctness criterion on the set of pairs consisting of corresponding instructions of a source program  $S$  and a target program  $T$ .

Definition 7.23 defines the first conjunct  $\text{TCC\_CF\_entry\_block}(B, \mathcal{A}_{CPA}, I)$  which checks if the entry block of a CFGB declaration  $B$  is well-formed and if the initial state  $\text{mapof}(I)$ , which is computed from a program input  $I$ , conforms with the invariant  $inv_0$ , which was computed by the compiler for the entry program point 0. The entry block of  $B$ ,  $bid_0$ , is well-formed iff it includes the entry block position of  $B$ ,  $pid_0$ , has the length equal one, the 0-th program point is allocated to  $pid_0$ , and is declared by  $B$  to be among his own predecessor blocks.

**Definition 7.23.**

```

TCC_CF_entry_block : BlkPosEnv × InvariantEnv × Input → Bool
TCC_CF_entry_block( $B, \mathcal{A}_{CPA}, I$ ) =
  let
    ( $pid_0, BP, BB, succBP, predB$ ) =  $B$ 
  in
    ∃  $bid_0$  set.  $BP(pid_0) = \text{Some}(pid_0, bid_0, 1, 0, 0) \wedge$ 
       $predB(pid_0) = \text{Some}(set) \wedge$ 
       $(bid_0, \text{FTYPE}) \in set$ 
    ∧
    ∃  $inv_0$ .  $\mathcal{A}_{CPA}(0) = \text{Some}(inv_0) \wedge \text{conf}(inv_0, \text{mapof}(I))$ 
  end

```

Definition 7.24 defines the predicate  $\text{TCC\_CF\_normal\_block}$  on a source program  $S$ , a target program  $T$ , a CFGB declaration  $B$ , a CPA result  $\mathcal{A}_{CPA}$ , a state  $s$ , and a block  $bid$  which checks if the block  $bid$  in the CFGB  $(S, B)$  and the block  $bid$  in the CFGB  $(T, B)$  fulfill the following optimization correctness criterion for the block pair  $(bid, bid)$ :  $\text{TCC\_CF\_normal\_block}(S, T, B, \mathcal{A}_{CPA}, s, bid)$  holds true iff there exist  $pc$ -th program points  $pc$  in  $S$  and  $T$ ; and a block position  $pid$  such that  $pc$  is allocated to  $pid$ ,  $pid$  is included by the block  $bid$ , and the pair  $(instrs!pc, instrs!pc)$  consisting of the  $pc$ -th instructions of  $S$  and  $T$ , respectively, are in the CF optimization relation `cf_transrel_instr` $(B, \mathcal{A}_{CPA}, pc, pid, bid, s)$ .

**Definition 7.24.**

```

TCC_CF_normal_block : Program × Program × BlkPosEnv × InvariantEnv × State
                                × BlkId → Bool

TCC_CF_normal_block(S, T, B, A_CPA, s, bid) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    ∃ pid pc. BB(bid) = Some(pid) ∧
              BP(pid) = Some(pid, bid, 1, 0, pc) ∧
              (instrs!pc, instrs'!pc) ∈ cf_transrel_instr(B, A_CPA, pc, pid, bid, s)
  end

```

Definition 7.25 defines our optimization correctness criterion for the CF optimizations on a source program  $S$ , a target program  $T$ , a CFGB declaration  $B$ , and a CPA result  $\mathcal{A}_{CPA}$  which checks if  $T$  is a correct CF optimization of  $S$  w.r.t.  $VB$  and  $\mathcal{A}_{CPA}$ . According to the definition of  $TCC_{CF}$ , an IL program  $T$  is a correct CF optimization of  $S$  iff the entry blocks of IL” programs  $(S, B)$  and  $(T, B)$  are well-formed and the initial state computed from the input of  $S$  is well-formed w.r.t. to the CPA result  $\mathcal{A}_{CPA}$ .

**Definition 7.25.**

```

TCC_CF : Program × Program × BlkPosEnv × InvariantEnv → Bool
TCC_CF(S, T, B, A_CPA) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    TCC_CF_entry_block(B, A_CPA, I)
    ∧
    ∀ s bid. TCC_CF_normal_block(S, T, B, A_CPA, s, bid)
  end

```

**7.1.4 Verification of the optimization correctness criterion  $TCC_{CF}$** 

This section presents the theorems which we proved in order to verify the specification of the optimization correctness criterion  $TCC_{CF}$  presented in the previous section. The main result in this section is a theorem which can be used directly in translation certificates generated by our compiler.

To verify the specification of the criterion  $TCC_{CF}$ , we proved Theorem 7.26 which is a statement about a source and a target programs of a concrete CF optimization,  $S$  and  $T$ , a program type  $\Phi$ , a CFGB declaration  $B$ , and a result of the CP analysis which was performed on  $S$  prior to that optimization,  $\mathcal{A}_{CPA}$ , and it says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B$  is well-formed w.r.t.  $S$  and  $T$ ; and  $S$  and  $T$  fulfill the optimization correctness criterion  $TCC_{CF}$  w.r.t.  $B$  and  $\mathcal{A}_{CPA}$ , then  $S$  and  $T$  fulfill the optimization independent translation correctness  $TCC$  w.r.t. the bisimulation relation  $\text{bisimrel}_{CF}(S, T, B, \mathcal{A}_{CPA})$ .

**Theorem 7.26.**

$$\begin{aligned}
& \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge \text{TCC}_{\text{CF}}(S, T, B, \mathcal{A}_{\text{CPA}}) \\
& \implies \\
& \text{TCC}(S, T, \text{bisimrel}_{\text{CF}}(S, T, B, \mathcal{A}_{\text{CPA}}))
\end{aligned}$$

□

Finally, we present the main result in this section, a theorem which is a statement about a source and a target programs of a concrete CF optimization,  $S$  and  $T$ , a program type  $\Phi$ , a CFGB declaration  $B$ , and a result of the CP analysis which was performed on  $S$  prior to that optimization,  $\mathcal{A}_{\text{CPA}}$ , and it says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B$  is well-formed w.r.t.  $S$  and  $T$ ; and  $S$  and  $T$  fulfill the optimization correctness criterion  $\text{TCC}_{\text{CF}}$  w.r.t.  $B$  and  $\mathcal{A}_{\text{CPA}}$ , then  $S$  and  $T$  fulfill the translation correctness predicate  $\text{corrTrans}$ .

**Theorem 7.27.**

$$\begin{aligned}
& \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge \text{TCC}_{\text{CF}}(S, T, B, \mathcal{A}_{\text{CPA}}) \\
& \implies \\
& \text{corrTrans}(S, T)
\end{aligned}$$

*Proof.*

$$\begin{aligned}
& \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge \text{TCC}_{\text{CF}}(S, T, B, \mathcal{A}_{\text{CPA}}) \\
& \implies \text{[by application of Theorem 7.26]} \\
& \text{TCC}(S, T, \text{bisimrel}_{\text{CF}}(S, T, B, \mathcal{A}_{\text{CPA}})) \\
& \implies \text{[by application of the existential introduction rule]} \\
& \exists \mathcal{R}. \text{TCC}(S, T, \mathcal{R}). \\
& \implies \text{[by application of Theorem 6.44]} \\
& \text{corrTrans}(S, T)
\end{aligned}$$

□

Theorem 7.27 is an instance of the corollary (I) in Section 3.6 and is directly applicable in translation certificates which are generated by the compiler for each CF optimization, see the end of Section 3.6 for a general application scheme of this theorem.

## 7.2 SVF for DAE optimizations

This section presents formalization of an optimization correctness criterion for DAE optimizations,  $\text{TCC}_{\text{DAE}}$ , and a corresponding optimization correctness theorem which is directly applied in translation certificates generated by our compiler front-end.

As aforementioned in Section 1.5.1, our implementation of the DAE procedure is standard [1, 3, 119]. In the following, we give a general description this procedure. The description only provides details which we need for the purpose of the explanation of the SVF for DAE optimizations.

In general, the DAE is performed by our compiler front-end in two steps:

The first step: The compiler performs a liveness analysis (LA) on the input program. The LA determines for each node  $pc$  in the node set of the CFG of a program, which variables are live at the exit from  $pc$  where a variable is live at the exit from a node  $pc$  if there exists a path from  $pc$  to a node  $pc'$  such that the  $pc'$ -th instruction of the program uses that variable and the nodes between  $pc$  and  $pc'$  do not re-define the variable. In the following, we call a variable *live in* or *live out* if it may be live at the entry into a CFG node or it may be live at the exit from a CFG node, respectively. The result of this analysis,  $\mathcal{A}_{LA}$ , is a mapping from program points to tuples of sets of program variables. Each tuple has the syntactic form  $(use, def, in, out)$ . If, during the LA, the compiler determines that whenever the flow of control leaves a node  $pc$  a variable  $v$  is live in or live out, then there exist sets  $use$ ,  $def$ ,  $in$ , and  $out$  such that  $\mathcal{A}_{LA}$  maps  $pc$  to the tuple  $(use, def, in, out)$  and  $v \in in$  or  $v \in out$ , respectively.

The second step: The compiler takes the source program  $S$  and the LA result  $\mathcal{A}_{CPA}$  as input and transforms  $S$  as follows. For each program point  $pc$  and each assignment of syntactic form  $v := e$ , the compiler checks if  $\mathcal{A}_{LA}$  declares  $v$  as live out at the program point  $pc$ . If it does not hold, the compiler replaces the assignment by a goto instruction `goto( $pc + 1$ )` which emulates a nop instruction.

It follows from the above description that the DAE optimization is structure preserving, i.e. does not modify the set of edges and nodes of the CFG of the source program. Therefore, the formalization of the optimization correctness criterion  $TCC_{DAE}$  makes the following assumptions about the CFG's and the CFGB declarations involved in the DAE optimization:

- The sets of nodes and edges of the CFG's of the source and the target programs are identical,
- The CFGB declarations for the source and the target programs of the DAE are identical,
- The corresponding program points relation between program points of the source and the target programs is defined as identity, i.e. as a one-to-one correspondence between program points of the source and the target programs,
- For both the source and the target program, the allocation relation between program points and block positions is a one-to-one correspondence,
- The "includes" relation between blocks and block positions is a one-to-one correspondence,
- The CFGB declaration used to formalize the criteria  $TCC_{DAE}$  comprises no nested blocks.

As aforementioned in Chapter 3, the SVF for DAE optimizations also provides formalizations of further notions which are needed to formulate the  $TCC_{DAE}$  criterion and to conduct the proof the optimization correctness theorem. These notions are as follows.

- The definition of  $TCC_{DAE}$  is based on the definition of a function `dae_transrel_instr` which computes a translation relation over instruction pairs for DAE optimizations. This function is an instance of the translation relation predicate `transrel0` in Chapter 3.
- The proof of the optimization correctness theorem adheres to the proof scheme described in Chapter 3. The first step of this proof scheme is based on the notion of bisimulation and a bisimulation relation. Therefore, the SVF presented in this section provides the definition of a function `bisimrelDAE` which computes a bisimulation relation  $\mathcal{R}_{DAE}$  as a function of the source and the target programs and the LA result involved in a DAE optimization. This function is an instance of the function `bisimrel0` in Chapter 3.

The rest of this section is organized as follows. Section 7.2.1 presents formation rules for the set of LA results. Section 7.2.2 presents the definitions of function `bisimrelDAE` which computes the bisimulation relation  $\mathcal{R}_{DAE}$ . Section 7.2.3 presents the definitions of the function `dae_transrel_instr` and the criterion  $TCC_{DAE}$ . Section 7.2.4 presents the optimization correctness theorem for DAE optimizations.

### 7.2.1 Abstract syntax of LA results

This section presents the definition of the abstract syntax of the LA results. We begin the presentation by listing syntactic sets associated with this notion:

- "use" variables **UseSet**,
- "def" variables **DefSet**,
- "in" variables **InSet**,
- "out" variables **OutSet**,
- tuples of sets of variables **VarSets**, and
- variable set environments **VarSetsEnv**;

and defining metavariables ranging over these sets:

- *use* ranges over "use" variables **UseSet**,
- *def* ranges over "def" variables **DefSet**,
- *in* ranges over "in" variables **InSet**,
- *out* ranges over "out" variables **OutSet**,
- *varsets* ranges over tuples of sets of variables **VarSets**, and
- $\mathcal{A}_{LA}$  ranges over variable set environments **VarSetsEnv**;

Definition 7.28 gives formation rules for the set of the LA results **VarSetsEnv**. An LA result  $\mathcal{A}_{LA}$  is a partial mapping from the set of instruction numbers **InstructionNr** to the set of tuples of variable sets **VarSets**. We call a LA result a *variable set environment* and if an LA result  $\mathcal{A}_{LA}$  is well-formed, then it is total. A tuple  $varsets = (use, def, in, out)$  consists of a set of "use" variables *use*, a set of "def" variables *def*, a set of "in" variables *in*, and a set of "out" variables *out*. The purpose of a LA result  $\mathcal{A}_{LA}$  is to model facts about the *live* variables of a program which were determined by the compiler during a concrete LA. A variable *v* is live at the exit from a node *pc* in the CFG of a program if there exists a path from *pc* to a node *pc'* such that the *pc'*-th instruction of the program uses *v* and the nodes between *pc* and *pc'* do not re-define the variable. In the following, we call a variable *live in* or *live out* if it may be live at the entry into a node or it may be live at the exit from a node, respectively. Thus, the LA determines for each program program point, which variables are live out at that program point. Informally, the meaning of a LA result  $\mathcal{A}_{LA}$  is as follows. Given that the compiler performs the LA on an IL program *P*, the result of this analysis is  $\mathcal{A}_{LA}$ , then  $\mathcal{A}_{LA}(pc) = \text{Some}(use, def, in, out) \wedge v \in out$  iff the compiler determines that the value of a variable *v* is live out each time the flow of control transfers from the program point *pc* to its successor.

#### Definition 7.28.

<i>use</i>	$\in \mathbf{UseSet}$	$= \mathcal{P}(\mathbf{Variable})$
<i>def</i>	$\in \mathbf{DefSet}$	$= \mathcal{P}(\mathbf{Variable})$
<i>in</i>	$\in \mathbf{InSet}$	$= \mathcal{P}(\mathbf{Variable})$
<i>out</i>	$\in \mathbf{OutSet}$	$= \mathcal{P}(\mathbf{Variable})$
<i>varsets</i>	$\in \mathbf{VarSets}$	$= \mathbf{UseSet} \times \mathbf{DefSet} \times \mathbf{InSet} \times \mathbf{OutSet}$
$\mathcal{A}_{LA}$	$\in \mathbf{VarSetsEnv}$	$= \mathbf{InstructionNr} \rightsquigarrow \mathbf{VarSets}$

### 7.2.2 Bisimulation relation for the DAE optimization

This section presents the definition of a function  $\text{bisimrel}_{\text{DAE}}$  which computes a bisimulation relation for two IL programs, which are the source and the target programs of a DAE optimization, and a result of the LA which was performed prior to this optimization.

Informally, our notion of a bisimulation relation  $\mathcal{R}_{\text{DAE}}$  can be characterized as follows.

- $\mathcal{R}_{\text{DAE}}$  is a function of a DAE optimization which is described by the following values:
  1. a source IL program  $S$ ,
  2. a target IL program  $T$ ,
  3. a CFGB declaration  $B$ , and
  4. a result of the LA,  $\mathcal{A}_{LA}$ .
- By Definition 6.40,  $\mathcal{R}_{\text{DAE}}$  has to be a subset of

**Configuration''**  $\times$  **Configuration''** .

- As the DAE is a structure preserving optimization, i.e. it does not modify the sets of nodes and edges in the CFG of the program  $S$ , the CFGB declarations  $B_S$  and  $B_T$  for  $S$  and  $T$ , respectively, are also identical,  $B_S = B_T$ . Therefore, we use in our SVF for DAE optimizations only one CFGB declaration,  $B$ , which for both  $S$  and  $T$ . On the other hand, our compiler performs DAE optimizations by replacing assignments to variables which are not live out by goto instructions emulating nop instructions. Thus, we know that it holds for two arbitrary partial executions of  $(S, B)$  and  $(T, B)$  of the same length that if they produce augmented configurations  $\sigma''_S$  and  $\sigma''_T$ , then all their corresponding components describing position in the CFGB are equal.

$\forall n S T B.$

$$M''((S, B), \text{init}(S, B), n) = \sigma''_S \wedge$$

$$M''((T, B), \text{init}(T, B), n) = \sigma''_T$$

$\longrightarrow$

$$\exists ss bs bposstat predbid tf af pc b s ss' bs' bposstat' predbid' tf' af' pc' b' s'.$$

$$\sigma''_S = (ss, bs, ss, bposstat, predbid, (tf, af, pc, b, s)) \wedge$$

$$\sigma''_T = (ss', bs', ss', bposstat', predbid', (tf', af', pc', b', s')) \wedge$$

$$ss = ss' \wedge$$

$$bs = bs' \wedge$$

$$bs = n \wedge$$

$$bposstat = bposstat' \wedge$$

$$predbid = predbid' \wedge$$

$$tf = tf' \wedge$$

$$af = af' \wedge$$

$$pc = pc' \wedge$$

$$b = b'$$

- As the proof of a theorem saying that

*If  $(T, B)$  is a correct DAE optimization of  $(S, B)$  implies  $\text{bisimulation}((S, B), (T, B), \mathcal{R}_{\text{DAE}})$  has to be conducted by induction on the length of partial execution of  $(S, B_S)$  and we know that the state components  $s$  and  $s'$  in the configurations  $\sigma''_S$  and  $\sigma''_T$  in the above invariant are not necessarily equal, we have to strengthen the above induction invariant by an additional statement expressing a property about two states and a LA result which, informally, says the following:*

*For all variables  $v$  which are live in at the program point  $pc$  holds  $s(v) = s'(v)$ .*

In order to be able to express the statements of this kind, we formalized the notion of conformance of two states with a LA result.

We begin the presentation of the definition of  $\mathcal{R}_{\text{DAE}}$  with the formalization of the notion of conformance.

Definition 7.29 defines a function `oper2use` which computes the set of "use" variables of an operand.

**Definition 7.29.**

$$\begin{aligned} \text{oper2use} &: \text{Operand} \rightarrow \text{UseSet} \\ \text{oper2use}(i) &= \{\} \\ \text{oper2use}(b) &= \{\} \\ \text{oper2use}(v) &= \{v\} \\ \text{oper2use}(a[i]) &= \{a\} \\ \text{oper2use}(a[v]) &= \{a, v\} \end{aligned}$$

Definition 7.30 defines a function `expr2use` which computes the set of "use" variables of an expression.

**Definition 7.30.**

$$\begin{aligned} \text{expr2use} &: \text{Expression} \rightarrow \text{UseSet} \\ \text{expr2use}(o) &= \text{oper2use}(o) \\ \text{expr2use}(o_1 + o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(o_1 - o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(o_1 * o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(-o) &= \text{oper2use}(o) \\ \text{expr2use}(o_1 \wedge o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(\neg o) &= \text{oper2use}(o) \\ \text{expr2use}(o_1 \vee o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(o_1 = o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(o_1 \neq o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(o_1 < o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \\ \text{expr2use}(o_1 \neq o_2) &= \text{oper2use}(o_1) \cup \text{oper2use}(o_2) \end{aligned}$$

Definition 7.31 defines a function `instr2use` which computes the set of "use" variables of an instruction.

**Definition 7.31.**

$$\begin{aligned} \text{instr2use} &: \text{Instruction} \rightarrow \text{UseSet} \\ \text{instr2use}(v:=e) &= \text{expr2use}(e) \\ \text{instr2use}(a[i]:=e) &= \{a\} \cup \text{expr2use}(e) \\ \text{instr2use}(a[v]:=e) &= \{a, v\} \cup \text{expr2use}(e) \\ \text{instr2use}(\text{printi}(e)) &= \text{expr2use}(e) \\ \text{instr2use}(\text{branch}(e, dst)) &= \text{expr2use}(e) \\ \text{instr2use}(\text{goto}(dst)) &= \{\} \\ \text{instr2use}(\text{exit}) &= \{\} \end{aligned}$$

Definition 7.32 defines a function `instr2def` which computes the set of "use" variables of an instruction.

**Definition 7.32.**

$\text{instr2def} : \text{Expression} \rightarrow \text{UseSet}$	
$\text{instr2def}(v:=e)$	$= \{v\}$
$\text{instr2def}(a[i]:=e)$	$= \{a\}$
$\text{instr2def}(a[v]:=e)$	$= \{a\}$
$\text{instr2def}(\text{printi}(e))$	$= \{\}$
$\text{instr2def}(\text{branch}(e, dst))$	$= \{\}$
$\text{instr2def}(\text{goto}(dst))$	$= \{\}$
$\text{instr2def}(\text{exit})$	$= \{\}$

Definition 7.33 defines a predicate `confvarsets` on an IL program  $P$ , an LA result  $\mathcal{A}_{LA}$ , a CFG edge  $(pc, succpc)$ , and two states,  $s$  and  $s'$ . The `confvarsets` makes a context assumption that  $s$  and  $s'$  are results of two partial executions of the source and the target program of a DAE optimization which are of the same length, and that the flows of control of those executions has made transitions along the CFG edge  $(pc, pc')$ . Then, the predicate checks the following:

1. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $succpc$  must be consistent with the operational semantics of the  $pc$ -th and  $succpc$ -th instructions of the program  $P$ .
2. The states  $s$  and  $s'$  must conform with the "in" set of variables declared by  $\mathcal{A}_{LA}$  for the node  $succpc$ .

**Definition 7.33.**

$\text{confvarsets} : \text{Program} \times \text{VarSetsEnv} \times \text{InstructionNr} \times \text{InstructionNr}$	$\times \text{State} \times \text{State} \rightarrow \text{Bool}$
$\text{confvarsets}(((vds, instrs), I), \mathcal{A}_{LA}, pc, succpc, s, s') =$	
$\exists use\ def\ in\ out\ use'\ def'\ in'\ out'.$	
$\mathcal{A}_{LA}(pc) = \text{Some}(use, def, in, out) \wedge$	
$\mathcal{A}_{LA}(succpc) = \text{Some}(use', def', in', out') \wedge$	
$use' = \text{instr2use}(instrs!succpc) \wedge$	
$def' = \text{instr2def}(instrs!succpc) \wedge$	
$in' = use' \cup (out' - def') \wedge$	
$in' \subseteq out \wedge$	
$\forall v \in in'. s(v) = s'(v)$	

In Section 6.4.2, we explained that computing the successor configuration by the function `exec'` can result in switching of the mode of execution into the emulation mode only, if the flow of control is within a block at a block position which is not the last one in that block and executing an instruction at a program point which is allocated to that block position results in an exception. In other words, if the flow of control leaves a block by transferring to a block position which is the first one in another block, then this transfer always results in switching into one of three modes: either the normal mode or the exception mode or the exit mode.

As we know what syntactic forms the augmented configurations have in those modes, we can define our bisimulation relation  $\mathcal{R}_{DAE}$  as a union of three disjoint sets which are defined for respective modes of execution as functions of the source and the target programs, a CFGB declaration, and an LA result, such that each of those sets fulfills the following properties:

- If the set is defined for a particular mode of execution, then it holds for each configuration pair  $(\sigma''_S, \sigma''_T)$  in this set that  $\sigma''_S$  and  $\sigma''_T$  have the syntactic forms which comply with that mode of execution, cf. Section 6.4.2 for the syntactic forms.

- For each configuration pair  $(\sigma_S'', \sigma_T'')$  in the set, it holds that all components except their state components are equal.
- For each configuration pair  $(\sigma_S'', \sigma_T'')$  in the set, it holds that  $\sigma_S''$  and  $\sigma_T''$  are results of partial executions of IL<sup>''</sup> programs  $(S, B)$  and  $(T, B)$  by the machine  $M''$  and that these executions of the same length.
- If the set is build for the normal mode of execution, then it additionally holds for each pair  $(\sigma_S'', \sigma_T'')$  in this set that the state components of  $\sigma_S''$  and  $\sigma_T''$  conform with the LA result  $\mathcal{A}_{LA}$ .

In the following, we specify these three sets by giving the definitions of functions which compute the subsets for respective modes of executions.

Definition 7.34 defines a function `bisimrel_DAE_normblk` which computes a subset of the bisimulation relation  $\mathcal{R}_{DAE}$  which is defined for the normal mode of execution and w.r.t. two IL<sup>''</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$ .

**Definition 7.34.**

```

bisimrel_DAE_normblk : Program × Program × BlckPosEnv × VarSetsEnv
                        → BisimulationRelation

bisimrel_DAE_normblk(S, T, B, ALA) =
  let
    (pid0, BP, BB, succBP, predB) = B
  in
    { (σS'', σT'') | ∃ bs ss pid bid pc predbid b s' bt set.
      σS'' = (ss, bs, ss, NORMBLCK(pid, bid, 1, 0, pc), predbid, (NT, ABok, pc, b, s)) ∧
      σT'' = (ss, bs, ss, NORMBLCK(pid, bid, 1, 0, pc), predbid, (NT, ABok, pc, b, s')) ∧
      M''((S, B), bs) = σS'' ∧
      M''((T, B), bs) = σT'' ∧
      BB(bid) = Some(pid) ∧
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      predB(pid) = Some(set) ∧
      (predbid, bt) ∈ set ∧
      BB(predbid) = Some(predpid) ∧
      BP(predpid) = Some(predpid, predbid, 1, 0, predpc) ∧
      confbuffer(bt, b) ∧
      confvarsets(S, ALA, predpc, pc, s, s') }

  end

```

Definition 7.35 defines a function `bisimrel_DAE_excblk` which computes a subset of the bisimulation relation  $\mathcal{R}_{DAE}$  which is defined for the exception mode of execution and w.r.t. two IL<sup>''</sup> programs,  $(S, B)$  and  $(T, B)$ .

**Definition 7.35.**

```

bisimrel_DAE_excblk : Program × Program × BlckPosEnv → BisimulationRelation

bisimrel_DAE_excblk(S, T, B) =
  { (σS'', σT'') | ∃ bs ss pc predbid n s s'.
    σS'' = (ss, bs, ss, EXCBLCK, predbid, (NT, AB, pc, FLUSH(n), s)) ∧
    σT'' = (ss, bs, ss, EXCBLCK, predbid, (NT, AB, pc, FLUSH(n), s')) ∧
    M''((S, B), bs) = σS'' ∧
    M''((T, B), bs) = σT'' }

```

Definition 7.36 defines a function `bisimrel_DAE_exitblk` computes a subset of the bisimulation relation  $\mathcal{R}_{\text{DAE}}$  which is defined for the exit mode of execution and w.r.t. two IL<sup>''</sup> programs,  $(S, B)$  and  $(T, B)$ .

**Definition 7.36.**

$$\begin{aligned} \text{bisimrel\_DAE\_exitblk} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \rightarrow \mathbf{BisimulationRelation} \\ \text{bisimrel\_DAE\_exitblk}(S, T, B) &= \\ &\{(\sigma''_S, \sigma''_T) \mid \exists bs \ ss \ pc \ predbid \ n \ s \ s'. \\ &\quad \sigma''_S = (ss, bs, ss, \text{EXITBLCK}, predbid, (T, \text{AB}_{\text{ok}}, pc, \text{FLUSH}(n), s)) \wedge \\ &\quad \sigma''_T = (ss, bs, ss, \text{EXITBLCK}, predbid, (T, \text{AB}_{\text{ok}}, pc, \text{FLUSH}(n), s')) \wedge \\ &\quad \mathbf{M}''((S, B), bs) = \sigma''_S \wedge \\ &\quad \mathbf{M}''((T, B), bs) = \sigma''_T \} \end{aligned}$$

Definition 7.37 defines a function `bisimrel_DAE` which computes the bisimulation relation  $\mathcal{R}_{\text{DAE}}$  for two IL<sup>''</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$ .

**Definition 7.37.**

$$\begin{aligned} \text{bisimrel}_{\text{DAE}} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \times \mathbf{VarSetsEnv} \rightarrow \mathbf{BisimulationRelation} \\ \text{bisimrel}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA}) &= \text{bisimrel\_DAE\_normblk}(S, T, B, \mathcal{A}_{LA}) \cup \\ &\quad \text{bisimrel\_DAE\_excblk}(S, T, B) \cup \\ &\quad \text{bisimrel\_DAE\_exitblk}(S, T, B) \end{aligned}$$

### 7.2.3 Optimization correctness criterion for the DAE optimization

This section presents the formalization of a translation relation predicate  $\text{TCC}_{\text{DAE}}$  on two IL<sup>''</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$ . The parameters of  $\text{TCC}_{\text{DAE}}$  denote a source and a target programs, a concrete DAE optimization, and a result of the liveness analysis that was performed by the compiler prior to that optimization, respectively, and, informally,  $\text{TCC}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA})$  holds true iff  $(T, B)$  is a correct DAE optimization of  $(S, B)$  w.r.t.  $\mathcal{A}_{LA}$ . In the following, we call the  $\text{TCC}_{\text{DAE}}$  predicate an *optimization correctness criterion for the DAE optimization*. In our implementation, the  $\text{TCC}_{\text{DAE}}$  predicate is an instance of the optimization correctness criterion  $\text{TCC}_O$  that was described in the overview of Layer 5 in Section 3.6

The rest of this section consists of the following four parts:

- The first part gives formation rules for the sets associated with the definition of the optimization correctness criterion  $\text{TCC}_{\text{DAE}}$ .
- The second part formalizes the notion of array-index-out-of-bounds exception safety for expression by giving the definition of a predicate `ABexc_safe_expr` on IL program  $P$  and an expression  $e$ . `ABexc_safe_expr`( $P, e$ ) holds true iff  $e$  is array-index-out-of-bounds exception safe (ABexc-safe) in  $P$ , i.e. it can be guaranteed that its evaluation during execution of  $P$  never results in raising the array-index-out-of-bounds exception.
- The third part presents the definition of a function `dae_transrel_instr` which computes a DAE optimization relation over instruction pairs  $(instr, instr')$  for two programs which are the source and the target of a DAE optimization and a result of the LA which was performed by the compiler prior to that optimization. The definition of `dae_transrel_instr` uses the definition of `ABexc_safe_expr`.
- The last part this section presents the definition of the optimization correctness criterion  $\text{TCC}_{\text{DAE}}$ .

We begin the presentation by introducing a set of translation relations over instruction pairs **InstrTransRel\_DAE**. Definition 7.38 gives formation rule for this set.

**Definition 7.38.**

$$\mathbf{InstrTransRel\_DAE} = \mathcal{P}(\mathbf{Instruction} \times \mathbf{Instruction})$$

Definition 7.39 defines a function **ABexc\_safe\_oper** which checks if an operand  $o$  is ABexc-safe in an IL program  $P$ . An operand  $o$  is ABexc-safe in a program  $((vds, instrs), I)$  iff it fulfills one of the following requirements:

1.  $o$  is an integer constant or a boolean constant.
2.  $o$  is a variable.
3.  $o$  is an indexed operator  $a[i]$  with a constant index  $i$  whose value is less than the length of  $a$  which is declared in the variable declaration  $vds$ .

**Definition 7.39.**

$$\begin{aligned} \mathbf{ABexc\_safe\_oper} : \mathbf{Program} \times \mathbf{Operand} &\rightarrow \mathbf{Bool} \\ \mathbf{ABexc\_safe\_oper}(((vds, instrs), I), i) &= \mathbf{True} \\ \mathbf{ABexc\_safe\_oper}(((vds, instrs), I), b) &= \mathbf{True} \\ \mathbf{ABexc\_safe\_oper}(((vds, instrs), I), v) &= \mathbf{True} \\ \mathbf{ABexc\_safe\_oper}(((vds, instrs), I), a[i]) &= \exists k n. k < \mathbf{length}(vds) \wedge vds!k = (a, \mathbf{barray}(n)) \wedge i < n \\ &\vee \\ &\exists k n. k < \mathbf{length}(vds) \wedge vds!k = (a, \mathbf{iarray}(n)) \wedge i < n \\ \mathbf{ABexc\_safe\_oper}(((vds, instrs), I), a[v]) &= \mathbf{False} \end{aligned}$$

Definition 7.40 defines a function **ABexc\_safe\_expr** which checks if an expression  $e$  is ABexc-safe in an IL program  $P$ . An expression  $e$  is ABexc-safe in a program  $P$  iff all operands in  $e$  are ABexc-safe in  $P$ .

**Definition 7.40.**

$$\begin{aligned} \mathbf{ABexc\_safe\_expr} : \mathbf{Program} \times \mathbf{Expression} &\rightarrow \mathbf{Bool} \\ \mathbf{ABexc\_safe\_expr}(P, o) &= \mathbf{ABexc\_safe\_oper}(P, o) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 + o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 - o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 * o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, -o) &= \mathbf{ABexc\_safe\_oper}(P, o) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 \wedge o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, \neg o) &= \mathbf{ABexc\_safe\_oper}(P, o) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 \vee o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 = o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 \neq o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 < o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \\ \mathbf{ABexc\_safe\_expr}(P, o_1 \leq o_2) &= \mathbf{ABexc\_safe\_oper}(P, o_1) \wedge \mathbf{ABexc\_safe\_oper}(P, o_2) \end{aligned}$$

Now, we present the third part of the formalization of  $\mathbf{TCC}_{\mathbf{DAE}}$ , the definition of a function **dae\_transrel\_instr** which computes a translation relation over instruction pairs for a block position in a CFGB declaration  $B$  w.r.t. an IL" program  $(P, B)$  and a LA result  $\mathcal{A}_{LA}$ . The result of this computation is a relation in the set **InstrTransRel\_DAE**. The function **dae\_transrel\_instr** computes a relation which is defined as a union of nine disjoint sets which arise from nine syntactic

optimization patterns which possible for a pair of corresponding instructions in the source and the target programs of a DAE optimization. These sets are computed by respective auxiliary functions which are presented below in Definitions 7.41, 7.42, 7.43, 7.44, 7.45, 7.46, 7.47, 7.48, 7.49, and 7.50. The auxiliary functions have the same parameters as the function `dae_transrel_instr` and make the following context assumptions about these parameter:

1. The compiler performed the LA on an IL program  $S$ . The result of this analysis is  $\mathcal{A}_{LA}$ .
2. The compiler performed the LA optimization on  $S$ . The result of this optimization is a target program  $T$ .
3. The compiler generated a CFGB declaration  $B$  which is identical for both  $S$  and  $T$ , and all blocks in the CFGB declared by  $B$  have the length equal one.
4. In both  $(S, B)$  and  $(T, B)$ , there exists a program point  $pc$  which is allocated to a block position  $pid$  which is included in a block  $bid$ .

Then, an auxiliary function computes a subset of the DAE optimization relation

**InstrTransRel\_DAE** such that each pair  $(instr, instr')$  in this subset consists of two instructions  $instr$  and  $instr'$  which are the  $pc$ -th instruction of  $S$  and the  $pc$ -th instruction of  $T$ , respectively, and they satisfy a particular optimization pattern which is specified by the function.

Definition 7.41 defines a function `dae_transrel_instr_case1` which computes a subset of a DAE optimization relation for two IL" programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation `dae_transrel_instr_case1` $(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of two assignments which are equal and have the syntactic form  $v:=e$ .
2. The LA result  $\mathcal{A}_{LA}$  declares that the variable  $v$  is live out at the CFG node  $pc$ .
3. The successor program point of  $pc$  is  $pc + 1$ .
4. The program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$  and has the length equal one.
5. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $pc + 1$  are consistent with the operational semantics of the assignment  $v:=e$ .

**Definition 7.41.**

```

dae_transrel_instr_case1 : Program × Program × BlkPosEnv × VarSetsEnv
                          × BlkId × BlkPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case1(S, T, B, ALA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(v:=e, v:=e) | ∃ bid' pid' pc' set use def in out use' def' in' out'.
      instrs!pc = v:=e ∧
      instrs'!pc = v:=e ∧
      ALA(pc) = Some(use, def, in, out) ∧
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
      ALA(pc') = Some(use', def', in', out') ∧
      use' = instr2use(instrs!pc') ∧
      def' = instr2def(instrs!pc') ∧
      in' = use' ∪ (out' - def') ∧
      in' ⊆ out ∧
      v ∈ out }
  end

```

Definition 7.42 defines a function `dae_transrel_instr_case2` which computes a subset of a DAE optimization relation for two IL<sup>n</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation `dae_transrel_instr_case2` $(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of an assignment and a goto instruction which emulates a nop instruction, i.e. tuple has the syntactic form  $(v:=e, goto pc)$ .
2. The LA result  $\mathcal{A}_{LA}$  declares that the variable  $v$  is not live out at the CFG node  $pc$ .
3. The expression  $e$  in the assignment is ABexc-safe in the IL program  $S$ .
4. The successor program point of  $pc$  is  $pc + 1$ .
5. The program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$  and has the length equal one.
6. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $pc + 1$  are consistent with the operational semantics of the assignment  $v:=e$ .

**Definition 7.42.**

```

dae_transrel_instr_case2 : Program × Program × BlckPosEnv × VarSetsEnv
  × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case2(S, T, B, ALA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(v:=e, goto(pc')) | ∃ bid' pid' pc' set use def in out use' def' in' out'.
      instrs!pc = v:=e ∧
      instrs'!pc = goto(pc') ∧
      ALA(pc) = Some(use, def, in, out) ∧
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
      ALA(pc') = Some(use', def', in', out') ∧
      use' = instr2use(instrs!pc') ∧
      def' = instr2def(instrs!pc') ∧
      in' = use' ∪ (out' - def') ∧
      in' ⊆ out ∧
      v ∉ out ∧
      ABexc_safe_expr(S, e) }
  end

```

Definition 7.43 defines a function `dae_transrel_instr_case3` which computes a subset of a DAE optimization relation for two IL<sup>9</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation `dae_transrel_instr_case3` $(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of two assignments, i.e. the pair has the syntactic form  $(v:=e, v:=e)$ .
2. The LA result  $\mathcal{A}_{LA}$  declares that the variable  $v$  is not live out at the CFG node  $pc$ .
3. The expression  $e$  in the assignment is ABexc-safe in the IL program  $S$ .
4. The successor program point of  $pc$  is  $pc + 1$ .
5. The program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$  and has the length equal one.
6. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $pc + 1$  are consistent with the operational semantics of the assignment  $v:=e$ .

The subset of the DAE optimization, which is computed by the function `dae_transrel_instr_case3` is actually not necessary in our framework for the DAE optimizations since but it enables the compiler to decide freely if it replaces an assignment to a variable that is not live out or not. In the following, we explain the reason why we formalized this function anyway.

The reader should recall that our compiler performs a chain of five optimizations: CF, DAE, NI, RAI, and RAE and that the RAI optimization replaces nop instructions by assignments to variables

which are not live out. Interestingly, the RAI optimization<sup>2</sup> has a reverse effect comparing to the DAE optimization and therefore the optimization correctness criterion  $TCC_{DAE}$  can be reused in an optimization correctness criterion for RAI optimizations if one treats the target program of the RAI optimization as a source program of the DAE optimization and the source program of the RAI optimization as a target program of the DAE optimization. However, to able to apply our framework also for RAI optimizations, we have to adapt our formalization of a translation relation over instruction pairs  $\mathbf{InstrTransRel\_DAE}$  as follows. Let us consider two IL programs  $S$  and  $T$  such that  $T$  is a RAI optimization of  $S$ . If there already exists a program point  $pc$  such that the  $pc$ -th instruction of the program  $S$  is an assignment to a variable which is not live out, the RAI optimization replaces a nop instruction at another program point  $pc'$  by another assignment to a variable which is not live out, and we want to verify that optimization as having effect which is reverse to the DAE optimization, then we are obliged to prove that  $S$  is a correct DAE optimization of  $T$  where the compiler freely decided to replace the assignment at the  $pc'$ -th program point by the nop instruction and to leave the  $pc$ -th instruction unchanged. To be able to deal with such optimizations, we augmented the translation relation  $\mathbf{InstrTransRel\_DAE}$  by a subset which is computed by the function `dae_transrel_instr_case3`.

**Definition 7.43.**

```

dae_transrel_instr_case3 : Program × Program × BlckPosEnv × VarSetsEnv
                          × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case3(S, T, B,  $\mathcal{A}_{LA}$ , bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(v:=e, v:=e) | ∃ bid' pid' pc' set use def in out use' def' in' out'.
      instrs!pc = v:=e ∧
      instrs'!pc = v:=e ∧
       $\mathcal{A}_{LA}(pc) = \text{Some}(use, def, in, out) \wedge$ 
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
       $\mathcal{A}_{LA}(pc') = \text{Some}(use', def', in', out') \wedge$ 
      use' = instr2use(instrs!pc') ∧
      def' = instr2def(instrs!pc') ∧
      in' = use' ∪ (out' - def') ∧
      in' ⊆ out ∧
      v ∉ out ∧
      ABexc_safe_expr(S, e) }
  end

```

Definition 7.44 defines a function `dae_transrel_instr_case4` which computes a subset of a DAE optimization relation for two IL<sup>2</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair

<sup>2</sup> The RAI should actually be called the RAI transformation because it does not makes the program more efficient but merely enables other transformation.

of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation  $dae\_transrel\_instr\_case4(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of two assignments to indexed l-values with an integer constant as index, i.e. tuple has the syntactic form  $(a[i]:=e, a[i]:=e)$ .
2. The successor program point of  $pc$  is  $pc + 1$ .
3. The program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$  and has the length equal one.
4. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $pc + 1$  are consistent with the operational semantics of the assignment  $a[i]:=e$ .

**Definition 7.44.**

```

dae_transrel_instr_case4 : Program × Program × BlckPosEnv × VarSetsEnv
                        × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case4(S, T, B,  $\mathcal{A}_{LA}$ , bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(a[i]:=e, a[i]:=e) | ∃ bid' pid' pc' set use def in out use' def' in' out'.
      instrs!pc = a[i]:=e ∧
      instrs'!pc = a[i]:=e ∧
       $\mathcal{A}_{LA}(pc) = \text{Some}(use, def, in, out) \wedge$ 
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
       $\mathcal{A}_{LA}(pc') = \text{Some}(use', def', in', out') \wedge$ 
      use' = instr2use(instrs!dst) ∧
      def' = instr2def(instrs!dst) ∧
      in' = use' ∪ (out' - def') ∧
      in' ⊆ out }
  end

```

Definition 7.45 defines a function  $dae\_transrel\_instr\_case5$  which computes a subset of a DAE optimization relation for two IL<sup>9</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation  $dae\_transrel\_instr\_case5(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of two assignments to indexed l-values with a variable index, i.e. tuple has the syntactic form  $(a[v]:=e, a[v]:=e)$ .
2. The successor program point of  $pc$  is  $pc + 1$ .
3. The program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$  and has the length equal one.
4. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $pc + 1$  are consistent with the operational semantics of the assignment  $a[v]:=e$ .

**Definition 7.45.**

```

dae_transrel_instr_case5 : Program × Program × BlckPosEnv × VarSetsEnv
    × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case5(S, T, B, ALA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    { (a[v]:=e, a[v]:=e) | ∃ bid' pid' pc' set use def in out use' def' in' out'.
      instrs!pc = a[v]:=e ∧
      instrs'!pc = a[v]:=e ∧
      ALA(pc) = Some(use, def, in, out) ∧
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
      ALA(pc') = Some(use', def', in', out') ∧
      use' = instr2use(instrs!dst) ∧
      def' = instr2def(instrs!dst) ∧
      in' = use' ∪ (out' - def') ∧
      in' ⊆ out }
  end

```

Definition 7.46 defines a function `dae_transrel_instr_case6` which computes a subset of a DAE optimization relation for two IL<sup>2</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation `dae_transrel_instr_case6` $(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of `printi` instructions, i.e. tuple has the syntactic form  $(\mathbf{printi}(e), \mathbf{printi}(e))$ .
2. The successor program point of  $pc$  is  $pc + 1$ .
3. The program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$  and has the length equal one.
4. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $pc + 1$  are consistent with the operational semantics of the `printi`  $\mathbf{printi}(e)$ .

**Definition 7.46.**

```

dae_transrel_instr_case6 : Program × Program × BlckPosEnv × VarSetsEnv
  × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case6(S, T, B, ALA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    { (printi(e), printi(e)) | ∃ bid' pid' pc' set use def in out use' def' in' out'.
      instrs!pc = printi(e) ∧
      instrs'!pc = printi(e) ∧
      ALA(pc) = Some(use, def, in, out) ∧
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set') ∧
      (bid, 0TYPE) ∈ set' ∧
      ALA(pc') = Some(use', def', in', out') ∧
      use' = instr2use(instrs!pc') ∧
      def' = instr2def(instrs!pc') ∧
      in' = use' ∪ (out' - def') ∧
      in' ⊆ out }
  end

```

Definition 7.47 defines a function `dae_transrel_instr_case7` which computes a subset of a DAE optimization relation for two IL<sup>''</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation `dae_transrel_instr_case7` $(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of two branch instructions, i.e. tuple has the syntactic form  $(\mathbf{branch}(e, dst), \mathbf{branch}(e, dst))$ .
2. The successor program points of  $pc$  are  $pc + 1$  and  $dst$ .
3. The program point  $pc$  is allocated to the block position  $pid$ . The block position  $pid$  is included by the block  $bid$  of the length equal one.
4. The program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$  and has the length equal one.
5. The program point  $dst$  is allocated to a block position  $pid''$  which is included by a block  $bid''$  which is the successor of  $bid$  and has the length equal one.
6. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$ ,  $pc + 1$ , and  $dst$  are consistent with the operational semantics of the `printi printi(e)`.

**Definition 7.47.**

```

dae_transrel_instr_case7 : Program × Program × BlckPosEnv × VarSetsEnv
                          × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case7(S, T, B, ALA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(branch(e, dst), branch(e, dst)) |
     ∃ bid' pid' bid'' pid'' set' set'' use def in out use' def' in' out' use'' def'' in'' out''.
     instrs!pc = branch(e, dst) ∧
     instrs'!pc = branch(e, dst) ∧
     ALA(pc) = Some(use, def, in, out) ∧
     pc' = Suc(pc) ∧
     succBP(pid, pc') = Some(pid') ∧
     BP(pid') = Some(pid', bid', 1, 0, pc') ∧
     BB(bid') = Some(pid') ∧
     predB(pid') = Some(set') ∧
     (bid, FTYPE) ∈ set' ∧
     ALA(pc') = Some(use', def', in', out') ∧
     use' = instr2use(instrs!pc') ∧
     def' = instr2def(instrs!pc') ∧
     in' = use' ∪ (out' - def') ∧
     in' ⊆ out ∧
     succBP(pid, dst) = Some(pid'') ∧
     BP(pid'') = Some(pid'', bid'', 1, 0, dst) ∧
     BB(bid'') = Some(pid'') ∧
     predB(pid'') = Some(set'') ∧
     (bid, FTYPE) ∈ set'' ∧
     ALA(dst) = Some(use'', def'', in'', out'') ∧
     use'' = instr2use(instrs!dst) ∧
     def'' = instr2def(instrs!dst) ∧
     in'' = use'' ∪ (out'' - def'') ∧
     in'' ⊆ out }
  end

```

Definition 7.48 defines a function `dae_transrel_instr_case8` which computes a subset of a DAE optimization relation for two IL<sup>2</sup> programs,  $(S, B)$  and  $(T, B)$ , and a LA result  $\mathcal{A}_{LA}$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$ . The following holds for the relation `dae_transrel_instr_case8` $(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of two branch instructions, i.e. tuple has the syntactic form  $(goto(dst), goto(dst))$ .
2. The successor program point of  $pc$  is  $dst$ .
3. The program point  $pc$  is allocated to the block position  $pid$ . The block position  $pid$  is included by the block  $bid$  of the length equal one.
4. The program point  $dst$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which is the successor of  $bid$ .
5. The variable sets declared by  $\mathcal{A}_{LA}$  for the program points  $pc$  and  $dst$  are consistent with the operational semantics of the goto instruction `goto(e)`.

**Definition 7.48.**

```

dae_transrel_instr_case8 : Program × Program × BlckPosEnv × VarSetsEnv
                          × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case8(S, T, B, ALA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(goto(dst), goto(dst)) | ∃ bid', pid' set use def in out use' def' in' out'.
      instrs!pc = goto(dst) ∧
      instrs'!pc = goto(dst) ∧
      ALA(pc) = Some(use, def, in, out) ∧
      succBP(pid, dst) = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, dst) ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
      ALA(dst) = Some(use', def', in', out') ∧
      use' = instr2use(instrs!dst) ∧
      def' = instr2def(instrs!dst) ∧
      in' = use' ∪ (out' - def') ∧
      in' ⊆ out }
  end

```

Definition 7.49 defines a function `dae_transrel_instr_case9` which computes a subset of a DAE optimization relation for two IL<sup>n</sup> programs,  $(S, B)$  and  $(T, B)$  w.r.t. a pair of block positions in  $(S, B)$  and  $(T, B)$  described by the tuple  $(bid, pid, pc)$  consisting of a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `dae_transrel_instr_case9` $(S, T, B, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of two branch instructions, i.e. tuple has the syntactic form  $(\mathbf{exit}, \mathbf{exit})$ .
2. The program point  $pc$  has no successor program points.
3. The program point  $pc$  is allocated to the block position  $pid$ . The block position  $pid$  is included by the block  $bid$  of the length equal one. The block  $bid$  has no successors.

**Definition 7.49.**

```

dae_transrel_instr_case9 : Program × Program × BlckPosEnv
                          × BlckId × BlckPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr_case9(S, T, B, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(\mathbf{exit}, \mathbf{exit}) | instrs!pc = \mathbf{exit} ∧
      instrs'!pc = \mathbf{exit} ∧
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      BB(bid) = Some(pid) }
  end

```

Definition 7.50 defines a function `dae_transrel_instr` which makes the same context assumptions as the functions in Definitions 7.41, 7.42, 7.43, 7.44, 7.45, 7.46, 7.47, 7.48, and 7.49 and computes a DAE optimization relation over instruction pairs `dae_transrel_instr(S, T, B, ALA, bid, pid, pc)` for two IL programs,  $S$  and  $T$ , a CFGB declaration  $B$ , a LA result  $A_{LA}$ , and three values which describes the position of the instruction pair in IL” programs  $(S, B)$  and  $(T, B)$ : a block  $bid$ , a block position  $pid$ , and a program point  $pc$ , i.e. for each instruction pair  $(instr, instr')$  in the relation, it holds that both  $instr$  and  $instr'$  are  $pc$ -th instructions in  $S$  and  $T$ , respectively.

The relation is defined as a union of neun disjoint sets which are computed by the functions from Definitions 7.41, 7.42, 7.43, 7.44, 7.45, 7.46, 7.47, 7.48, and 7.49, respectively.

**Definition 7.50.**

```

dae_transrel_instr : Program × Program × BlkPosEnv × VarSetsEnv
                    × BlkId × BlkPosId × InstructionNr → InstrTransRel_DAE
dae_transrel_instr(S, T, B, ALA, bid, pid, pc) =
  dae_transrel_instr_case1(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case2(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case3(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case4(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case5(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case6(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case7(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case8(S, T, B, ALA, bid, pid, pc) ∪
  dae_transrel_instr_case9(S, T, B, bid, pid, pc)

```

Finally, with the definitions of the conformance predicate `confvarsets` and the function `dae_transrel_instr` at hand, we can give the definition of the optimization correctness criterion  $TCC_{DAE}$  which is a predicate on two IL programs,  $S$  and  $T$ , a CFGB declaration  $B$ , and a LA result  $A_{LA}$ . The definition of  $TCC_{DAE}$  formalizes what does it mean that  $T$  is a correct DAE optimization of  $S$  w.r.t.  $B$  and  $A_{LA}$ .

In the beginning of the presentation of the definition of the function `dae_transrel_instr`, we explained that this function makes the following context assumptions:

- The compiler performed the LA on a source IL program  $S$ .
- The result of this analysis is  $A_{LA}$ .
- The result of the LA optimization is an IL program  $T$ .
- Additionally, the compiler generates a CFGB declaration  $B$  which is for both  $S$  and  $T$  since the allocation mappings from the program points of those programs to block positions are identical.

The definition of the correctness criterion  $TCC_{DAE}$  makes the same context assumptions.

The definition of  $TCC_{DAE}$  consists of two conjuncts: The first conjunct expresses a well-formedness property of the entry block of  $B$  and the LA result  $A_{LA}$  w.r.t. to the source program  $S$ . The second conjunct expresses that all instruction pairs consisting of instructions of the programs  $S$  and  $T$ , which are at program points included by the same block  $bid$ , are in the optimization relation computed by the function `dae_transrel_instr`.

Definition 7.51 defines a function `TCC_DAE_entry_block` which checks if the first conjunct of the definition of  $TCC_{DAE}$  is holds true. The first conjunct `TCC_DAE_entry_block(S, B, ALA)` holds true iff

1. the entry block of a CFGB declaration  $B$  is well-formed. The entry block of  $B$ ,  $bid_0$ , is well-formed iff it includes the entry block position of  $B$ ,  $pid_0$ , has the length equal one, the 0-th

program point is allocated to  $pid_0$ , and is declared by  $B$  to be among his own predecessor blocks.

2. The variable sets declared by  $\mathcal{A}_{LA}$  for the program point 0 is consistent with the operational semantics of the 0-th instruction of the program  $S$ .

**Definition 7.51.**

```

TCC_DAE_entry_block : Program × BlkPosEnv × VarSetsEnv → Bool
TCC_DAE_entry_block(S, B,  $\mathcal{A}_{LA}$ ) =
  let
    ((vds, instrs), I) = S;
    (pid0, BP, BB, succBP, predB) = B
  in
    ∃ bid0 set.
      BP(pid0) = Some(pid0, bid0, 1, 0, 0) ∧
      predB(pid0) = Some(set) ∧
      (bid0, FTYPE) ∈ set
    ∧
    ∃ use def, in out.
      use = instr2use(instrs!0) ∧
      def = instr2def(instrs!0) ∧
      in = instr2use(instrs!0) ∪ (out - (instr2def(instrs!0)))
       $\mathcal{A}_{LA}(0)$  = Some(use, def, in, out) ∧
      (use ∪ (out - def)) ⊆ out
  end

```

Definition 7.52 defines the predicate  $TCC\_DAE\_normal\_block$  on a source program  $S$ , a target program  $T$ , a CFGB declaration  $B$ , a LA result  $\mathcal{A}_{LA}$ , and a block  $bid$  which checks if the block  $bid$  in the CFGB  $(S, B)$  and the block  $bid$  in the CFGB  $(T, B)$  fulfill an optimization correctness criterion for the block pair  $(bid, bid)$  which is defined as follows:  $TCC\_DAE\_normal\_block(S, T, B, \mathcal{A}_{LA}, bid)$  holds true iff there exist  $pc$ -th program points  $pc$  in  $S$  and  $T$ ; and a block position  $pid$  such that  $pc$  is allocated to  $pid$ ,  $pid$  is included by the block  $bid$ , and the pair  $(instrs!pc, instrs'!pc)$  consisting of the  $pc$ -th instruction of  $S$  and the  $pc$ -th instruction of  $T$ , respectively, are in the DAE optimization relation  $dae\_transrel\_instr(S, T, B, \mathcal{A}_{LA}, bid, pid, pc)$ .

**Definition 7.52.**

```

TCC_DAE_normal_block : Program × Program × BlkPosEnv × VarSetsEnv → Bool
TCC_DAE_normal_block(S, T, B,  $\mathcal{A}_{LA}$ , bid) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    ∃ pid pc.
      BB(bid) = Some(pid) ∧
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      (instrs!pc, instrs'!pc) ∈ dae_transrel_instr(S, T, B,  $\mathcal{A}_{LA}$ , bid, pid, pc)
  end

```

Definition 7.53 defines our optimization correctness criterion for the DAE optimization on a source program  $S$ , a target program  $T$ , a CFGB declaration  $B$ , and a LA result  $\mathcal{A}_{LA}$  which checks

if  $T$  is a correct DAE optimization of  $S$  w.r.t.  $B$  and  $\mathcal{A}_{LA}$ . According to the definition of  $\text{TCC}_{\text{DAE}}$ , an IL program  $T$  is a correct DAE optimization of  $S$  iff

1. the entry blocks of IL" programs  $(S, B)$  and  $(T, B)$  fulfill the criterion  $\text{TCC\_DAE\_entry\_block}$  w.r.t. the LA result  $\mathcal{A}_{LA}$ , and
2. all pairs of corresponding blocks in  $(S, B)$  and  $(T, B)$  fulfill the criterion  $\text{TCC\_DAE\_normal\_block}$  w.r.t. the LA result  $\mathcal{A}_{LA}$ .

**Definition 7.53.**

```

TCCDAE : Program × Program × BkPosEnv × VarSetsEnv → Bool
TCCDAE(S, T, B, ALA) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    TCC_DAE_entry_block(S, B, ALA)
    ∧
    ∀ bid. TCC_DAE_normal_block(S, T, B, ALA, bid)
  end

```

**7.2.4 Verification of the optimization correctness criterion  $\text{TCC}_{\text{DAE}}$**

This section presents the theorems which we proved in order to verify the specification of the optimization correctness criterion  $\text{TCC}_{\text{DAE}}$  presented in the previous section. The main result in this section is a theorem which can be used directly in translation certificates generated by our compiler.

To verify the specification of the criterion  $\text{TCC}_{\text{DAE}}$ , we proved Theorem 7.54 which is a statement about a source and a target program of a concrete DAE optimization,  $S$  and  $T$ , a program type  $\Phi$ , a CFGB declaration  $B$ , and a result of the LA analysis which was performed on  $S$  prior to that optimization,  $\mathcal{A}_{LA}$ , and it says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B$  is well-formed w.r.t.  $S$  and  $T$ ; and  $S$  and  $T$  fulfill the optimization correctness criterion  $\text{TCC}_{\text{DAE}}$  w.r.t.  $B$  and  $\mathcal{A}_{LA}$ , then  $S$  and  $T$  fulfill the optimization independent translation correctness  $\text{TCC}$  w.r.t. the bisimulation relation  $\text{bisimrel}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA})$ .

**Theorem 7.54.**

$$\begin{aligned}
 & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge \text{TCC}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA}) \\
 & \implies \\
 & \text{TCC}(S, T, \text{bisimrel}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA}))
 \end{aligned}$$

□

Finally, we present the main result in this section, a theorem which is a statement about a source and a target programs of a concrete DAE optimization,  $S$  and  $T$ , a program type  $\Phi$ , a CFGB declaration  $B$ , and a result of the LA which was performed on  $S$  prior to that optimization,  $\mathcal{A}_{LA}$ , which says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B$  is well-formed w.r.t.  $S$  and  $T$ ; and  $S$  and  $T$  fulfill the optimization correctness criterion  $\text{TCC}_{\text{DAE}}$  w.r.t.  $B$  and  $\mathcal{A}_{LA}$ , then  $S$  and  $T$  fulfill the translation correctness predicate  $\text{corrTrans}$ .

**Theorem 7.55.**

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge \text{TCC}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA}) \\ & \implies \\ & \text{corrTrans}(S, T) \end{aligned}$$

*Proof.*

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge \text{TCC}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA}) \\ & \implies \text{[by application of Theorem 7.54]} \\ & \text{TCC}(S, T, \text{bisimrel}_{\text{DAE}}(S, T, B, \mathcal{A}_{LA})) \\ & \implies \text{[by application of the existential introduction rule]} \\ & \exists \mathcal{R}. \text{TCC}(S, T, \mathcal{R}). \\ & \implies \text{[by application of Theorem 6.44]} \\ & \text{corrTrans}(S, T) \end{aligned}$$

□

This theorem is directly applicable in translation certificates which are generated by the compiler for each DAE optimization.

Theorem 7.55 is an instance of the corollary (I) in Section 3.6 and is directly applicable in translation certificates which are generated by the compiler for each DAE optimization, see the end of Section 3.6 for a general application scheme of this theorem.

### 7.3 SVF for NI optimizations

This section presents formalization of an optimization correctness criterion for NI optimizations,  $\text{TCC}_{\text{NI}}$ , and a corresponding optimization correctness theorem which is directly applied in translation certificates generated by our compiler front-end.

In Sections 7.1 and 7.2, we presented formalizations of optimization correctness criteria for the optimizations CF and DAE, respectively. Both optimizations have in common that they are structure preserving, i.e. that they do not modify the sets of edges and nodes of the CFG of the source program. Therefore, the formalizations of the optimization correctness criteria  $\text{TCC}_{\text{CF}}$  and  $\text{TCC}_{\text{DAE}}$  make the following assumptions about the CFG's and the CFGB declarations involved in the optimizations CF and DAE:

- The CFG's of the source and the target programs are identical,
- The CFGB declarations for the source and the target programs are identical,
- The corresponding program points relation between program points of the source and the target programs is defined as identity, i.e. a one-to-one correspondence between program points of the source and the target programs,
- For both the source and the target program, the allocation relation between program points and block positions is a one-to-one correspondence,
- The "includes" relation between blocks and block positions is a one-to-one correspondence,
- The CFGB declaration used to formalize the criteria  $\text{TCC}_{\text{CF}}$  and  $\text{TCC}_{\text{DAE}}$  comprises no nested blocks.

This results in relatively straightforward formalizations of the bisimulation relations over pairs of augmented configurations,  $\mathcal{R}_{CF}$  and  $\mathcal{R}_{DAE}$ , presented in Sections 7.1 and 7.2: As the source and the target CFGB declarations are equal, it holds for all configuration pairs  $(\sigma_S'', \sigma_T'')$  in  $\mathcal{R}_{CF}$  and  $\mathcal{R}_{DAE}$  that corresponding components of  $\sigma_S''$  and  $\sigma_T''$  describing position of the flow of control in a respective CFGB are equal. The sole aggravating factor which has to be taken into account when formalizing the relations  $\mathcal{R}_{CF}$  and  $\mathcal{R}_{DAE}$  is the fact that the optimizations CF and DAE modify instructions and that we have to strengthen the execution invariants expressed by  $\mathcal{R}_{CF}$  and  $\mathcal{R}_{DAE}$  by introducing additional requirements about conformance of the state components with the results of data flow analysis  $\mathcal{A}_{CPA}$  and  $\mathcal{A}_{LA}$ .

In contrast to the optimizations CF and DAE, the NI optimization is a structure modifying transformation which modifies the sets of nodes and edges of the CFG of the source program. As a result, the CFGB declarations for the source and the target programs of the NI optimization are also different and our formalization of the optimization correctness criterion  $TCC_{NI}$  has to take the following determining factors into account:

- The CFG's of the source and the target programs are not equal.
- The CFGB declarations for the source and the target programs are not equal.
- The corresponding program points relation between program points of the source and the target program is a one-to-many relation.
- The allocation relation between program points of the source program and block positions of the CFGB declaration for this program is a one-to-one correspondence.
- The "includes" relation between blocks and block positions of the CFGB declaration for the source program is a one-to-one correspondence.
- The CFGB declaration for the source program declares no nested blocks.
- The allocation relation between program points of the target program and block positions of the CFGB declaration for this program is a one-to-many correspondence.
- The "includes" relation between blocks and block positions of the CFGB declaration for the target program is a one-to-many correspondence.
- The CFGB declaration for the target program declares at least one nested block.
- The NI optimization inserts nop instructions into the source program and does not modify the rest of its instructions in the following sense:
  - An assignment instruction is not modified by the NI optimization,
  - A printi instruction is not modified by the NI optimization,
  - As inserting of nop instruction alters the length of the program, the destination address  $dst$  in a branch instruction  $\mathbf{branch}(e, dst)$  has to be adjusted in order to maintain the semantic of the source program. The expression  $e$  in the branch instruction is not changed.
  - As inserting of nop instruction alters the length of the program, the destination address  $dst$  in a goto instruction  $\mathbf{goto}(dst)$  has to be adjusted in order to maintain the semantic of the source program.
  - An exit instruction is not modified by the NI optimization.

The above determining factors express constraints which are the starting point for the formalization of a bisimulation relation  $\mathcal{R}_{NI}$  which is necessary to formulate the statement of a theorem verifying correctness of the specification of the  $TCC_{NI}$  criterion. The theorem is formulated analogously to Theorems 7.26 and 7.54.

The determining factors of the NI optimization are dual to the determining factors for the optimizations CF and DAE. This implies dual constraints of the bisimulation relation  $\mathcal{R}_{NI}$ :

- It holds for each pair of augmented configurations  $(\sigma_S'', \sigma_T'') \in \mathcal{R}_{NI}$  that all corresponding components of  $\sigma_S''$  and  $\sigma_T''$  which describe position of the flow of control in the CFGB declarations for the source and the target programs are, in general, not equal. This constraint results from

the fact that the optimization NI alters the length of the program and that we have to define two different CFGB declarations to formalize the relation  $\mathcal{R}_{\text{NI}}$ . In these CFGB declarations, both the block sets and the block position sets are different.

- It holds for each pair of augmented configurations  $(\sigma''_S, \sigma''_T) \in \mathcal{R}_{\text{NI}}$  that the state components in  $\sigma''_S$  and  $\sigma''_T$  are equal. This results from the fact that the optimization NI modifies neither expressions nor l-values in the source program instructions. It merely adjusts the destination values in the branch and goto instructions.
- Our compiler front-end does not perform any data flow analyses prior to the optimization NI. This constraint results from the fact that the optimization NI is actually the first intermediate program transformation in a chain of three program transformations which altogether perform the LIH optimization. The function of the optimization NI is to insert a limited number of nop instructions which serve as placeholders for the next program transformation which replaces these instructions by dead assignments, the RAI optimization.
- Our compiler front-end performs only such intermediate NI optimizations, which insert nop instructions between a  $pc$ -th and a  $pc'$ -th instruction where the following holds true:
  - The  $pc'$ -th instruction is the successor of the  $pc$ -th instruction.
  - The  $pc'$ -th instruction is a loop header of the loop being the result of translation of a do-while loop in the original  $\mu\text{C}$  program.

This constraint results from the fact that the optimization NI is the first intermediate program transformation in a chain of three program transformations, which altogether perform the LIH optimization, and that LIH optimizations are only possible on do-while loops, cf. [3].

In order to be able to formalize the bisimulation relation  $\mathcal{R}_{\text{NI}}$  which satisfies the above constraints, we define two different CFGB declarations for the source and the target program of an optimization NI which are as follows. For the purpose of the explanation, we use two CFGB's, which are generated by our compiler front-end for the programs  $IL_2$  and  $IL_3$ , where  $IL_2$  and  $IL_3$  are the source and the target programs of the NI optimization in our example illustrating the work-flow of our compiler front-end in Figure 1.5.

The left side of Figure 7.1 depicts two CFGB's which are declared for the pair of IL programs  $IL_2$  and  $IL_3$  which are the source and the target programs of the NI optimization described in the example illustrating the work-flow of our compiler in Figure 1.5. The programs themselves are depicted in Figure A.3. In Figure 7.1, the left CFGB is declared for the program  $IL_2$  and the right CFGB is declared for the program  $IL_3$ . In the following explanation, we call the CFGB's for the programs  $IL_2$  and  $IL_3$  a source and a target CFGB's, respectively. The optimization NI inserted a nop instruction between the 10-th and the 11-th instructions of the program  $IL_2$ . In the source CFGB, the program points 10 and 11 are allocated to block positions  $p_{10,10}$  and  $p_{11,11}$ , respectively. Inserting the nop instruction resulted in translating the loop header instruction and all instructions below the loop header in the program  $IL_2$ , the 11-th instruction, one program point downwards. In the target program  $IL_3$ , the inserted nop instruction is the 11-th instruction and the loop preheader instruction became the 12-th instruction. Further, the sets of block identifiers and block position identifiers are different for the source and the target CFGB's. In the target CFGB, the program point 11 is allocated to the block position  $p_{11,11}$  and the program point 12 is allocated to the block position  $p_{11,12}$ . As the inserted nop instruction at the program point 11 does not modify the state during execution of the program  $IL_3$ , we merged the block positions  $p_{11,11}$  and  $p_{11,12}$  into one block because execution of instructions in the blocks  $b_{11}$  and  $b_{11}$  in the source and the target CFGB's, respectively, yields equal states, if it starts from equal states. As the nop instruction allocated to the block position  $p_{11,11}$  in the target CFGB is a placeholder for an assignment instruction which will be executed before the flow of control enters the loop whose header is the program point 12 allocated to block position  $p_{11,12}$ , the jumping destination value in

the branch instruction at the program point 27 has to be set to 12. In doing so, we assure that the semantics of the program  $IL_2$  is maintained after the NI optimization. Whenever during execution of the program  $IL_3$  the flow of control to transfer from the program point 27 to a successor program point, the next executed instruction will be the same as it would be in case of making transition from the program point 26 in the program  $IL_2$ . This introduces a complication: In the source program  $IL_2$ , the successor of the block position  $p_{26,26}$  is the block position  $p_{11,11}$  and the successor of the block  $b_{26}$  is the block  $b_{11}$ . However, in the target program  $IL_3$ , the block position  $p_{11,12}$  can not be declared as a successor of  $p_{26,27}$  because it would violate the well-formedness of the target CFGB. As the block position  $p_{26,27}$  is the last one in the block  $b_{26}$  its successor must be a block position which is the first one in a block by which it is included, i.e. must be a block position with the block index equal 0 in the block. Therefore, we have to declare a block  $b_{29}$  which includes an additional block position  $p_{29,12}$  to which the loop header program point is allocated. As the program point 12 already is allocated to the block position  $p_{11,12}$ , this means that we introduce a nested block  $b_{29}$ .

As the graphs of the source and the target CFGB's are not isomorph, we can not formalize a one-to-one correspondence relation over block pairs or over block position pairs, as we did in the SVF for CF optimizations and in the SVF for DAE optimizations, respectively: For this reason, we can not use the same technique to formalize the bisimulation relation  $\mathcal{R}_{NI}$  which we used to formalize the bisimulation relations  $\mathcal{R}_{CF}$  and  $\mathcal{R}_{DAE}$ . To cope with this issue, we introduced into our framework a notion which replaces the notion of corresponding block position, a relation of labeled pairs consisting of corresponding block edges. Each labeled pair  $((bid, bid'), (bid'', bid'''), bt)$  consists of two pairs of block identifiers  $(bid, bid')$  and  $(bid'', bid''')$  and a buffertype  $bt$  which is a label of the pairs. In the following, we call these pairs *corresponding block edge pairs*. For each optimization NI, our compiler front-end generates a set of corresponding block edge pairs *CBEP* and the meaning of a pair

$$((bid, bid'), (bid'', bid'''), bt) \in CBEP$$

is that the blocks  $bid$  and  $bid''$  in the source and the target CFGB's, respectively, are declared as corresponding and if the flows of control transfer during executions of the source and the target CFGB to blocks  $bid'$  and  $bid'''$ , then the output buffers in respective configurations have the forms which conform to the value of the buffertype  $bt$  and there exists a corresponding block edge pair

$$((bid_2, bid'_2), (bid''_2, bid'''_2), bt_2) \in CBEP$$

such that

$$bid' = bid_2 \wedge bid''' = bid''_2$$

*Example 7.56.* As aforementioned, the left side of Figure 7.1 shows the source and the target CFGB's for the programs  $IL_2$  and  $IL_3$ , respectively. The right side of Figure 7.1 shows a set of corresponding block edge pairs *CBEP* which was generated by our compiler front-end for those CFGB's. For instance, if the flows of control transfer along the block position edges  $(p_{26,26}, p_{11,11})$  and  $(p_{26,27}, p_{29,12})$  in the source and the target CFGB's, respectively, then they proceed along the edges  $(p_{11,11}, p_{12,12})$  and  $(p_{29,12}, p_{12,13})$ , respectively. Further, the 26-th and 11-th instructions of the program  $IL_2$  are `BRANCH _tB_2 [11]` and `GOTO [12]`, respectively, the 27-th and 12-th instructions of the program  $IL_3$  are `BRANCH _tB_2 [12]` and `GOTO 13`, respectively. Therefore, the set *CBEP* comprises the following two corresponding block edge pairs:

1.  $((b_{26}, b_{11}), (b_{26}, b_{29}), FTYPE)$
2.  $((b_{11}, b_{12}), (b_{29}, b_{12}), FTYPE)$

◇

The rest of the section is organized as follows. Section 7.3.1 presents the formalization of the relation of corresponding block edge pairs. Section 7.3.2 presents the formalization of the bisimulation relation  $\mathcal{R}_{\text{NI}}$ . Section 7.6.2 presents the formalization of an optimization correctness criterion for NI optimizations  $\text{TCC}_{\text{NI}}$ .

### 7.3.1 Corresponding block edge pairs

This section presents the formalization of a relation over labeled pairs of corresponding block edges. We begin the presentation by listing sets associated with this notion:

- labeled block edge pairs **BlockEdgePair**,
- relations over labeled pairs of corresponding block edges **CorrespBlockEdgePairSet**;

and defining metavariables ranging over these sets:

- *blkedgepair* ranges over labeled pairs over corresponding block edges **BlockEdgePair**,
- *CBEP* ranges over relations over labeled pairs of corresponding block edges **CorrespBlockEdgePairSet**.

Definition 7.57 gives formation rules for the set of relations over labeled pairs of corresponding block edges. The purpose and the meaning of elements in a relation  $CBEP \in \text{CorrespBlockEdgePairSet}$  was explained in the beginning of Section 7.3.

#### Definition 7.57.

$$\begin{aligned} \mathbf{BlockEdgePair} \ni \text{blkedgepair} & \quad := ((bid, bid), (bid, bid), bt) \\ CBEP \in \mathbf{CorrespBlockEdgePairSet} & = \mathcal{P}(\mathbf{BlockEdgePair}) \end{aligned}$$

### 7.3.2 Bisimulation relation for the NI optimization

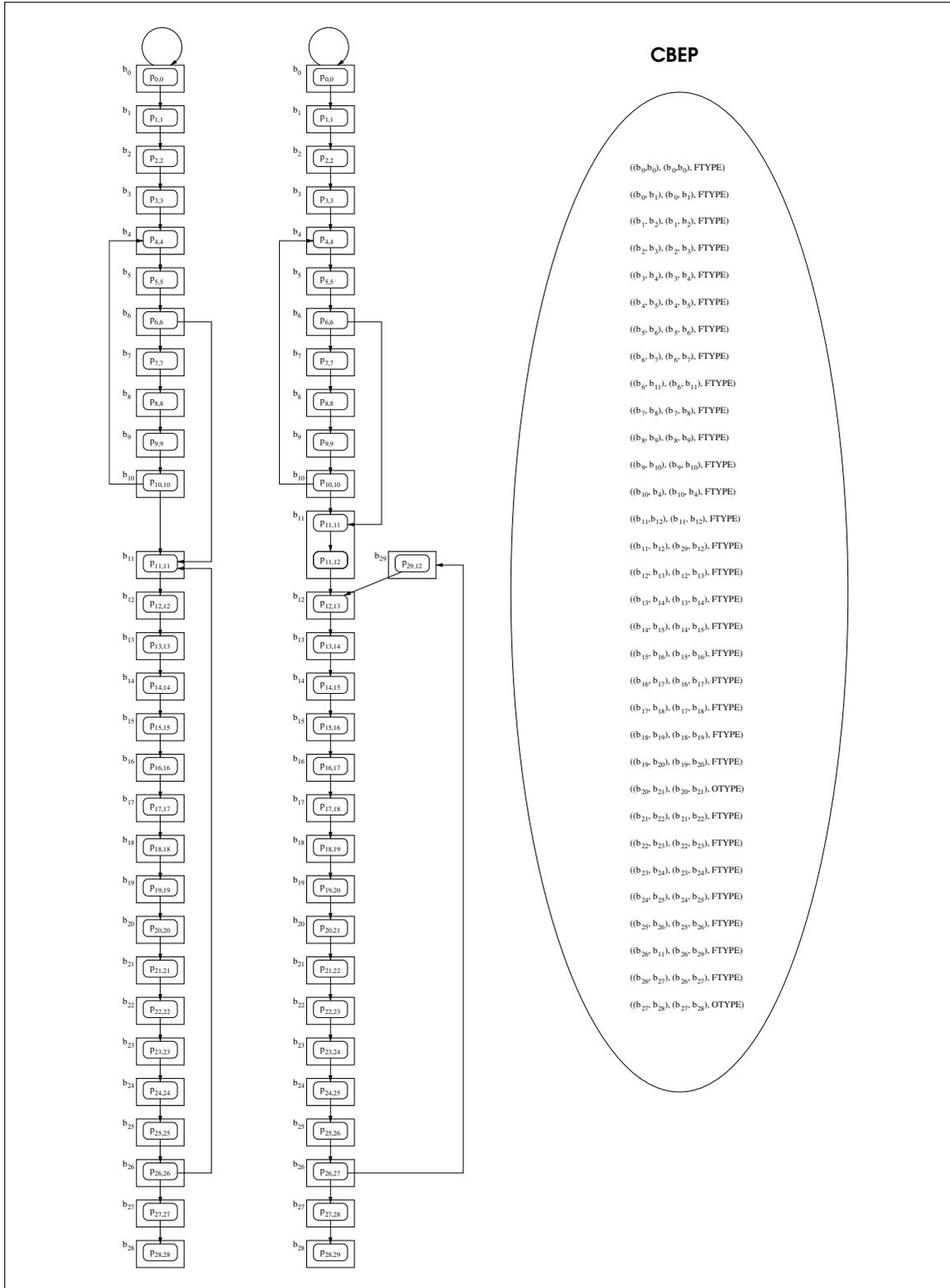
This section presents the definition of a function  $\text{bisimrel}_{\text{NI}}$  which computes a bisimulation relation  $\mathcal{R}_{\text{NI}}$  for two IL' programs, which are the source and the target programs of a NI optimization, a source and a target CFGB declarations, and a relation over corresponding block edge pairs.

Informally, a bisimulation relation  $\mathcal{R}_{\text{NI}}$  can be characterized as follows.

- $\mathcal{R}_{\text{NI}}$  is a function of a NI optimization, which is described by the following values:
  1. a source IL program  $S$ ,
  2. a target IL program  $T$ ,
  3. a source CFGB declaration  $B_S$ ,
  4. a target CFGB declaration  $B_T$ , and
  5. a corresponding block edges relation  $CBEP$ .
- By Definition 6.40,  $\mathcal{R}_{\text{NI}}$  has to be a subset of

$$\mathbf{Configuration''} \times \mathbf{Configuration''}.$$

- As aforementioned in the beginning of Section 7.3, the NI is a structure modifying transformation, i.e. it modifies the sets of nodes and edges of the CFG of the source program and that it does not modify the expressions and l-values of instructions in the source program. For this reason, the CFGB declarations  $B_S$  and  $B_T$  for  $S$  and  $T$ , respectively, can not be equal. This is a consequence of the fact that the sets of block identifiers, block position identifiers, and block position descriptors, which are the starting points for the declarations of  $B_S$  and  $B_T$ , has to be



**Fig. 7.1.** On the left side of the figure, the CFGB declarations for the IL programs  $IL_2$  and  $IL_3$  which are depicted in Figure A.3.  $IL_2$  and  $IL_3$  are the source and the target programs of the NI optimization. On the right side of the figure, the corresponding block edge pairs relation  $CBEP$ .

different. Further, it holds for two IL" programs  $(S, B_S)$  and  $(T, B_T)$  that if these programs are partially executed block-wise and these executions have the same lengths and they produce augmented configurations  $\sigma_S''$  and  $\sigma_T''$ , then it holds the following:

1. All corresponding components of  $\sigma_S''$  and  $\sigma_T''$  counting transfers of the flow of control from block position to block position are, in general, not equal. This is due to the fact the numbers of block positions in  $B_S$  and  $B_T$  are not equal.
2. All corresponding components of  $\sigma_S''$  and  $\sigma_T''$  describing positions of the flow of control in the source and target CFGB declarations are not equal.
3. All corresponding components of  $\sigma_S''$  and  $\sigma_T''$  describing observable and non-observable behaviors of  $(S, B_S)$  and  $(T, B_T)$  are equal. These components are components of the configuration component  $\sigma$  in an augmented configuration  $\sigma'' = (ss, bs, ss, bposstat, predbid, \sigma)$  and they are the following: the termination flag  $tf$ , the array-index-out-of-bounds exception flag  $af$ , the output buffer  $b$ , and the state  $s$ .

$\forall n S B_S T B_T.$

$$\mathbf{M}''((S, B_S), \text{init}(S, B_S), n) = \sigma_S'' \wedge$$

$$\mathbf{M}''((T, B_T), \text{init}(T, B_T), n) = \sigma_T''$$

$\longrightarrow$

$$\exists ss bs bposstat predbid tf af pc b s ss' bs' bposstat' predbid' tf' af' pc' b' s'.$$

$$\sigma_S'' = (ss, bs, ss, bposstat, predbid, (tf, af, pc, b, s)) \wedge$$

$$\sigma_T'' = (ss', bs', ss', bposstat', predbid', (tf', af', pc', b', s')) \wedge$$

$$ss = ss' \vee ss \neq ss' \wedge$$

$$bs = bs' \wedge$$

$$bposstat \neq bposstat' \wedge$$

$$predbid \neq predbid' \wedge$$

$$tf = tf' \wedge$$

$$af = af' \wedge$$

$$pc \neq pc' \wedge$$

$$b = b' \wedge$$

$$s = s'$$

- In Section 6.4.2, we explained that computing the successor configuration  $\text{exec}'((VP, B), \sigma'')$  for an IL" program  $(P, B)$  and an augmented configuration  $\sigma''$  by the function  $\text{exec}'$  can result in switching of the mode of execution into the emulation mode only, if the flow of control is within a block  $bid$  at a block position  $pid$  which is not the last one in  $bid$  and executing an instruction at a program point which is allocated to  $pid$  results in an exception. In other words, if the flow of control leaves a block by transferring to a block position which is the first one in another block, then this transfer can result in switching into one of three modes of execution only: either the normal mode or the exception mode or the exit mode. As we know what syntactic forms do the augmented configurations have in those modes, we can define our bisimulation relation  $\mathcal{R}_{\text{NI}}$  as a union of three sets which are defined for respective modes of execution as a function of the source and the target programs, a source and a target CFGB declarations, and a corresponding block edges relation such that each of those sets fulfills the following properties:
  - If the set is defined for a particular mode of execution, then it holds for each configuration pair  $(\sigma_S'', \sigma_T'')$  in this set that  $\sigma_S''$  and  $\sigma_T''$  have the syntactic forms which comply with that mode of execution, cf. Section 6.4.2 for the syntactic forms.
  - For each configuration pair  $(\sigma_S'', \sigma_T'')$  in the set, it holds that the values of corresponding components of  $\sigma_S''$  and  $\sigma_T''$  are relating to each other as described above.

- Consider two augmented configurations which are results of partial executions of the IL” programs  $(S, B_S)$  and  $(T, B_T)$  which have the same length  $n$ :

$$\begin{aligned} M''((S, B_S), \text{init}(S, B_S), n) &= \sigma''_S \\ M''((T, B_T), \text{init}(T, B_T), n) &= \sigma''_T \end{aligned}$$

In these configurations, the values of block position status  $bposstat$  and  $bposstat'$  denote block positions of the flow of control in the CFGB declarations  $B_S$  and  $B_T$ , respectively. As both programs are executed block-wise and program executions made  $n$  block-wise transitions, both flows of control are at block positions  $pid$  and  $pid'$  which are entry block positions of blocks  $bid$  and  $bid'$ , respectively. If both programs are executed in the normal mode of execution, then we require that execution of  $(S, B_S)$  and  $(T, B_T)$  makes transitions from block to block along block edges which are declared by the corresponding block edges relation  $CBEP$  as corresponding, i.e. we require that there exists a buffertype  $bt$  such that

$$((predbid, bid), (predbid', bid), bt) \in CBEP$$

In the following, we give three definitions of functions which compute three subsets of the bisimulation relation  $\mathcal{R}_{NI}$  for respective modes of executions as described above.

Definition 7.58 defines a function  $\text{bisimrel\_NI\_normblk}$  which computes a subset of bisimulation relation  $\mathcal{R}_{NI}$ ,  $\text{bisimrel\_NI\_normblk}(S, T, B_S, B_T, CBEP)$ , for two IL” programs,  $(S, B)$  and  $(T, B)$ , and a corresponding block edges relation  $CBEP$ . The following holds for each pair of augmented configurations  $(\sigma''_S, \sigma''_T)$  in this subset:

- $\sigma''_S$  and  $\sigma''_T$  are produced by partial executions of  $(S, B)$  and  $(T, B)$  of equal lengths and that these executions were performed in the normal mode of execution.
- During partial executions of  $(S, B)$  and  $(T, B)$ , the flows of control made the same numbers of transitions from block to block.
- Each time partial executions of  $(S, B)$  and  $(T, B)$  made the  $n$ -th transition from block  $predbid$  to block  $bid$  and from block  $predbid'$  to block  $bid'$ , respectively, then their states of output buffers were equal, i.e. were both equal  $b$ .
- Each time partial executions of  $(S, B)$  and  $(T, B)$  made the  $n$ -th transition from block  $predbid$  to block  $bid$  and from block  $predbid'$  to block  $bid'$ , respectively, then the respective flows of control transferred along block edges which are declared by the relation  $CBEP$  as corresponding, there exists buffertype  $bt$  such that

$$((predbid, bid), (predbid', bid), bt) \in CBEP$$

**Definition 7.58.**

```

bisimrel_NI_normblck : Program × Program × BlckPosEnv × BlckPosEnv
                        × CorrespBlckEdgePairSet → BisimulationRelation

bisimrel_NI_normblck(S, T, BS, BT, CBEP) =
  let
    (pid0, BPS, BBS, succBPS, predBS) = S
    (pid'0, BPT, BBT, succBPT, predBT) = T
  in
    { (σ''S, σ''T) | ∃ ss bs pid bid pc predbid b s bt set ss' pid' bid' pc' predbid' set'.
      σ''S = (ss, bs, ss, NORMBLCK(pid, bid, 1, 0, pc), predbid, (NT, ABok, pc, b, s)) ∧
      σ''T = (ss', bs, ss', NORMBLCK(pid', bid', bsize', 0, pc'), predbid', (NT, ABok, pc', b, s)) ∧
      M''((S, BS), bs) = σ''S ∧
      M''((T, BT), bs) = σ''T ∧
      BBS(bid) = Some(pid) ∧
      BPS(pid) = Some(pid, bid, 1, 0, pc) ∧
      predBS(pid) = Some(set) ∧
      (predbid, bt) ∈ set ∧
      BBT(bid') = Some(pid') ∧
      BPT(pid') = Some(pid', bid', bsize', 0, pc') ∧
      predBT(pid') = Some(set') ∧
      (predbid', bt) ∈ set' ∧
      confbuffer(bt, b) ∧
      ((predbid, bid), (predbid', bid'), bt) ∈ CBEP }

  end

```

Definition 7.59 defines a function `bisimrel_NI_exclck` which computes a subset of the bisimulation relation  $\mathcal{R}_{\text{NI}}$ , `bisimrel_NI_exclck(S, T, BS, BT, CBEP)`, such that the following holds for each pair augmented configurations  $(\sigma''_S, \sigma''_T)$  in this subset:

- $\sigma''_S$  and  $\sigma''_T$  are produced by partial executions of  $(S, B)$  and  $(T, B)$  of equal length  $n$ , i.e. the respective flows of control made  $n$  transitions from block to block.
- Producing these configurations resulted in switching into the exception mode of execution.

**Definition 7.59.**

```

bisimrel_NI_exclck : Program × Program × BlckPosEnv × BlckPosEnv
                    → BisimulationRelation

bisimrel_NI_exclck(S, T, BS, BT) =
  { (σ''S, σ''T) | ∃ ss bs predbid pc b n s ss' predbid' pc'.
    σ''S = (ss, bs, ss, EXCBLCK, predbid, (NT, AB, pc, FLUSH(n), s)) ∧
    σ''T = (ss', bs, ss', EXCBLCK, predbid', (NT, AB, pc', FLUSH(n), s)) ∧
    M''((S, BS), bs) = σ''S ∧
    M''((T, BT), bs) = σ''T }

```

Definition 7.60 defines a function `bisimrel_NI_exitck` which computes a subset of the bisimulation relation  $\mathcal{R}_{\text{NI}}$ , `bisimrel_NI_exitck(S, T, BS, BT, CBEP)`, such that the following holds for each pair augmented configurations  $(\sigma''_S, \sigma''_T)$  in this subset:

- $\sigma''_S$  and  $\sigma''_T$  are produced by partial executions of  $(S, B)$  and  $(T, B)$  of equal length  $n$ , i.e. the respective flows of control made  $n$  transitions from block to block.
- Producing these configurations resulted in switching into the exit mode of execution.

**Definition 7.60.**

**bisimrel\_NI\_exitblk : Program × Program × BlckPosEnv × BlckPosEnv**  
**→ BisimulationRelation**

**bisimrel\_NI\_exitblk**( $S, T, B_S, B_T$ ) =  
 $\{(\sigma''_S, \sigma''_T) \mid \exists ss \ bs \ pid \ bid \ pc \ predbid \ n \ s \ ss' \ pid' \ bid' \ pc' \ predbid'. \\
\sigma''_S = (ss, bs, ss, \text{EXITBLCK}, predbid, (T, \text{AB}_{\text{ok}}, pc, \text{FLUSH}(n), s)) \wedge \\
\sigma''_T = (ss', bs, ss', \text{EXITBLCK}, predbid', (T, \text{AB}_{\text{ok}}, pc', \text{FLUSH}(n), s)) \wedge \\
M''((T, B_S), bs) = \sigma''_S \wedge \\
M''((S, B_T), bs) = \sigma''_T \}$

Definition 7.61 defines a function  $\text{bisimrel}_{\text{NI}}$  which computes a bisimulation relation  $\mathcal{R}_{\text{NI}} = \text{bisimrel}_{\text{NI}}(S, T, B_S, B_T, CBEP)$  for two IL<sup>n</sup> programs  $(S, B_S)$  and  $(T, B_T)$ , which are the source and the target programs, respectively, of a NI optimization, and a corresponding block edges relation  $CBEP$ . As aforementioned, a bisimulation relation  $\text{bisimrel}_{\text{NI}}(S, T, B_S, B_T, CBEP)$  is defined as a union of three disjoint sets comprising pairs of augmented configurations having three possible syntactic forms:

1. Each configuration pair  $(\sigma''_S, \sigma''_T)$  in the set  $\text{bisimrel\_NI\_normblk}(S, T, B_S, B_T, CBEP)$  consists of augmented configurations  $\sigma''_S$  and  $\sigma''_T$  such that the following holds:
  - a)  $\sigma''_S$  and  $\sigma''_T$  are produced by partial executions of the programs  $(S, B_S)$  and  $(T, B_T)$ , respectively.
  - b) These partial executions have the same length.
  - c) The syntactic forms of both  $\sigma''_S$  and  $\sigma''_T$  conform with configurations which are produced by partial executions in the normal mode.
2. Each configuration pair  $(\sigma''_S, \sigma''_T)$  in the set  $\text{bisimrel\_NI\_excbk}(S, T, B_S, B_T)$  consists of augmented configurations  $\sigma''_S$  and  $\sigma''_T$  such that the following holds:
  - a)  $\sigma''_S$  and  $\sigma''_T$  are produced by partial executions of the programs  $(S, B_S)$  and  $(T, B_T)$ , respectively.
  - b) These partial executions have the same length.
  - c) The syntactic forms of both  $\sigma''_S$  and  $\sigma''_T$  conform with configurations which are produced by partial executions in the exception mode.
3. Each configuration pair  $(\sigma''_S, \sigma''_T)$  in the set  $\text{bisimrel\_NI\_exitblk}(S, T, B_S, B_T)$  consists of augmented configurations  $\sigma''_S$  and  $\sigma''_T$  such that the following holds:
  - a)  $\sigma''_S$  and  $\sigma''_T$  are produced by partial executions of the programs  $(S, B_S)$  and  $(T, B_T)$ , respectively.
  - b) These partial executions have the same length.
  - c) The syntactic forms of both  $\sigma''_S$  and  $\sigma''_T$  conform with configurations which are produced by partial executions in the exit mode.

**Definition 7.61.**

**bisimrel<sub>NI</sub> : Program × Program × BlckPosEnv × BlckPosEnv**  
**× CorrespBlockEdgePairSet → BisimulationRelation**

**bisimrel<sub>NI</sub>**( $S, T, B_S, B_T, CBEP$ ) =  $\text{bisimrel\_NI\_normblk}(S, T, B_S, B_T, CBEP) \cup \\
\text{bisimrel\_NI\_excbk}(S, T, B_S, B_T) \cup \\
\text{bisimrel\_NI\_exitblk}(S, T, B_S, B_T,)$

### 7.3.3 Optimization correctness criterion for NI transformations

This section presents the formalization of a translation relation predicate  $TCC_{NI}$  on two IL<sup>''</sup> programs,  $(S, B_S)$  and  $(T, B_T)$ , and a corresponding block edge relation  $CBEP$ . The programs  $(S, B_S)$  and  $(T, B_T)$  are the source and the target program of a NI optimization and the relation  $CBEP$  is a byproduct of data flow analysis performed by the compiler for the LIH optimization and gives information on how the program transformation was performed. Informally,  $TCC_{NI}((S, B_S), (T, B_T), CBEP)$  holds true iff  $(T, B_T)$  is a correct NI optimization of  $(S, B_S)$  w.r.t.  $CBEP$ .

In the context of the definition  $TCC_{NI}$ , the relation  $CBEP$  can be seen as a proof hint generated by the compiler front-end for its proof generation unit. In the following, we call  $TCC_{NI}$  an *optimization correctness criterion for NI optimizations*. In our implementation, the  $TCC_{NI}$  predicate is an instance of the optimization correctness criterion  $OptCC_O$  that was described in the overview of Layer 5 in Section 3.6

The rest of this section consists of the following three parts:

- The first part gives formation rules for the sets associated with the definition of the optimization correctness criterion  $TCC_{NI}$ .
- The second part presents the definition a function `ni_transrel_block` which computes a NI optimization relation over block pairs for two IL<sup>''</sup> programs which are the source and the target of a NI optimization and a corresponding block edge relation which describes this optimization.
- The last part of this section presents the definition of the optimization correctness criterion  $TCC_{NI}$ .

We begin the presentation by introducing a set of relations over entry block position pairs.

#### **EntryBlockPosTransRel\_NI.**

Definition 7.62 gives formation rules for this set. A relation between entry block positions comprises pairs which have the syntactic form  $((bid, pid, pc), (bid, pid, pc))$ . Each such pair consists of two triples of the syntactic form  $(bid, pid, pc)$  consisting of a block identifier  $bid$ , a block position identifier  $pid$ , and an instruction number  $pc$ . In the context of the definition of the criterion  $TCC_{NI}$ , a triple  $(bid, pid, pc)$  denotes a block position  $pid$  which is the entry of a block  $bid$ . Additionally, the triple indicates that the program point  $pc$  is allocated to the entry block position  $pid$ . As we can always retrieve all informations about a block  $bid$  from an entry block position triple  $(bid, pid, pc)$  and a CFGB declaration  $B = (pid_0, BP, BB, succBP, predB)$  by computing appropriate lookups in the mappings of  $B$ , we use the triple as unique representations of blocks in the context of CFGB declarations. Thus, a set of entry block position pairs  $((bid, pid, pc), (bid, pid, pc))$  can be used to define a relation between blocks of two CFGB declarations in the context of two CFGB declarations.

#### **Definition 7.62.**

$$\begin{aligned} \text{EntryBlockPos} &= \text{BlkId} \times \text{BlkPosId} \times \text{InstructionNr} \\ \text{EntryBlockPosTransRel\_NI} &= \mathcal{P}(\text{EntryBlockPos} \times \text{EntryBlockPos}) \end{aligned}$$

Now, we present the second part of the formalization of  $TCC_{NI}$ , the formalization of a function `ni_transrel_block` which computes a translation relation over entry block position pairs for two IL<sup>''</sup> programs,  $(S, B_S)$  and  $(T, B_T)$ , and a corresponding block edge relation  $CBEP$ .

The definition of the function `ni_transrel_block` and the definitions of auxiliary functions used by `ni_transrel_block` make the following context assumptions:

- The compiler performed the loop invariant code analysis on a source program  $S$  and detected that  $n$  instructions in a loop with the program point  $pc$  as loop header are movable out of this loop.
- The compiler performed a NI transformation by inserting  $n$  goto instructions emulating nop instructions between the  $pc - 1$ -th and the  $pc$ -th instructions of  $S$ . The result of this program transformation is the program  $T$ .
- The number of nop instructions which can be inserted by the compiler is limited by a number  $n'$  which is determined by the capacity of the SVF. This capacity depends directly on the definition of the function `ni_transrel_block` which we explain later on. Our SVF provides a definition of `ni_transrel_block` which allows proving NI optimizations correct which insert maximally 10 nop instructions between two instructions. This thesis presents a definition `ni_transrel_block` which allows proving NI optimizations correct which insert maximally 3 nop instruction between two instructions.
- The byproduct of the NI transformation are a source and a target CFGB declarations  $B_S$  and  $B_T$  for  $S$  and  $T$ , respectively.
- All blocks in the source CFGB declaration  $B_S$  have a length equal one.
- The blocks in the target CFGB declaration  $B_T$  have a length greater than or equal one.
- The compiler generated a corresponding block edge relation  $CBEP$  from  $B_S$  and  $B_T$  that declares, among other things, which blocks of  $B_S$  and  $B_T$  are corresponding.

Let us consider source and target CFGB declarations  $B_S$  and  $B_T$ , a block  $bid$  in the set of blocks of  $B_S$ , a block  $bid'$  in the set of blocks of  $B_T$ , and a corresponding block edge relation  $CBEP$ . Let us assume that  $CBEP$  declares that the block  $bid$  and  $bid'$  are corresponding, i.e. there exist two blocks  $prebid$  and  $prebid'$  and a buffertype  $bt$  such that

$$((prebid, bid), (prebid', bid'), bt) \in CBEP.$$

Then, it follows from the context assumptions, which we described above that there are three possibilities when it comes to the lengths of the blocks  $bid$  and  $bid'$ :

1. the length of  $bid$  and  $bid'$  are equal 1 and 1, respectively, or
2. the length of  $bid$  and  $bid'$  are equal 1 and 2, respectively, or
3. the length of  $bid$  and  $bid'$  are equal 1 and 3, respectively.

As each block  $bid$  in the CFGB declaration  $B$  can be represented by a unique entry block position  $(bid, pid, pc)$ , there are further five cases for each of the above possibilities which arise from five cases of the syntactic form the  $pc$ -th instruction: The  $pc$ -th instruction can be either an assignment instruction or a printi instruction or a branch instruction or a goto instruction or an exit instruction. Altogether, there are fifteen possibilities for a pair of blocks  $(bid, bid')$  which are declared by  $CBEP$  as corresponding. As obviously there are same number of possibilities for pairs  $((bid, pid, pc), (bid', pid', pc'))$  consisting of entry block positions representing the blocks  $bid$  and  $bid'$  in the context of  $B_S$  and  $B_T$ , we define the translation relation over pairs of corresponding blocks as a relation between entry block position pairs which is a union of fifteen disjoint sets which arise from the above possibilities for the pair of blocks  $(bid, bid')$ .

In the following, we present the definitions of three auxiliary functions which compute three respective subsets of the relation between entry block position pairs  $((bid, pid, pc), (bid', pid', pc'))$  for the case that the  $pc$ -th instruction is an assignment.

Definition 7.63 defines a function `ni_transrel_assign_1_1` which make the above context assumptions about its parameters and computes a relation over entry block position pairs for two

IL” programs  $(S, B_S)$  and  $(T, B_T)$ ; and a corresponding block edge relation  $CBEP$ . For each entry block position pair

$$\begin{aligned} ((src\_bid_1, src\_pid_1, src\_pc_1), (trg\_bid_1, trg\_pid_1, trg\_pc_1)) \\ \in ni\_transrel\_assign\_1\_1((S, B_S), (T, B_T), CBEP) \end{aligned}$$

the following holds:

- The entry block position  $(src\_bid_1, src\_pid_1, src\_pc_1)$  represents the block  $src\_bid_1$  of the length equal one.
- The entry block position  $(trg\_bid_1, trg\_pid_1, trg\_pc_1)$  represents the block  $trg\_bid_1$  of the length equal one.
- Both blocks  $src\_bid_1$  and  $trg\_bid_1$  include exactly one block position  $src\_pid_1$  and  $trg\_pid_1$ , respectively.
- Program points  $src\_pc_1$  and  $trg\_pc_1$  are allocated to block positions  $src\_pid_1$  and  $trg\_pid_1$ , respectively.
- The  $src\_pc_1$ -th and  $trg\_pc_1$ -th instructions of the programs  $S$  and  $T$ , respectively, are assignments which are equal.
- The blocks  $src\_succbid$  and  $trg\_succbid$  are successors of the blocks  $src\_bid_1$  and  $trg\_bid_1$ , respectively.
- The blocks  $src\_succbid$  and  $trg\_succbid$  are declared as corresponding by the relation  $CBEP$ , respectively:

$$((src\_bid_1, src\_succbid), (trg\_bid_1, trg\_succbid), FTYPE) \in CBEP$$

- The CFGB declarations  $B_S$  and  $B_T$  declare  $src\_bid_1$  and  $trg\_bid_1$  as predecessor blocks of  $src\_succbid$  and  $trg\_succbid$ , respectively.

**Definition 7.63.**

```

ni_transrel_assign_1_1 : Program × Program × BlkPosEnv × BlkPosEnv
                        × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI

ni_transrel_assign_1_1(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_succbid succbsize trg_set.
      instrs!src_pc1 = (lval:=e) ∧
      instrs'!trg_pc1 = (lval:=e) ∧
      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧
      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧
      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, FTYPE) ∈ src_set ∧
      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 1, 0, trg_pc1) ∧
      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_succbid, succbsize, 0, trg_pc2) ∧
      trg_BB(trg_succbid) = Some(trg_pid2) ∧
      trg_predB(trg_pid2) = Some(trg_set) ∧
      (trg_bid1, FTYPE) ∈ trg_set ∧
      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }
end

```

*Example 7.64.* Consider Figure 7.1 depicting the source and the target CFGB declarations for the programs  $IL_2$  and  $IL_3$  depicted in Figure A.3 and the corresponding block edge pair relation  $CBEP$ . Given that our compiler front-end generates a theory with definitions of logic constants  $B_S$ ,  $B_T$ ,  $S$ ,  $T$ , and  $CBEP$  which represent the source CFGB declaration, the target CFGB declaration, the program  $IL_2$ , the program  $IL_3$ , and the relation  $CBEP$  on the logic level, respectively, then the following holds:

$$((b_1, p_{1,1}, 1), (b_1, p_{1,1}, 1), \text{FTYPE}) \in \text{ni\_transrel\_assign\_1\_1}(S, T, B_S, B_T, CBEP)$$

Consider the source and the target CFGB declarations for the programs  $IL_2$  and  $IL_3$  depicted in Figure A.3. According to the definitions of those declarations and programs, it holds that the entry block position pair ()

◇

Definition 7.65 defines a function  $\text{ni\_transrel\_assign\_1\_2}$  which make the same context assumptions about its parameters as the function  $\text{ni\_transrel\_assign\_1\_1}$  and computes a relation

over entry block position pairs for two IL<sup>''</sup> programs  $(S, B_S)$  and  $(T, B_T)$ ; and a corresponding block edge relation  $CBEP$ . For each entry block position pair

$$\begin{aligned} ((src\_bid_1, src\_pid_1, src\_pc_1), (trg\_bid_1, trg\_pid_1, trg\_pc_1)) \\ \in ni\_transrel\_assign\_1\_2((S, B_S), (T, B_T), CBEP) \end{aligned}$$

the following holds:

- The entry block position  $(src\_bid_1, src\_pid_1, src\_pc_1)$  represents the block  $src\_bid_1$  of the length equal one.
- The entry block position  $(trg\_bid_1, trg\_pid_1, trg\_pc_1)$  represents the block  $trg\_bid_1$  of the length equal two.
- The block  $src\_bid_1$  includes exactly one block position  $src\_pid_1$ .
- The program point  $src\_pc_1$  is allocated to block positions  $src\_pid_1$ .
- The  $src\_pc_1$ -th instruction of the programs  $S$  is an assignment.
- The block  $trg\_bid_1$  includes two block positions  $trg\_pid_1$  and  $trg\_pid_2$ . The block indexes of  $trg\_bid_1$  and  $trg\_bid_2$  are equal 0 and 1, respectively.
- Program points  $trg\_pc_1$  and  $trg\_pc_2$  are allocated to block positions  $trg\_pid_1$  and  $trg\_pid_2$ , respectively.
- The  $trg\_pc_1$ -th and  $trg\_pc_2$ -th instructions of the program  $T$  are a goto instruction, which emulates a nop instruction, and an assignment instruction, respectively.
- The  $src\_pc_1$ -th instruction of the program  $S$  and  $trg\_pc_2$ -th instruction of the program  $T$  are equal.
- The blocks  $src\_succbid$  and  $trg\_succbid$  are successors of the blocks  $src\_bid_1$  and  $trg\_bid_1$ , respectively.
- The blocks  $src\_succbid$  and  $trg\_succbid$  are declared as corresponding by the relation  $CBEP$ , respectively:

$$((src\_bid_1, src\_succbid), (trg\_bid_1, trg\_succbid), FTYPE) \in CBEP$$

- The CFGB declarations  $B_S$  and  $B_T$  declare  $src\_bid_1$  and  $trg\_bid_1$  as predecessor blocks of  $src\_succbid$  and  $trg\_succbid$ , respectively.

**Definition 7.65.**

```

ni_transrel_assign_1_2 : Program × Program × BlkPosEnv × BlkPosEnv
                        × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI

ni_transrel_assign_1_2(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_bid2 trg_pc3 trg_pid3
    trg_succbid trg_set.
      instrs!src_pc1 = (lval:=e) ∧
      instrs'!trg_pc1 = (goto(trg_pc2)) ∧
      instrs'!trg_pc2 = (lval:=e) ∧
      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧
      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧
      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, FTYPE) ∈ src_set ∧
      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 2, 0, trg_pc1) ∧
      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 2, 1, trg_pc2) ∧
      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_succbid, succbsize, 0, trg_pc3) ∧
      trg_BB(trg_succbid) = Some(trg_pid3) ∧
      trg_predB(trg_pid3) = Some(trg_set) ∧
      (trg_bid1, FTYPE) ∈ trg_set ∧
      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }

end

```

Definition 7.66 defines a function `ni_transrel_assign_1_3` which make the same context assumptions about its parameters as the function `ni_transrel_assign_1_1` and `ni_transrel_assign_1_2` and computes a relation over entry block position pairs for two IL" programs  $(S, B_S)$  and  $(T, B_T)$ ; and a corresponding block edge relation  $CBEP$ . For each entry block position pair

$$\begin{aligned}
& ((src\_bid_1, src\_pid_1, src\_pc_1), (trg\_bid_1, trg\_pid_1, trg\_pc_1)) \\
& \in ni\_transrel\_assign\_1\_3((S, B_S), (T, B_T), CBEP)
\end{aligned}$$

the following holds:

- The entry block position  $(src\_bid_1, src\_pid_1, src\_pc_1)$  represents the block  $src\_bid_1$  of the length equal one.
- The entry block position  $(trg\_bid_1, trg\_pid_1, trg\_pc_1)$  represents the block  $trg\_bid_1$  of the length equal three.
- The block  $src\_bid_1$  includes exactly one block position  $src\_pid_1$ .
- The program point  $src\_pc_1$  is allocated to block positions  $src\_pid_1$ .

- The  $src\_pc_1$ -th instruction of the programs  $S$  is an assignment.
- The block  $trg\_bid_1$  includes three block positions  $trg\_pid_1$ ,  $trg\_pid_2$ , and  $trg\_pid_3$ . The block indexes of  $trg\_bid_1$ ,  $trg\_bid_2$ , and  $trg\_bid_3$  are equal 0, 1, and 3, respectively.
- Program points  $trg\_pc_1$ ,  $trg\_pc_2$ , and  $trg\_pc_3$  are allocated to block positions  $trg\_pid_1$ ,  $trg\_pid_2$ , and  $trg\_pid_3$ , respectively.
- The  $trg\_pc_1$ -th,  $trg\_pc_2$ -th, and  $trg\_pc_3$ -th instructions of the program  $T$  are a goto instruction, a goto instruction, and an assignment instruction, respectively. The goto instructions emulate nop instructions.
- The  $src\_pc_1$ -th instruction of the program  $S$  and  $trg\_pc_3$ -th instruction of the program  $T$  are equal.
- The blocks  $src\_succbid$  and  $trg\_succbid$  are successors of the blocks  $src\_bid_1$  and  $trg\_bid_1$ , respectively.
- The blocks  $src\_succbid$  and  $trg\_succbid$  are declared as corresponding by the relation  $CBEP$ , respectively:

$$((src\_bid_1, src\_succbid), (trg\_bid_1, trg\_succbid), \text{FTYPE}) \in CBEP$$

- The CFGB declarations  $B_S$  and  $B_T$  declare  $src\_bid_1$  and  $trg\_bid_1$  as predecessor blocks of  $src\_succbid$  and  $trg\_succbid$ , respectively.

**Definition 7.66.**

```

ni_transrel_assign_1_3 : Program × Program × BlckPosEnv × BlckPosEnv
                        × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI

ni_transrel_assign_1_3(S, T, BS, BT, CBEP) =
let
  ((vs, instrs), I) = S;
  ((vs', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_bid2 trg_pc3 trg_pid3
      trg_succbid trg_set.
      instrs!src_pc1 = lval := e ∧
      instrs!trg_pc1 = goto(trg_pc2) ∧
      instrs!trg_pc2 = goto(trg_pc3) ∧
      instrs!trg_pc3 = lval := e ∧
      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧
      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧
      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, FTYPE) ∈ src_set ∧
      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 3, 0, trg_pc1) ∧
      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 3, 1, trg_pc2) ∧
      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_bid1, 3, 2, trg_pc3) ∧
      trg_pc4 = Suc(trg_pc3) ∧
      trg_succBP(trg_pid3, trg_pc4) = Some(trg_pid4) ∧
      trg_BP(trg_pid4) = Some(trg_pid4, trg_succbid, succbsize, 0, trg_pc4) ∧
      trg_BB(trg_succbid) = Some(trg_pid4) ∧
      trg_predB(trg_pid4) = Some(trg_set) ∧
      (trg_bid1, FTYPE) ∈ trg_set ∧
      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }
end

```

Definition 7.67 defines a function `ni_transrel_block` which make the same context assumptions about its parameters as the auxiliary functions `ni_transrel_assign_1_1` through `ni_transrel_assign_1_3` and computes a relation over entry block position pairs for two IL<sup>n</sup> programs ( $S, B_S$ ) and ( $T, B_T$ ); and a corresponding block edge relation  $CBEP$ . The relation is defined as a union of fifteen disjoint sets which arise from fifteen cases for an entry block position pair, which we explained above. For brevity, we moved a part of the definitions of functions computing those sets into Chapter C:

- The definition of the function `ni_transrel_printi_1_1` is shown in Definition C.4.
- The definition of the function `ni_transrel_printi_1_2` is shown in Definition C.5.
- The definition of the function `ni_transrel_printi_1_3` is shown in Definition C.6.
- The definition of the function `ni_transrel_branch_1_1` is shown in Definition C.7.

- The definition of the function `ni_transrel_branch_1_2` is shown in Definition C.8.
- The definition of the function `ni_transrel_branch_1_3` is shown in Definition C.9.
- The definition of the function `ni_transrel_goto_1_1` is shown in Definition C.10.
- The definition of the function `ni_transrel_goto_1_2` is shown in Definition C.11.
- The definition of the function `ni_transrel_goto_1_3` is shown in Definition C.12.
- The definition of the function `ni_transrel_exit_1_1` is shown in Definition C.13.
- The definition of the function `ni_transrel_exit_1_2` is shown in Definition C.14.
- The definition of the function `ni_transrel_exit_1_3` is shown in Definition C.15.

**Definition 7.67.**

$$\begin{aligned} \text{ni\_transrel\_block} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \times \mathbf{BlckPosEnv} \\ &\quad \times \mathbf{CorrespBlckEdgePairSet} \rightarrow \mathbf{EntryBlckPosTransRel\_NI} \\ \text{ni\_transrel\_block}(S, T, B_S, B_T, CBEP) &= \\ &\quad \text{ni\_transrel\_assign\_1\_1}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_assign\_1\_2}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_assign\_1\_3}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_printi\_1\_1}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_printi\_1\_2}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_printi\_1\_3}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_branch\_1\_1}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_branch\_1\_2}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_branch\_1\_3}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_goto\_1\_1}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_goto\_1\_2}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_goto\_1\_3}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_exit\_1\_1}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_exit\_1\_2}(S, T, B_S, B_T, CBEP) \cup \\ &\quad \text{ni\_transrel\_exit\_1\_3}(S, T, B_S, B_T, CBEP) \end{aligned}$$

Finally, with the definition of the function `ni_transrel_block` at hand, we can present the last part of the formalization of the optimization correctness criterion for NI optimizations, namely, the definition of the function  $\text{TCC}_{\text{NI}}$  which has the form of a predicate on two IL<sup>2</sup> programs,  $(S, B_S)$  and  $(T, B_T)$ , and a corresponding block edges relation  $CBEP$ , makes the same context assumptions about  $(S, B_S)$ ,  $(T, B_T)$ , and  $CBEP$  as the definition of the function `ni_transrel_block`, and which checks if  $(S, B_S)$  and  $(T, B_T)$  are in a translation relation w.r.t.  $CBEP$ . Informally, checking if  $\text{TCC}_{\text{NI}}((S, B_S), (T, B_T), CBEP)$  holds true is done in third steps as follows. The first step computes a list of block pairs  $(bid, bid')$  consisting of blocks  $bid$  and  $bid'$  which are in the sets of blocks of the CFGB declarations  $B_S$  and  $B_T$ , respectively. The second step takes the list of block pairs computed in the first step and the source and target CFGB declarations  $B_S$  and  $B_T$  as input and maps this list to the list of entry block position pairs  $((bid, pid, pc), (bid', pid', pc'))$  by mapping blocks to their unique representations as entry block positions in the context of the respective CFGB declaration. The third step checks if each entry block position pairs  $((bid, pid, pc), (bid', pid', pc'))$  is in the translation relation  $\text{ni\_transrel\_block}(S, T, B_S, B_T, CBEP)$ . Additionally, the if the entry blocks of  $B_S$  and  $B_T$  are declared by the relation  $CBEP$  as corresponding. Thus, the predicate  $\text{TCC}_{\text{NI}}$  is a defined as extension of a predicate on a single entry block position pair of the form:

$$((bid, pid, pc), (bid', pid', pc')) \in \text{ni\_transrel\_block}(S, T, B_S, B_T, CBEP)$$

to a predicate on a set of entry block position pairs whose elements consist of entry block positions representing blocks declared by  $CBEP$  as corresponding.

Definition 7.68 defines a function `TCC_NI_entry_block` which takes a source and a target programs  $S$  and  $T$ , respectively, a source and a target CFGB declarations  $B_S$  and  $B_T$ , respectively, and a corresponding block edges relation  $CBEP$  and checks if the entry blocks of  $B_S$  and  $B_T$ ,  $src\_bid_0$  and  $trg\_bid_0$ , respectively, are well-formed and if they are declared by  $CBEP$  as corresponding. The entry blocks  $src\_bid_0$  and  $trg\_bid_0$  are well-formed and corresponding w.r.t. to  $B_S$ ,  $B_T$ , and  $CBEP$  iff they fulfill the following requirements:

1. The length of  $src\_bid_0$  is equal one.
2. The 0-th instruction of  $S$  is a non-printing instruction.
3. The CFGB declaration  $B_S$  declares  $src\_bid_0$  as a predecessor of  $src\_bid_0$ .
4. The length of  $trg\_bid_0$  is equal to a value  $trgbsize$  one.
5. The  $(trgbsize - 1)$ -th instruction of  $T$  is a non-printing instruction.
6. The CFGB declaration  $B_T$  declares  $trg\_bid_0$  as a predecessor of  $trg\_bid_0$ .
7. The  $CBEP$  relation declares the blocks  $src\_bid_0$  and  $trg\_bid_0$  as corresponding, i.e.  $CBEP$  comprises a corresponding block edge  $((src\_bid_0, src\_bid_0), (trg\_bid_0, trg\_bid_0), FTYPE)$ .

**Definition 7.68.**

```

TCC_NI_entry_block : Program × Program × BlckPosEnv × BlckPosEnv
                    × CorrespBlockEdgePairSet → Bool

TCC_NI_entry_block( $S, T, B_S, B_T, CBEP$ ) =
  let
     $((vds, instrs), I) = S;$ 
     $((vds', instrs'), I') = T;$ 
     $(pid_0, src\_BP, src\_BB, src\_succBP, src\_predB) = B_S$ 
     $(pid'_0, trg\_BP, trg\_BB, trg\_succBP, trg\_predB) = B_T$ 
  in
     $\exists src\_bid_0\ src\_set\ trg\_bid_0\ trg\_set\ trgbsize.$ 
       $src\_BP(src\_pid_0) = \text{Some}(src\_pid_0, src\_bid_0, 1, 0, 0) \wedge$ 
       $src\_predB(src\_pid_0) = \text{Some}(src\_set) \wedge$ 
       $(src\_bid_0, FTYPE) \in src\_set \wedge$ 
       $trg\_BP(trg\_pid_0) = \text{Some}(trg\_pid_0, trg\_bid_0, trgbsize, 0, 0) \wedge$ 
       $trg\_predB(trg\_pid_0) = \text{Some}(trg\_set) \wedge$ 
       $(trg\_bid_0, FTYPE) \in trg\_set \wedge$ 
       $((src\_bid_0, src\_bid_0), (trg\_bid_0, trg\_bid_0), FTYPE) \in CBEP$ 
  end

```

Definition 7.69 defines the predicate `TCC_NI_normal_block` on a source program  $S$ , a target program  $T$ , a source CFGB declaration  $B_S$ , a target declaration  $B_T$ , a corresponding block edges relation  $CBEP$ , and a corresponding block edge  $((prebid, bid), (prebid', bid'), bt)$ , which declares blocks  $bid$  and  $bid'$  as corresponding, and checks if the pair of entry block positions representing these blocks is in the translation relation computed by the function `ni_transrel_block` for  $S$ ,  $T$ ,  $B_S$ ,  $B_T$ , and  $CBEP$ .

**Definition 7.69.**

$$\begin{aligned} \text{TCC\_NI\_normal\_block} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \times \mathbf{BlckPosEnv} \\ &\quad \times \mathbf{CorrespBlockEdgePairSet} \times \mathbf{BlockEdgePair} \rightarrow \mathbf{Bool} \\ \text{TCC\_NI\_normal\_block}(S, T, B_S, B_T, CBEP, blckedgepair) = & \\ \text{let} & \\ & ((vds, instrs), I) = S; \\ & ((vds', instrs'), I') = T; \\ & (pid_0, src\_BP, src\_BB, src\_succBP, src\_predB) = B_S \\ & (pid'_0, trg\_BP, trg\_BB, trg\_succBP, trg\_predB) = B_T \\ & ((predbid, bid), (predbid', bid'), bt) = blckedgepair \\ \text{in} & \\ & \exists pid\ pid'\ bsize'\ pc\ pc'. \\ & \quad src\_BB(bid) = \text{Some}(pid) \wedge \\ & \quad trg\_BB(bid') = \text{Some}(pid') \wedge \\ & \quad src\_BP(pid) = \text{Some}(pid, bid, 1, 0, pc) \wedge \\ & \quad trg\_BP(pid') = \text{Some}(pid', bid', bsize', 0, pc') \wedge \\ & \quad ((bid, pid, pc), (bid', pid', pc')) \in \text{ni\_transrel\_block}(S, T, B_S, B_T, CBEP) \\ \text{end} & \end{aligned}$$

Definition 7.70 defines an optimization correctness criterion for NI optimizations on a source program  $S$ , a target program  $T$ , a source CFGB declaration  $B_S$ , a target CFGB declaration  $B_T$ , and a corresponding block edges relation  $CBEP$  which checks if entry blocks of  $B_S$  and  $B_T$  are well-formed and declared by  $CBEP$  as corresponding; and all pairs consisting of entry block positions representing block which are declared by  $CBEP$  as corresponding are in the translation relation  $\text{ni\_transrel\_block}(S, T, B_S, B_T, CBEP)$ .

**Definition 7.70.**

$$\begin{aligned} \text{TCC}_{\text{NI}} &: \mathbf{Program} \times \mathbf{Program} \times \mathbf{BlckPosEnv} \times \mathbf{BlckPosEnv} \\ &\quad \times \mathbf{CorrespBlockEdgePairSet} \rightarrow \mathbf{Bool} \\ \text{TCC}_{\text{NI}}(S, T, B_S, B_T, CBEP) = & \\ \text{let} & \\ & ((vds, instrs), I) = S; \\ & ((vds', instrs'), I') = T; \\ & (pid_0, src\_BP, src\_BB, src\_succBP, src\_predB) = B_S \\ & (pid'_0, trg\_BP, trg\_BB, trg\_succBP, trg\_predB) = B_T \\ \text{in} & \\ & \text{TCC\_NI\_entry\_block}(S, T, B_S, B_T, CBEP) \\ & \wedge \\ & \forall blckedgepair \in CBEP. \text{TCC\_NI\_normal\_block}(S, T, B_S, B_T, CBEP, blckedgepair) \\ \text{end} & \end{aligned}$$
**7.3.4 Verification of the optimization correctness criterion  $\text{TCC}_{\text{NI}}$** 

This section presents the theorems which we proved in order to verify the specification of the optimization correctness criterion  $\text{TCC}_{\text{NI}}$  presented in the previous section. The main result in this section is a theorem which can be used directly in translation certificates generated by our compiler.

To verify the specification of the criterion  $\text{TCC}_{\text{NI}}$ , we proved Theorem 7.71 which is a statement about a source and a target programs of a concrete NI optimization,  $S$  and  $T$ , a program type  $\Phi$ ,

a source CFGB declaration  $B_S$ , a target declaration  $B_T$ , and a corresponding block edge relation  $CBEP$  which says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B_S$  and  $B_T$  are well-formed w.r.t.  $S$  and  $T$ , respectively; and  $S$  and  $T$  fulfill the optimization correctness criterion  $TCC_{NI}$  w.r.t.  $B_S$ ,  $B_T$ , and  $CBEP$ , then  $S$  and  $T$  fulfill the optimization independent translation correctness  $TCC$  w.r.t. the bisimulation relation  $\text{bisimrel}_{NI}(S, T, B_S, B_T, CBEP)$ .

**Theorem 7.71.**

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge TCC_{NI}(S, T, B_S, B_T, CBEP) \\ & \implies \\ & TCC(S, T, \text{bisimrel}_{NI}(S, T, B_S, B_T, CBEP)) \end{aligned}$$

□

Finally, we present the main result in this section, a theorem which is a statement about a source and a target programs of a concrete NI optimization,  $S$  and  $T$ , a program type  $\Phi$ , a source CFGB declaration  $B_S$ , a target CFGB declaration  $B_T$ , and a corresponding block edge relation  $CBEP$  which says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B_S$  and  $B_T$  are well-formed w.r.t.  $S$  and  $T$ ; and  $S$  and  $T$  fulfill the optimization correctness criterion  $TCC_{NI}$  w.r.t.  $B_S$ ,  $B_T$ , and  $CBEP$ , then  $S$  and  $T$  fulfill the translation correctness predicate  $\text{corrTrans}$ .

**Theorem 7.72.**

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge TCC_{NI}(S, T, B_S, B_T, CBEP) \\ & \implies \\ & \text{corrTrans}(S, T) \end{aligned}$$

*Proof.*

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B_S) \wedge \text{wfB}(T, B_T) \wedge TCC_{NI}(S, T, B_S, B_T, CBEP) \\ & \implies \text{[by application of Theorem 7.71]} \\ & TCC(S, T, \text{bisimrel}_{NI}(S, T, B_S, B_T, CBEP)) \\ & \implies \text{[by application of the existential introduction rule]} \\ & \exists \mathcal{R}. TCC(S, T, \mathcal{R}). \\ & \implies \text{[by application of Theorem 6.44]} \\ & \text{corrTrans}(S, T) \end{aligned}$$

□

Theorem 7.72 is an instance of the corollary (I) in Section 3.6 and is directly applicable in translation certificates which are generated by the compiler for each NI transformation, see the end of Section 3.6 for a general application scheme of this theorem.

## 7.4 SVF for RAI optimizations

This section presents formalization of a formal verification framework for proving RAI transformations correct.

As aforementioned in Section 1.5.1, the RAI transformation takes an integer  $pc$  and an assignment instruction  $x := e$  as input and replaces the  $pc$ -th instruction of a program by this assignment. We call this transformation a redundant assignment insertion because it is used to insert an assignment  $x := e$  only at a program point  $pc$  at which it is made dead by an equal assignment instruction at another program point  $pc'$ . The RAI transformations which are performed by our compiler front-end can be characterized as follows.

- The RAI transformation is the second intermediate program transformation in a chain of three program transformations which altogether perform the LIH optimization.
- The first transformation in this chain is the NI transformation which inserts a limited number of nop instructions between the header and preheader of the loop which is the result of translation of a do-while loop<sup>3</sup> in the original  $\mu C$  program and contains loop invariant instructions. The number of inserted nop instructions depends on the number of loop invariant instructions and on how many of those instructions can be moved outside the loop without modifying observable behavior of the program.
- Given that the above loop contains an assignment of the syntactic form  $v := e$  which is invariant and movable outside the loop, then the function of the RAI transformation is to replace a nop instruction inserted by the NI transformation between the preheader and header of this loop by the same assignment  $v := e$ . As the loop is the result of translation of a do-while loop, the loop header node in the CFG of the program is the dominator<sup>4</sup> of all nodes in the loop. The assignment replaces a nop instruction which is a label of a node inserted by the NI transformation between the preheader and header nodes of the loop. As this node is the dominator of the loop header, it is the dominator of the node inside loop which labeled by the assignment  $v := e$ . This implies that the assignment placed by the RAI transformation is made dead by the same assignment inside the loop.

*Example 7.73.* Figure A.4 depicts two programs  $IL_3$  and  $IL_4$  which are the source and the target programs of the RAI transformation in our example illustrating the work-flow of our compiler front-end in Figure 1.5. The CFG's of the programs  $IL_3$  and  $IL_4$  are equal. The CFG of the program  $IL_3$  contains a loop comprising the program points: 12 through 27. The node 12 is the header of this loop. The 11-th instruction is the goto instruction `GOTO [12]` which emulates a nop instruction. The 13-th and 14-th instructions, `_tI_6 = n * n` and `tmp = _tI_6`, respectively, are loop invariant and are both movable outside the loop but only moving the 13-th instruction in one step can be proved correct using our SVF. Therefore, our compiler front-end performed the RAI transformation which replaced the 11-th instruction by the assignment `_tI_6 = n * n`. The result of this transformation is the program  $IL_4$ . In the program  $IL_4$ , the 11-th instruction `_tI_6 = n * n` is made dead by the 13-th instruction `_tI_6 = n * n`. Inserting both instructions `_tI_6 = n * n` and `tmp = _tI_6` between the preheader and header of the loop in one step would not be a proper RAI transformation because the instruction `tmp = _tI_6` uses the variable `_tI_6` and thus it would make the instruction `_tI_6 = n * n` live.  $\diamond$

In general, the RAI transformation can be characterized as follows.

- The RAI is a structure preserving transformation, i.e. it does not modify the sets of edges and nodes of the CFG of the source program. For this reason, the CFG's and the CFGB declarations involved in RAI transformations have the same properties as CFG's and CFGB declarations

<sup>3</sup> The reader should recall that the LIH optimizations are only possible on do-while loops, cf. [3].

<sup>4</sup> In this thesis, the notion of dominators strictly follows the definition given by Andrew W. Appel in [3], page 407: "Each control-flow graph must have a start node  $s_0$  with no predecessors, where program (or procedure) execution is assumed to begin. A node  $d$  dominates a node  $n$  if every path of directed edges from  $s_0$  to  $n$  must go through  $d$ . Every node dominates itself."

involved in CF and DAE optimizations performed by our compiler front-end, cf. Sections 7.1 and 7.2:

- The CFG of the source and the target programs of a RAI transformation are identical,
- The CFGB declarations for the source and the target programs of a RAI transformation are identical,
- The corresponding program points relation between program points of the source and the target programs of a RAI transformations is defined as identity, i.e. a one-to-one correspondence between program points of these programs,
- For both the source and the target program of a RAI transformation, the allocation relation between program points and block positions is a one-to-one correspondence,
- The "includes" relation between blocks and block positions is a one-to-one correspondence,
- The CFGB declaration used in a RAI transformation no nested blocks.
- The RAI transformation increases the number of instructions which are dead and decreases the number of goto instructions which emulate nop instructions in a program since it replaces the nop instructions of a program by dead assignments. Therefore, the RAI can be seen as a transformation which is can be seen as a "reverse" transformation of the DAE optimization because the latter replaces dead assignments by goto instructions which emulate nop instructions.

The above general properties of the RAI transformation enables the compiler to *reuse* the SVF for DAE optimizations in the correctness proof of a RAI transformation. Reusing is done in the following way: Let us assume that our compiler front-end has to generate a proof that an IL program  $T$  is a correct RAI transformation of an IL program  $S$ , i.e. has to generate a proof of the statement  $\text{corrTrans}(S, T)$ . Then, the compiler performs the liveness analysis on the program  $T$ . The result of the analysis are a liveness analysis result  $\mathcal{A}_{LA}$  and a byproduct of the analysis, a CFGB declaration  $B$ . The compiler passes  $S$ ,  $T$ ,  $B$ , and  $\mathcal{A}_{LA}$  to its proof generation unit which generates a proof script which consists of three parts:

1. The first part comprises definitions of logic constants representing  $T$ ,  $S$ ,  $B$ ,  $\mathcal{A}_{LA}$ , and a program type  $\Phi$  which is the type of both  $S$  and  $T$ .
2. The second part of the script treats the programs  $T$  and  $S$  as the source and the target of a DAE optimization and proves the statement  $\text{corrTrans}(T, S)$  by applying SVF for DAE optimizations presented in Section 7.2. In particular, the first step of this proof applies Theorem 7.55. The subsequent proof steps discharge the assumptions of the theorem by proving that
  - $\text{wtp}(T, \Phi)$  holds true,
  - $\text{wtp}(S, \Phi)$  holds true,
  - $\text{wfB}(T, B)$  holds true,
  - $\text{wfB}(S, B)$  holds true, and
  - $\text{TCC}_{\text{DAE}}(T, S, B, \mathcal{A}_{LA})$  holds true.
3. The third part of the script derives the statement  $\text{corrTrans}(S, T)$  from the statement  $\text{corrTrans}(T, S)$  proved in the second part of the script which is trivial by the definition of  $\text{corrTrans}$ .

## 7.5 SVF for RAE

This section presents formalization of an optimization correctness criterion for RAE optimizations,  $\text{TCC}_{\text{RAE}}$ , and a corresponding optimization correctness theorem which is directly applied in translation certificates generated by our compiler front-end.

As aforementioned in Section 1.5.1, the RAE transformation takes an integer  $pc$  as input and replaces the  $pc$ -th instruction of a program, which is an assignment  $x := e$ , by a goto instruction

which emulates a nop instruction. We call this transformation a *redundant assignment elimination* because it is used to remove an assignment  $x := e$  which is the label of a node  $pc$  in the CFG of a program and is *redundant* at the entry to this node. Given an IL program  $P$ , the CFG of  $P$ , and a node  $pc$  in the set of nodes of the CFG, then an assignment  $lval := e$  is redundant as a label of the node  $pc$  iff

- the assignment has the syntactic form  $v := e$  and the expression  $e$  is ABexc-safe in  $P$ ,
- for all paths from the entry node of the CFG to the node  $pc$ , there exists a node  $pc'$  such that the assignment  $v := e$  is the label of  $pc'$  and the nodes between  $pc'$  and  $pc$  neither re-define the variable  $v$  nor the variables of the expression  $e$ .

The above definition implies the following: Whenever the flow of control transfers to a node  $pc$  labeled by a redundant assignment  $v := e$ , evaluation of the variable  $v$  and the expression  $e$  in the context of current state of computation yields equal results. Executing the assignment  $v := e$  results in updating the state but evaluation of the variable  $v$  and the expression  $e$  in the context of new state of computation also yields equal results. Therefore, we call the assignment  $v := e$  redundant.

To be able to perform a correct RAE optimization, the compiler has to perform a data flow analysis which we call *available equations analysis* (AEA) and which is a composition of reaching definitions analysis and available expressions analysis. Informally, the result of this analysis is a mapping from program points to sets of a *available equations*,  $\mathcal{A}_{AEA}$ , where an available equation is a pair  $(v, e)$  consisting of a variable  $v$  and an expression  $e$ . If the compiler determines during the AEA that an assignment  $v := e$  is redundant as a label of the node  $pc$ , then the result of the AEA,  $\mathcal{A}_{AEA}$ , maps  $pc$  to an available equation set which comprises the pair  $(v, e)$ .

### 7.5.1 Abstract syntax of AEA results

This section presents the definition of the abstract syntax of the AEA results. We begin the presentation by listing syntactic sets associated with this notion:

- "gen" available equations **GenSet**,
- "kill" available equations **KillSet**,
- "in" available equations **InSet**,
- "out" available equations **OutSet**,
- tuples of sets of available equations **AESets**, and
- available equation sets environments **AESetsEnv**;

and defining metavariables ranging over these sets:

- $gen$  ranges over "gen" available equations **GenSet**,
- $kill$  ranges over "kill" available equations **KillSet**,
- $in$  ranges over "in" available equations **InSet**,
- $out$  ranges over "out" available equations **OutSet**,
- $aesets$  ranges over tuples of available equation sets **AESets**, and
- $\mathcal{A}_{AEA}$  ranges over available equation sets environments **AESetsEnv**;

Definition 7.74 gives formation rules for the set of the AEA results **AESetsEnv**. An AEA result  $\mathcal{A}_{AEA}$  is a partial mapping from instruction numbers **InstructionNr** to tuples of available equation sets **AESets**. We call the result of an AEA a *available equation environment*. If an AEA result  $\mathcal{A}_{AEA}$  is well-formed, it is total. A tuple  $aesets = (gen, kill, in, out)$  consists of a set of "gen" variables  $gen$ , a set of "kill" variables  $kill$ , a set of "in" variables  $in$ , and a set of "out" variables  $out$ . The purpose of a AEA result  $\mathcal{A}_{AEA}$  is to model facts about the sets *available equations* for all program points of a program which were determined by the compiler during a concrete AEA. An equation  $(v, e)$  is available at the entry to a node  $pc$  in the CFG of a program iff

- the expression  $e$  is ABexc-safe in  $P$ ,
- for all paths from the entry node of the CFG to the node  $pc$ , there exists a node  $pc'$  such that the assignment  $v:=e$  is the label of  $pc'$  and the nodes between  $pc'$  and  $pc$  neither re-define the variable  $v$  nor the variables of the expression  $e$ .

If the compiler determines during the AEA that an equation  $(v, e)$  is available at the entry to a node  $pc$  in the CFG of a program, then the AEA result  $\mathcal{A}_{AEA}$  maps  $pc$  to a tuple  $(gen, kill, in, out)$  and  $(v, e) \in in$ . If an equation  $(v, e)$  is available at the entry to a node  $pc$ , then the assignment  $v:=e$  is redundant as a label of  $pc$  and can be replaced by a goto instruction which emulates a nop instruction.

**Definition 7.74.**

<b>AvailableEquation</b>	$\ni ae$	$:= (v, e)$
$gen$	$\in \mathbf{GenSet}$	$= \mathcal{P}(\mathbf{AvailableEquation})$
$kill$	$\in \mathbf{KillSet}$	$= \mathcal{P}(\mathbf{AvailableEquation})$
$in$	$\in \mathbf{InSet}$	$= \mathcal{P}(\mathbf{AvailableEquation})$
$out$	$\in \mathbf{OutSet}$	$= \mathcal{P}(\mathbf{AvailableEquation})$
$aesets$	$\in \mathbf{AESets}$	$= \mathbf{GenSet} \times \mathbf{KillSet} \times \mathbf{InSet} \times \mathbf{OutSet}$
$\mathcal{A}_{AEA}$	$\in \mathbf{AESetsEnv}$	$= \mathbf{InstructionNr} \rightsquigarrow \mathbf{AESets}$

### 7.5.2 Bisimulation relation for the RAE optimization

This section presents the definition of a function  $\text{bisimrel}_{RAE}$  which computes a bisimulation relation for two IL" programs, which are the source and the target programs of a RAE optimization, a result of the AEA which was performed by the compiler prior to that optimization, and a corresponding block edges relation. In the following presentation, we use convention that we write abbreviation  $\mathcal{R}_{RAE}$  when we mean a bisimulation relation for a RAE optimization.

Informally, our notion of a bisimulation relation  $\mathcal{R}_{RAE}$  can be characterized as follows.

- $\mathcal{R}_{RAE}$  is a function of a RAE optimization which is described by
  - a source IL program  $S$ ,
  - a target IL program  $T$ ,
  - a CFGB declaration  $B$ ,
  - a result of the AEA  $\mathcal{A}_{AEA}$ , and
  - a corresponding block edges relation  $CBEP$ .
- By Definition 6.40,  $\mathcal{R}_{RAE}$  has to be a subset of

**Configuration''**  $\times$  **Configuration''**.

- As the RAE is a structure preserving optimization, i.e. it does not modify the sets of nodes and edges of the CFG of a program, the CFGB declarations  $B_S$  and  $B_T$  for  $S$  and  $T$ , respectively, are also identical. Therefore, our compiler front-end computes only one CFGB declaration  $B$ , for both  $S$  and  $T$ . On the other hand, our compiler performs RAE optimizations by replacing redundant assignments by goto instructions emulating nop instructions. Thus, we know that it

holds for two arbitrary partial executions of  $(S, B)$  and  $(T, B)$  of the same length that if they produce augmented configurations  $\sigma_S''$  and  $\sigma_T''$ , then all their components are equal

$$\begin{aligned}
& \forall n S T B. \\
& \quad \mathbb{M}''((S, B), \text{init}(S, B), n) = \sigma_S'' \wedge \\
& \quad \mathbb{M}''((T, B), \text{init}(T, B), n) = \sigma_T'' \\
& \quad \longrightarrow \\
& \quad \exists ss \ bs \ bposstat \ predbid \ tf \ af \ pc \ b \ s \ ss' \ bs' \ bposstat' \ predbid' \ tf' \ af' \ pc' \ b' \ s'. \\
& \quad \quad \sigma_S'' = (ss, bs, ss, bposstat, predbid, (tf, af, pc, b, s)) \wedge \\
& \quad \quad \sigma_T'' = (ss', bs', ss', bposstat', predbid', (tf', af', pc', b', s')) \wedge \\
& \quad \quad ss = ss' \wedge \\
& \quad \quad bs = bs' \wedge \\
& \quad \quad bs = n \wedge \\
& \quad \quad bposstat = bposstat' \wedge \\
& \quad \quad predbid = predbid' \wedge \\
& \quad \quad tf = tf' \wedge \\
& \quad \quad af = af' \wedge \\
& \quad \quad pc = pc' \wedge \\
& \quad \quad b = b' \wedge \\
& \quad \quad s = s'
\end{aligned}$$

- The proof of a statement saying that  $(T, B)$  is a correct RAE optimization of  $(S, B)$  implies  $\text{bisimulation}((S, B), (T, B), \mathcal{R}_{RAE})$  has to be conducted by induction on the length of partial execution of the IL" program  $(S, B)$ , i.e. induction on the number of block-wise transitions made by the flow of control during execution of  $(S, B)$ . For the induction step, one has to prove that equality of corresponding components of two augmented configurations

$$\begin{aligned}
& \mathbb{M}''((S, B), \text{init}(S, B), n) \\
& \mathbb{M}''((T, B), \text{init}(T, B), n)
\end{aligned}$$

implies equality of corresponding components of their successors

$$\begin{aligned}
& \mathbb{M}''((S, B), \text{init}(S, B), n + 1) \\
& \mathbb{M}''((T, B), \text{init}(T, B), n + 1)
\end{aligned}$$

As some of assignments in  $S$  are replaced by goto's, we can not prove that the state components in the successor configurations always remain equal after making block-wise transition. Therefore, we have to strengthen the above execution invariant by an additional statement expressing conformance of a state  $s$  and a result of the AEA which, informally, says the following:

*For all equations  $(v, e)$  which are available at the program point  $pc$ , it holds that evaluation of  $v$  and  $e$  in the context of  $s$  yields equal values without causing an array-index-out-of-bounds exception.*

In order to be able to express the statements of this kind, we formalized the notion of conformance of a state with a AEA result.

- Analogously to the bisimulation relations  $\mathcal{R}_{CF}$ ,  $\mathcal{R}_{DAE}$ , and  $\mathcal{R}_{NI}$ , we define the bisimulation relation  $\mathcal{R}_{RAE}$  as a union of three disjoint sets. Each of these sets is defined for one of three modes of execution into which execution of an IL" program can switch as a result of making transition: the normal mode, the exception mode, and the exit mode, cf. Section 6.4.2. As we know what syntactic forms do the augmented configurations have in those modes, we define three functions which compute these sets as a function of the source and the target programs of a RAE optimization, a CFGB declaration, a AEA result, and a corresponding block edges relation.

We begin the presentation of the definition of  $\mathcal{R}_{RAE}$  with the formalization of the notion of conformance of a state with an AEA result.

Definition 7.75 defines a function `prg2Uset` which computes the universe set of available equations for an IL program, i.e. the set of all available equations which are found in the program.

**Definition 7.75.**

```

prg2Uset : Program → P(AvailableEquation)
prg2Uset(P) =
  let
    ((vds, instrs), I) = P
  in
    { (v, e) | ∃ pc. 0 ≤ pc ∧ pc < length(instrs) ∧
                instrs!pc = (v:=e) ∧ v ∉ expr2use(e) ∧ ABexc_safe_expr(P, e) } end

```

Definition 7.76 defines a function `instr2gen` which computes the "gen" set from an instruction, i.e. the set of available equations which are *generated* by the instruction.

An equation  $(v, e)$  is generated by an assignment  $v := e$  in a program  $P$  iff the expression  $e$  is ABexc-safe in  $P$  and  $v$  is not in the set of variables of the expression  $e$ .

**Definition 7.76.**

```

instr2gen : Program × Instruction → GenSet
instr2gen(P, v:=e)      = if ABexc_safe_expr(P, e) ∧ v ∉ expr2use(e) then {(v, e)} else {}
instr2gen(P, a[i]:=e)  = {}
instr2gen(P, a[v]:=e)  = {}
instr2gen(P, printi(e)) = {}
instr2gen(P, branch(e, dst)) = {}
instr2gen(P, goto(dst)) = {}
instr2gen(P, exit)     = {}

```

Definition 7.77 defines a function `instr2kill` which computes the "kill" set from an instruction, i.e. the set of available equations which are *killed* by the instruction.

An assignment  $lval := e$  in a program  $P$  kills an available equation  $(v, e)$  from the universe set of  $P$  iff executing an assignment can modify the value of  $v$  or  $e$  in the current state of computation.

**Definition 7.77.**

```

instr2kill : Program × Instruction → KillSet
instr2kill(P, v:=e)      = { (v', e') | (v', e') ∈ prg2Uset(P) ∧ (v' = v ∨ v ∈ expr2use(e')) }
instr2kill(P, a[i]:=e)  = { (v', e') | (v', e') ∈ prg2Uset(P) ∧ a ∈ expr2use(e') }
instr2kill(P, a[v]:=e)  = { (v', e') | (v', e') ∈ prg2Uset(P) ∧ a ∈ expr2use(e') }
instr2kill(P, printi(e)) = {}
instr2kill(P, branch(e, dst)) = {}
instr2kill(P, goto(dst)) = {}
instr2kill(P, exit)     = {}

```

Definition 7.78 defines a predicate `confaesets` on an AEA result  $\mathcal{A}_{AEA}$ , a CFG edge  $(pc, succpc)$ , and a state  $s$  which checks if the state  $s$  conforms with the AEA result  $\mathcal{A}_{AEA}$  w.r.t. the CFG edge  $(pc, succpc)$ . The `confaesets` makes the context assumptions as follows

- $pc$  and  $s$  are the program counter and state components of a configuration which is the result of partial execution of an IL program  $S$  which is the source program of a RAE optimization.

- The flow of control is about to make transition along the CFG edge  $(pc, pc')$ .

and checks if the following holds:

1. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program points  $pc$  and  $succpc$  must be a proper solution of data flow equations defined for the program  $S$ .
2. The state  $s$  must conform with the "out" set of available equations declared by  $\mathcal{A}_{AEA}$  for the node  $pc$ .

**Definition 7.78.**

$\text{confaesets} : \text{AESetsEnv} \times \text{InstructionNr} \times \text{InstructionNr} \times \text{State} \rightarrow \text{Bool}$   
 $\text{confaesets}(\mathcal{A}_{AEA}, pc, succpc, s) =$   
 $\exists gen\ kill\ in\ out\ gen'\ kill'\ in'\ out'.$   
 $\mathcal{A}_{AEA}(pc) = \text{Some}(gen, kill, in, out) \wedge$   
 $\mathcal{A}_{AEA}(succpc) = \text{Some}(gen', kill', in', out') \wedge$   
 $out = gen \cup (in - kill) \wedge$   
 $in' \subseteq out \wedge$   
 $\forall (v, e) \in out. \exists val. \text{evale}(v, s) = (\text{AB}_{\text{ok}}, \text{Some}(val)) \wedge \text{evale}(e, s) = (\text{AB}_{\text{ok}}, \text{Some}(val))$

With the definition of the function  $\text{confaesets}$  at hand, we can give the definition of a function  $\text{bisimrel}_{\text{RAE}}$  which computes the bisimulation relation  $\mathcal{R}_{\text{RAE}}$ . The definition of the function consists of two parts: The first part defines three auxiliary functions:  $\text{bisimrel\_RAE\_normblk}$ ,  $\text{bisimrel\_RAE\_excbk}$ , and  $\text{bisimrel\_RAE\_exitblk}$ , which compute subsets of  $\mathcal{R}_{\text{RAE}}$  for respective modes of executions mentioned in the beginning of this section. The second part gives the definition of the function  $\text{bisimrel}_{\text{RAE}}$  itself which computes the bisimulation relation  $\mathcal{R}_{\text{RAE}}$  as a union of three disjoint sets computed by the respective auxiliary functions.

Definition 7.79 defines a function  $\text{bisimrel\_RAE\_normblk}$  computes a subset of the bisimulation relation  $\mathcal{R}_{\text{RAE}}$  which is defined for the normal mode of execution w.r.t. two IL<sup>9</sup> programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$ . A configuration pair  $(\sigma''_S, \sigma''_T)$  is in the subset  $\text{bisimrel\_RAE\_normblk}(S, T, B, CBEP, \mathcal{A}_{AEA})$  iff the following holds:

- $\sigma''_S$  and  $\sigma''_T$  are produced by partial executions of  $(S, B)$  and  $(T, B)$ , respectively. Both executions have the same length and are performed in the normal mode.
- All corresponding components of  $\sigma''_S$  and  $\sigma''_T$  are equal.
- The value of the block status and the predecessor block components  $\sigma''_S$  and  $\sigma''_T$  have the syntactic forms  $\text{NORMBLCK}(pid, bid, 1, 0, pc)$  and  $predbid$  which means that in both programs the flow of control made transition along the block edge  $(predbid, bid)$ .  $\sigma''_S$  and  $\sigma''_T$  are in the subset if the relation  $CBEP$  declares the block edges  $(predbid, bid)$  and  $(predbid, bid)$  in  $(S, B)$  and  $(T, B)$ , respectively, as corresponding.
- The predecessor of the program point  $pc$  is  $predpc$ . The state components  $s$  in both configurations conform with the AEA result  $\mathcal{A}_{AEA}$  w.r.t. the CFG edge  $(predpc, pc)$ .

**Definition 7.79.**

```

bisimrel_RAE_normblk : Program × Program × BlkPosEnv × CorrespBlockEdgePairSet
                        × AESetsEnv → BisimulationRelation

bisimrel_RAE_normblk(S, T, B, CBEP,  $\mathcal{A}_{AEA}$ ) =
  let
    (pid0, BP, BB, succBP, predB) = B
  in
    {( $\sigma''_S, \sigma''_T$ ) |  $\exists bs ss pid bid pc predbid b s' bt set.$ 
       $\sigma''_S = (ss, bs, ss, NORMBLCK(pid, bid, 1, 0, pc), predbid, (NT, AB_{ok}, pc, b, s)) \wedge$ 
       $\sigma''_T = (ss, bs, ss, NORMBLCK(pid, bid, 1, 0, pc), predbid, (NT, AB_{ok}, pc, b, s)) \wedge$ 
       $M''((S, B), bs) = \sigma''_S \wedge$ 
       $M''((T, B), bs) = \sigma''_T \wedge$ 
       $BB(bid) = Some(pid) \wedge$ 
       $BP(pid) = Some(pid, bid, 1, 0, pc) \wedge$ 
       $predB(pid) = Some(set) \wedge$ 
       $(predbid, bt) \in set \wedge$ 
       $BB(predbid) = Some(predpid) \wedge$ 
       $BP(predpid) = Some(predpid, predbid, 1, 0, predpc) \wedge$ 
       $confbuffer(bt, b) \wedge$ 
       $((predbid, bid), (predbid, bid), bt) \in CBEP \wedge$ 
       $confaesets(\mathcal{A}_{AEA}, predpc, pc, s) \}$ 
    }
  end

```

Definition 7.80 defines a function `bisimrel_RAE_excblk` computes a subset of the bisimulation relation  $\mathcal{R}_{RAE}$  which is defined for the exception mode of execution and w.r.t. two IL<sup>n</sup> programs,  $(S, B)$  and  $(T, B)$ .

**Definition 7.80.**

```

bisimrel_RAE_excblk : Program × Program × BlkPosEnv → BisimulationRelation

bisimrel_RAE_excblk(S, T, B) =
  {( $\sigma''_S, \sigma''_T$ ) |  $\exists bs ss pc predbid n s s'.$ 
     $\sigma''_S = (ss, bs, ss, EXCBLCK, predbid, (NT, AB, pc, FLUSH(n), s)) \wedge$ 
     $\sigma''_T = (ss, bs, ss, EXCBLCK, predbid, (NT, AB, pc, FLUSH(n), s)) \wedge$ 
     $M''((S, B), bs) = \sigma''_S \wedge$ 
     $M''((T, B), bs) = \sigma''_T \}$ 
  }

```

Definition 7.81 defines a function `bisimrel_RAE_exitblk` computes a subset of the bisimulation relation  $\mathcal{R}_{RAE}$  which is defined for the exit mode of execution and w.r.t. two IL<sup>n</sup> programs,  $(S, B)$  and  $(T, B)$ .

**Definition 7.81.**

```

bisimrel_RAE_exitblk : Program × Program × BlkPosEnv → BisimulationRelation

bisimrel_RAE_exitblk(S, T, B) =
  {( $\sigma''_S, \sigma''_T$ ) |  $\exists bs ss pc predbid n s s'.$ 
     $\sigma''_S = (ss, bs, ss, EXITBLCK, predbid, (T, AB_{ok}, pc, FLUSH(n), s)) \wedge$ 
     $\sigma''_T = (ss, bs, ss, EXITBLCK, predbid, (T, AB_{ok}, pc, FLUSH(n), s)) \wedge$ 
     $M''((S, B), bs) = \sigma''_S \wedge$ 
     $M''((T, B), bs) = \sigma''_T \}$ 
  }

```

Definition 7.82 defines a function  $\text{bisimrel}_{\text{RAE}}$  which computes the bisimulation relation  $\mathcal{R}_{\text{RAE}}$  for two IL” programs,  $(S, B)$  and  $(T, B)$ , a corresponding block edges relation  $CBEP$ , and a AEA result  $\mathcal{A}_{\text{AEA}}$ .

The function  $\text{bisimrel}_{\text{RAE}}$  makes the following assumptions:

- $(S, B)$  is the source program of a RAE optimization.
- The compiler performed the AEA on  $(S, B)$ .
- The result of the analysis is  $\mathcal{A}_{\text{AEA}}$ .
- The result of the RAE optimization with  $(S, B)$  and  $\mathcal{A}_{\text{AEA}}$  as input is the target program  $(T, B)$ .

The bisimulation relation  $\mathcal{R}_{\text{RAE}}$  is defined as a union of three disjoint sets which are computed by the functions  $\text{bisimrel\_RAE\_normblk}$ ,  $\text{bisimrel\_RAE\_excbk}$ , and  $\text{bisimrel\_RAE\_exitblk}$ , for the respective modes of execution.

**Definition 7.82.**

$$\begin{aligned} \text{bisimrel}_{\text{RAE}} : \text{Program} \times \text{Program} \times \text{BlkPosEnv} \times \text{CorrespBlockEdgePairSet} \\ \times \text{AESetsEnv} \rightarrow \text{BisimulationRelation} \\ \text{bisimrel}_{\text{RAE}}(S, T, B, CBEP, \mathcal{A}_{\text{AEA}}) = \text{bisimrel\_RAE\_normblk}(S, T, B, CBEP, \mathcal{A}_{\text{AEA}}) \cup \\ \text{bisimrel\_RAE\_excbk}(S, T, B) \cup \\ \text{bisimrel\_RAE\_exitblk}(S, T, B) \end{aligned}$$

### 7.5.3 Optimization correctness criterion for the RAE optimization

This section presents the formalization of a translation relation predicate  $\text{TCC}_{\text{RAE}}$  on two IL” programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{\text{AEA}}$ , and a corresponding block edges relation  $CBEP$ . The parameters of  $\text{TCC}_{\text{RAE}}$  relate to a RAE optimization and denote the following:

- $(S, B)$  denotes the source program of the RAE optimization.
- $(T, B)$  denotes the target program of the RAE optimization.
- $\mathcal{A}_{\text{AEA}}$  is the result of the AEA with which was performed by the compiler prior to the REA optimization with  $S$  as input,
- $CBEP$  is a corresponding block edges relation which determines which blocks in the IL” programs  $(S, B)$  and  $(T, B)$  are corresponding.

Informally,  $\text{TCC}_{\text{RAE}}(S, T, B, \mathcal{A}_{\text{AEA}}, CBEP)$  holds true iff  $(T, B)$  is a correct RAE optimization of  $(S, B)$  w.r.t.  $\mathcal{A}_{\text{AEA}}$  and  $CBEP$ .

In the following, we call the  $\text{TCC}_{\text{RAE}}$  predicate an *optimization correctness criterion for RAE optimizations*. In our implementation, the  $\text{TCC}_{\text{RAE}}$  predicate is an instance of the optimization correctness criterion  $\text{OptCC}_O$  that was described in the overview of Layer 5 in Section 3.6

The rest of this section consists of three parts which are organized as follows.

- The first part gives formation rules for the set of relations over instruction pairs, **InstrTransRel\_RAE**, which is associated with the definition of the optimization correctness criterion  $\text{TCC}_{\text{RAE}}$ .
- The second part presents the definition of a function  $\text{rae\_transrel\_instr}$  which computes a RAE optimization relation over instruction pairs  $(instr, instr')$  for two programs which are the source and the target of a RAE optimization and a result of the AEA which was performed by the compiler prior to this optimization.
- The third part presents the definition of the optimization correctness criterion  $\text{TCC}_{\text{RAE}}$  itself.

We begin the presentation by introducing a set of translation relations over instruction pairs **InstrTransRel\_RAE**. Definition 7.83 gives formation rule for this set.

**Definition 7.83.**

$$\mathbf{InstrTransRel\_RAE} = \mathcal{P}(\mathbf{Instruction} \times \mathbf{Instruction})$$

The second part of the formalization of the  $\text{TCC}_{\text{RAE}}$  criteria comprises the definition of a function `rae_transrel_instr` which computes a translation relation over instruction pairs for two IL" program  $(S, B)$  and  $(T, B)$ , an AEA result  $\mathcal{A}_{\text{AEA}}$ , a corresponding block edges relation  $\text{CBEP}$ , a block  $\text{bid}$ , a block position  $\text{pid}$ , and a program point  $\text{pc}$ . The translation relation  $\text{rae\_transrel\_instr}(S, T, B, \text{CBEP}, \mathcal{A}_{\text{AEA}}, \text{bid}, \text{pid}, \text{pc})$  is defined as a union of eight disjoint sets which arise from eight syntactic optimization patterns which are possible for a pair of blocks which are declared as corresponding in  $(S, B)$  and  $(T, B)$  by the triple  $(\text{bid}, \text{pid}, \text{pc})$ . These sets are computed by respective auxiliary functions presented below in Definitions 7.84, 7.85, 7.86, 7.87, 7.88, 7.89, 7.90, 7.91, and 7.92. The auxiliary functions have the same parameters as the function `rae_transrel_instr` and make the following context assumptions these parameters:

1. The compiler performed the AEA on an IL program  $S$ . The result of this analysis is  $\mathcal{A}_{\text{AEA}}$ .
2. The compiler performed the AEA optimization on  $S$ . The result of this optimization is a target program  $T$ .
3. The compiler generated a CFGB declaration  $B$  which is identical for both  $S$  and  $T$ , and all blocks in the CFGB declared by  $B$  have the length equal one.
4. The compiler generated a corresponding block edges relation  $\text{CBEP}$  that defines, among other things, which blocks in  $(S, B)$  and  $(T, B)$  are corresponding.
5. In both  $(S, B)$  and  $(T, B)$ , there exists a program point  $\text{pc}$  which is allocated to a block position  $\text{pid}$  which is included in a block  $\text{bid}$ .

Then, an auxiliary function computes a subset of the RAE optimization relation

**InstrTransRel\_RAE** such that each pair  $(\text{instr}, \text{instr}')$  in this subset consists of two instructions  $\text{instr}$  and  $\text{instr}'$  which are the  $\text{pc}$ -th instruction of  $S$  and the  $\text{pc}$ -th instruction of  $T$ , respectively, and they satisfy a particular optimization pattern which is specified by the function.

Definition 7.84 defines a function `rae_transrel_instr_case1` which computes a subset of a RAE optimization relation for two IL" programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{\text{AEA}}$ , and a corresponding block edges relation  $\text{CBEP}$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $\text{bid}$ , a block position  $\text{pid}$ , and a program point  $\text{pc}$ . The following holds for the relation `rae_transrel_instr_case1` $(S, T, B, \text{CBEP}, \mathcal{A}_{\text{AEA}}, \text{bid}, \text{pid}, \text{pc})$ :

1. Each instruction pair  $(\text{instr}, \text{instr}')$  in this relation consists of two assignments to a variable which are equal, i.e. has the syntactic form  $(v:=e, v:=e)$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $\text{pc}$  is allocated to a block position  $\text{pid}$  which is included by a block  $\text{bid}$  which has the length equal one.
3. The CFG's of both  $S$  and  $T$  comprise an edge  $(\text{pc}, \text{pc} + 1)$ , i.e.  $\text{pc} + 1$  is the successor of  $\text{pc}$  in both  $S$  and  $T$ .
4. In both  $(S, B)$  and  $(T, T)$ , the program point  $\text{pc} + 1$  is allocated to a block position  $\text{pid}'$  which is included by a block  $\text{bid}'$  which has the length equal one.
5. The blocks  $\text{bid}$  and  $\text{bid}'$  are declared as corresponding by the corresponding block edges relation  $\text{CBEP}$ .
6. The available equation sets in tuples which are declared by  $\mathcal{A}_{\text{AEA}}$  for the program points  $\text{pc}$  and  $\text{pc} + 1$  are well-formed: the "gen" and the "kill" are consistent with the operational

semantics of the assignment  $v:=e$  and the "in" and "out" sets are a proper solution of data flow equations for the AEA defined for the program  $S$ .

7. The equation  $(v, e)$  is not available at the entry to the CFG node  $pc$ .

**Definition 7.84.**

```

rae_transrel_instr_case1 : Program × Program × BlckPosEnv × CorrespBlockEdgePairSet
    × AESetsEnv × BlckId × BlckPosId × InstructionNr
    → InstrTransRel_RAE
rae_transrel_instr_case1(S, T, B, CBEP, AAEA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(v:=e, v:=e) | ∃ bid' pid' pc' set gen kill in out gen' kill' in' out'.
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      BB(bid) = Some(pid) ∧
      instrs!pc = (v:=e) ∧
      instrs'!pc = (v:=e) ∧
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
      AAEA(pc) = Some(gen, kill, in, out) ∧
      AAEA(pc') = Some(gen', kill', in', out') ∧
      gen = instr2gen(S, instrs!pc) ∧
      kill = instr2kill(S, instrs!pc) ∧
      in ⊆ prg2Uset(S) ∧
      out = gen ∪ (in - kill) ∧
      in' ⊆ out ∧
      (v, e) ∈ in ∧
      ((bid, bid'), (bid, bid'), FTYPE) ∈ CBEP }
  end

```

Definition 7.85 defines a function `rae_transrel_instr_case2` which computes a subset of a RAE optimization relation for two IL<sup>n</sup> programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `rae_transrel_instr_case2` $(S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation consists of an assignment to a variable and a goto instruction which emulates a nop instruction, i.e. has the syntactic form  $(v:=e, \text{goto}(dst))$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc$  is allocated to a block position  $pid$  which is included by a block  $bid$  which has the length equal one.
3. The value of the destination  $dst$  in the goto instruction in the pair is equal  $pc + 1$ .
4. The CFG's of both  $S$  and  $T$  comprise an edge  $(pc, pc + 1)$ , i.e.  $pc + 1$  is the successor of  $pc$  in both  $S$  and  $T$ .
5. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which has the length equal one.

6. The blocks  $bid$  and  $bid'$  are declared as corresponding by the corresponding block edges relation  $CBEP$ .
7. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program points  $pc$  and  $pc + 1$  are well-formed: the "gen" and the "kill" are consistent with the operational semantics of the assignment  $v:=e$  and the "in" and "out" sets are a proper solution of data flow equations for the AEA defined for the program  $S$ .
8. The equation  $(v, e)$  is available at the entry to the CFG node  $pc$ .

**Definition 7.85.**

```

rae_transrel_instr_case2 : Program × Program × BlckPosEnv × CorrespBlockEdgePairSet
                          × AESetsEnv × BlckId × BlckPosId × InstructionNr
                          → InstrTransRel_RAE
rae_transrel_instr_case2(S, T, B, CBEP,  $\mathcal{A}_{AEA}$ , bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(v:=e, goto(pc')) | ∃ bid' pid' pc' set gen kill in out gen' kill' in' out'.
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      BB(bid) = Some(pid) ∧
      instrs!pc = (v:=e) ∧
      instrs'!pc = (goto(pc')) ∧
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, FTYPE) ∈ set ∧
       $\mathcal{A}_{AEA}(pc) = \text{Some}(gen, kill, in, out) \wedge$ 
       $\mathcal{A}_{AEA}(pc') = \text{Some}(gen', kill', in', out') \wedge$ 
      gen = instr2gen(S, instrs!pc) ∧
      kill = instr2kill(S, instrs!pc) ∧
      in ⊆ prg2Uset(S) ∧
      out = gen ∪ (in - kill) ∧
      in' ⊆ out ∧
      (v, e) ∈ in ∧
      ((bid, bid'), (bid, bid'), FTYPE) ∈ CBEP }
  end

```

Definition 7.86 defines a function `rae_transrel_instr_case3` which computes a subset of a RAE optimization relation for two IL<sup>9</sup> programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `rae_transrel_instr_case3` $(S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation has the syntactic form  $(a[i]:=e, a[i]:=e)$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc$  is allocated to a block position  $pid$  which is included by a block  $bid$  which has the length equal one.
3. The CFG's of both  $S$  and  $T$  comprise an edge  $(pc, pc + 1)$ , i.e.  $pc + 1$  is the successor of  $pc$  in both  $S$  and  $T$ .

4. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which has the length equal one.
5. The blocks  $bid$  and  $bid'$  are declared as corresponding by the corresponding block edges relation  $CBEP$ .
6. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program points  $pc$  and  $pc + 1$  are well-formed: the "gen" and the "kill" are consistent with the operational semantics of the assignment  $a[i]:=e$  and the "in" and "out" sets are a proper solution of data flow equations for the AEA defined for the program  $S$ .

**Definition 7.86.**

```

rae_transrel_instr_case3 : Program × Program × BlkPosEnv × CorrespBlockEdgePairSet
                          × AESetsEnv × BlkId × BlkPosId × InstructionNr
                          → InstrTransRel_RAE
rae_transrel_instr_case3( $S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc$ ) =
  let
     $((vds, instrs), I) = S;$ 
     $((vds', instrs'), I') = T;$ 
     $(pid_0, BP, BB, succBP, predB) = B$ 
  in
    {  $(a[i]:=e, a[i]:=e) \mid \exists bid' pid' pc' set gen kill in out gen' kill' in' out'.$ 
       $BP(pid) = \text{Some}(pid, bid, 1, 0, pc) \wedge$ 
       $BB(bid) = \text{Some}(pid) \wedge$ 
       $instrs!pc = (a[i]:=e) \wedge$ 
       $instrs'!pc = (a[i]:=e) \wedge$ 
       $pc' = \text{Suc}(pc) \wedge$ 
       $succBP(pid, pc') = \text{Some}(pid') \wedge$ 
       $BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge$ 
       $BB(bid') = \text{Some}(pid') \wedge$ 
       $predB(pid') = \text{Some}(set) \wedge$ 
       $(bid, \text{FTYPE}) \in set \wedge$ 
       $\mathcal{A}_{AEA}(pc) = \text{Some}(gen, kill, in, out) \wedge$ 
       $\mathcal{A}_{AEA}(pc') = \text{Some}(gen', kill', in', out') \wedge$ 
       $gen = \text{instr2gen}(S, instrs!pc) \wedge$ 
       $kill = \text{instr2kill}(S, instrs!pc) \wedge$ 
       $in \subseteq \text{prg2Uset}(S) \wedge$ 
       $out = gen \cup (in - kill) \wedge$ 
       $in' \subseteq out \wedge$ 
       $((bid, bid'), (bid, bid'), \text{FTYPE}) \in CBEP \}$ 
  end

```

Definition 7.87 defines a function `rae_transrel_instr_case4` which computes a subset of a RAE optimization relation for two IL<sup>2</sup> programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `rae_transrel_instr_case4` $(S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation has the syntactic form  $(a[v]:=e, a[v]:=e)$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc$  is allocated to a block position  $pid$  which is included by a block  $bid$  which has the length equal one.
3. The CFG's of both  $S$  and  $T$  comprise an edge  $(pc, pc + 1)$ , i.e.  $pc + 1$  is the successor of  $pc$  in both  $S$  and  $T$ .

4. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which has the length equal one.
5. The blocks  $bid$  and  $bid'$  are declared as corresponding by the corresponding block edges relation  $CBEP$ .
6. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program points  $pc$  and  $pc + 1$  are well-formed: the "gen" and the "kill" are consistent with the operational semantics of the assignment  $a[v]:=e$  and the "in" and "out" sets are a proper solution of data flow equations for the AEA defined for the program  $S$ .

**Definition 7.87.**

```

rae_transrel_instr_case4 : Program × Program × BlkPosEnv × CorrespBlockEdgePairSet
    × AESetsEnv × BlkId × BlkPosId × InstructionNr
    → InstrTransRel_RAE
rae_transrel_instr_case4( $S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc$ ) =
  let
     $((vds, instrs), I) = S;$ 
     $((vds', instrs'), I') = T;$ 
     $(pid_0, BP, BB, succBP, predB) = B$ 
  in
    { $(a[v]:=e, a[v]:=e) \mid \exists bid' pid' pc' set gen kill in out gen' kill' in' out'.$ 
       $BP(pid) = \text{Some}(pid, bid, 1, 0, pc) \wedge$ 
       $BB(bid) = \text{Some}(pid) \wedge$ 
       $instrs!pc = (a[v]:=e) \wedge$ 
       $instrs'!pc = (a[v]:=e) \wedge$ 
       $pc' = \text{Suc}(pc) \wedge$ 
       $succBP(pid, pc') = \text{Some}(pid') \wedge$ 
       $BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge$ 
       $BB(bid') = \text{Some}(pid') \wedge$ 
       $predB(pid') = \text{Some}(set) \wedge$ 
       $(bid, \text{FTYPE}) \in set \wedge$ 
       $\mathcal{A}_{AEA}(pc) = \text{Some}(gen, kill, in, out) \wedge$ 
       $\mathcal{A}_{AEA}(pc') = \text{Some}(gen', kill', in', out') \wedge$ 
       $gen = \text{instr2gen}(S, instrs!pc) \wedge$ 
       $kill = \text{instr2kill}(S, instrs!pc) \wedge$ 
       $in \subseteq \text{prg2Uset}(S) \wedge$ 
       $out = gen \cup (in - kill) \wedge$ 
       $in' \subseteq out \wedge$ 
       $((bid, bid'), (bid, bid'), \text{FTYPE}) \in CBEP$  }
  end

```

Definition 7.88 defines a function `rae_transrel_instr_case5` which computes a subset of a RAE optimization relation for two IL<sup>2</sup> programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `rae_transrel_instr_case5( $S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc$ )`:

1. Each instruction pair  $(instr, instr')$  in this relation has the syntactic form  $(\text{printi}(e), \text{printi}(e))$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc$  is allocated to a block position  $pid$  which is included by a block  $bid$  which has the length equal one.
3. The CFG's of both  $S$  and  $T$  comprise an edge  $(pc, pc + 1)$ , i.e.  $pc + 1$  is the successor of  $pc$  in both  $S$  and  $T$ .

4. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which has the length equal one.
5. The blocks  $bid$  and  $bid'$  are declared as corresponding by the corresponding block edges relation  $CBEP$ .
6. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program points  $pc$  and  $pc + 1$  are well-formed: the "gen" and the "kill" are consistent with the operational semantics of the printi instruction  $\text{printi}(e)$  and the "in" and "out" sets are a proper solution of data flow equations for the AEA defined for the program  $S$ .

**Definition 7.88.**

```

rae_transrel_instr_case5 : Program × Program × BlckPosEnv × CorrespBlockEdgePairSet
                          × AESetsEnv × BlckId × BlckPosId × InstructionNr
                          → InstrTransRel_RAE
rae_transrel_instr_case5(S, T, B, CBEP,  $\mathcal{A}_{AEA}$ , bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(printi(e), printi(e)) | ∃ bid' pid' pc' set gen kill in out gen' kill' in' out'.
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      BB(bid) = Some(pid) ∧
      instrs!pc = (printi(e)) ∧
      instrs'!pc = (printi(e)) ∧
      pc' = Suc(pc) ∧
      succBP(pid, pc') = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set) ∧
      (bid, 0TYPE) ∈ set ∧
       $\mathcal{A}_{AEA}(pc)$  = Some(gen, kill, in, out) ∧
       $\mathcal{A}_{AEA}(pc')$  = Some(gen', kill', in', out') ∧
      gen = instr2gen(S, instrs!pc) ∧
      kill = instr2kill(S, instrs!pc) ∧
      in ⊆ prg2Uset(S) ∧
      out = gen ∪ (in - kill) ∧
      in' ⊆ out ∧
      ((bid, bid'), (bid, bid'), 0TYPE) ∈ CBEP }
  end

```

Definition 7.89 defines a function `rae_transrel_instr_case6` which computes a subset of a RAE optimization relation for two IL<sup>2</sup> programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `rae_transrel_instr_case6`( $S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc$ ):

1. Each instruction pair  $(instr, instr')$  in this relation has the syntactic form  $(\text{branch}(e, dst), \text{branch}(e, dst))$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc$  is allocated to a block position  $pid$  which is included by a block  $bid$  which has the length equal one.
3. The CFG's of both  $S$  and  $T$  comprise edges  $(pc, pc + 1)$  and  $(pc, dst)$ , i.e.  $pc + 1$  and  $dst$  are the successors of  $pc$  in both  $S$  and  $T$ .

4. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc + 1$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which has the length equal one.
5. In both  $(S, B)$  and  $(T, T)$ , the program point  $dst$  is allocated to a block position  $pid''$  which is included by a block  $bid''$  which has the length equal one.
6. The blocks  $bid$  and  $bid'$  are declared as corresponding by the corresponding block edges relation  $CBEP$ .
7. The blocks  $bid$  and  $bid''$  are declared as corresponding by the corresponding block edges relation  $CBEP$ .
8. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program points  $pc$ ,  $pc + 1$ , and  $dst$  are well-formed: the "gen" and the "kill" are consistent with the operational semantics of the branch instruction  $\mathbf{branch}(e, dst)$  and the "in" and "out" sets are a proper solution of data flow equations for the AEA defined for the program  $S$ .

**Definition 7.89.**

```

rae_transrel_instr_case6 : Program × Program × BlckPosEnv × CorrespBlockEdgePairSet
    × AEsSetsEnv × BlckId × BlckPosId × InstructionNr
    → InstrTransRel_RAE
rae_transrel_instr_case6(S, T, B, CBEP, AAEA, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(branch(e, dst), branch(e, dst)) |
      ∃ bid' pid' pc' set gen kill in out gen' kill' in' out' gen'' kill'' in'' out''.
        BP(pid) = Some(pid, bid, 1, 0, pc) ∧
        BB(bid) = Some(pid) ∧
        instrs!pc = (branch(e, dst)) ∧
        instrs'!pc = (branch(e, dst)) ∧
        pc' = Suc(pc) ∧
        succBP(pid, pc') = Some(pid') ∧
        BP(pid') = Some(pid', bid', 1, 0, pc') ∧
        BB(bid') = Some(pid') ∧
        predB(pid') = Some(set') ∧
        (bid, FTYPE) ∈ set' ∧
        succBP(pid, dst) = Some(pid'') ∧
        BP(pid'') = Some(pid'', bid'', 1, 0, dst) ∧
        BB(bid'') = Some(pid'') ∧
        predB(pid'') = Some(set'') ∧
        (bid, FTYPE) ∈ set'' ∧
        AAEA(pc) = Some(gen, kill, in, out) ∧
        gen = instr2gen(S, instrs!pc) ∧
        kill = instr2kill(S, instrs!pc) ∧
        in ⊆ prg2Uset(S) ∧
        out = gen ∪ (in - kill) ∧
        AAEA(pc') = Some(gen', kill', in', out') ∧
        in' ⊆ out ∧
        AAEA(dst) = Some(gen'', kill'', in'', out'') ∧
        in'' ⊆ out ∧
        ((bid, bid'), (bid, bid'), FTYPE) ∈ CBEP ∧
        ((bid, bid''), (bid, bid''), FTYPE) ∈ CBEP }
  end

```

Definition 7.90 defines a function `rae_transrel_instr_case7` which computes a subset of a RAE optimization relation for two IL<sup>n</sup> programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `rae_transrel_instr_case7` $(S, T, B, CBEP, \mathcal{A}_{AEA}, bid, pid, pc)$ :

1. Each instruction pair  $(instr, instr')$  in this relation has the syntactic form  $(goto(dst), goto(dst))$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc$  is allocated to a block position  $pid$  which is included by a block  $bid$  which has the length equal one.
3. The CFG's of both  $S$  and  $T$  comprise an edge  $(pc, dst)$ , i.e.  $dst$  is the successor of  $pc$  in both  $S$  and  $T$ .

4. In both  $(S, B)$  and  $(T, T)$ , the program point  $dst$  is allocated to a block position  $pid'$  which is included by a block  $bid'$  which has the length equal one.
5. The blocks  $bid$  and  $bid'$  are declared as corresponding by the corresponding block edges relation  $CBEP$ .
6. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program points  $pc$  and  $dst$  are well-formed: the "gen" and the "kill" are consistent with the operational semantics of the goto instruction  $goto(dst)$  and the "in" and "out" sets are a proper solution of data flow equations for the AEA defined for the program  $S$ .

**Definition 7.90.**

```

rae_transrel_instr_case7 : Program × Program × BlckPosEnv × CorrespBlockEdgePairSet
                          × AESetsEnv × BlckId × BlckPosId × InstructionNr
                          → InstrTransRel_RAE
rae_transrel_instr_case7(S, T, B, CBEP,  $\mathcal{A}_{AEA}$ , bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(goto(dst), goto(dst)) | ∃ bid' pid' pc' set gen kill in out gen' kill' in' out'.
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      BB(bid) = Some(pid) ∧
      instrs!pc = (goto(dst)) ∧
      instrs'!pc = (goto(dst)) ∧
      succBP(pid, dst) = Some(pid') ∧
      BP(pid') = Some(pid', bid', 1, 0, pc') ∧
      BB(bid') = Some(pid') ∧
      predB(pid') = Some(set') ∧
      (bid, FTYPE) ∈ set' ∧
       $\mathcal{A}_{AEA}(pc)$  = Some(gen, kill, in, out) ∧
      gen = instr2gen(S, instrs!pc) ∧
      kill = instr2kill(S, instrs!pc) ∧
      in ⊆ prg2Uset(S) ∧
      out = gen ∪ (in - kill) ∧
       $\mathcal{A}_{AEA}(dst)$  = Some(gen', kill', in', out') ∧
      in' ⊆ out ∧
      ((bid, bid'), (bid, bid'), FTYPE) ∈ CBEP }
  end

```

Definition 7.91 defines a function `rae_transrel_instr_case8` which computes a subset of a RAE optimization relation for two IL<sup>''</sup> programs,  $(S, B)$  and  $(T, B)$ , w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . The following holds for the relation `rae_transrel_instr_case8(S, T, B, CBEP,  $\mathcal{A}_{AEA}$ , bid, pid, pc)`:

1. Each instruction pair  $(instr, instr')$  in this relation has the syntactic form  $(exit, exit)$ .
2. In both  $(S, B)$  and  $(T, T)$ , the program point  $pc$  is allocated to a block position  $pid$  which is included by a block  $bid$  which has the length equal one.

**Definition 7.91.**

```

rae_transrel_instr_case8 : Program × Program × BlckPosEnv × BlckId × BlckPosId
    × InstructionNr → InstrTransRel_RAE
rae_transrel_instr_case8(S, T, B, bid, pid, pc) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    {(exit, exit) | ∃ bid' pid' pc' set gen kill in out gen' kill' in' out' gen'' kill'' in'' out'' .
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      BB(bid) = Some(pid) ∧
      instrs!pc = exit ∧
      instrs'!pc = exit }
end

```

Definition 7.92 defines a function `rae_transrel_instr` which computes the RAE optimization relation for two IL programs,  $(S, B)$  and  $(T, B)$ , a AEA result  $\mathcal{A}_{AEA}$ , and a corresponding block edges relation  $CBEP$  w.r.t. a pair of corresponding block positions in  $(S, B)$  and  $(T, B)$  described by a block  $bid$ , a block position  $pid$ , and a program point  $pc$ . As explained in the motivation to the definition of the function `rae_transrel_instr`, the optimization relation  $rae\_transrel\_instr(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc)$  is defined as a union of eight disjoint sets which arise from eight syntactic patterns.

**Definition 7.92.**

```

rae_transrel_instr : Program × Program × BlckPosEnv × AESetsEnv
    × CorrespBlockEdgePairSet × BlckId × BlckPosId × InstructionNr
    → InstrTransRel_RAE
rae_transrel_instr(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) =
  rae_transrel_instr_case1(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) ∪
  rae_transrel_instr_case2(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) ∪
  rae_transrel_instr_case3(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) ∪
  rae_transrel_instr_case4(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) ∪
  rae_transrel_instr_case5(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) ∪
  rae_transrel_instr_case6(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) ∪
  rae_transrel_instr_case7(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc) ∪
  rae_transrel_instr_case8(S, T, B, bid, pid, pc)

```

Finally, with the definitions of the conformance predicate `confaesets` and the function `rae_transrel_instr`, we can give the definition of the optimization correctness criterion  $TCC_{RAE}$  on two IL programs,  $S$  and  $T$ , a CFGB declaration  $B$ , a AEA result  $\mathcal{A}_{AEA}$ , a corresponding block edges relation  $CBEP$  which formalizes what does it mean that  $T$  is a correct RAE optimization of  $S$  w.r.t.  $B, \mathcal{A}_{AEA}$ , and  $CBEP$ .

The definition of the optimization correctness criterion  $TCC_{RAE}$  makes the context assumptions about its parameters which are analogous to assumptions made by the definition of the function `rae_transrel_instr`:

1. The compiler performed the AEA on an IL program  $S$ . The result of this analysis is  $\mathcal{A}_{AEA}$ .
2. The compiler performed the AEA optimization on  $S$ . The result of this optimization is a target program  $T$ .

3. The compiler generated a CFGB declaration  $B$  which is identical for both  $S$  and  $T$ , and all blocks in the CFGB declared by  $B$  have the length equal one.
4. The compiler generated a corresponding block edges relation  $CBEP$  that defines, among other things, which blocks in  $(S, B)$  and  $(T, B)$  are corresponding.

Analogously to the definitions of the criteria  $TCC_{CF}$ ,  $TCC_{DAE}$ , and  $TCC_{NI}$ , the definition of  $TCC_{RAE}$  consists of two conjuncts: The first conjunct formalizes a well-formedness criterion for the entry block of the CFGB declaration  $(T, B)$  and the AEA result  $\mathcal{A}_{AEA}$  w.r.t. the source program  $S$ . The second conjunct formalizes an optimization correctness criterion on the set of pairs consisting of corresponding blocks in IL" programs  $(S, B)$  and  $(T, B)$ .

Definition 7.93 defines a function  $TCC\_RAE\_entry\_block$  which checks if the entry block of the IL" program  $(S, B)$  fulfills the following well-formedness criteria:

1. The entry block of a CFGB declaration  $B$ ,  $bid_0$ , includes the entry block position of  $B$ ,  $pid_0$ , and has the length equal one.
2. The program point 0 is allocated to  $pid_0$ .
3. The CFGB declaration declares  $bid_0$  as one of predecessors of  $bid_0$ .
4. The relation  $CBEP$  declares that both  $(S, B)$  and  $(T, B)$  have to declare the block edges  $(bid_0, bid_0)$  and  $(bid_0, bid_0)$ .
5. The available equation sets in tuples which are declared by  $\mathcal{A}_{AEA}$  for the program point 0 are well-formed: the "gen" and the "kill" are consistent with the operational semantics of the 0-th instruction of  $S$ . The "in" set is initialized as empty set and the "out" set is initialized as follows.

$$\begin{aligned} out &= gen \cup (in - kill) \\ &= instr2gen(S, instrs!0) \cup (\{\} - instr2kill(S, instrs!0)) \\ &= instr2gen(S, instrs!0) \end{aligned}$$

6. The initial state of the program  $S$ ,  $mapof(I)$ , conforms with the AEA result  $\mathcal{A}_{AEA}$  w.r.t. the pair of program points  $(0, 0)$ .

**Definition 7.93.**

```

TCC_RAE_entry_block : Program × BlckPosEnv × AESetsEnv × CorrespBlockEdgePairSet
                    → Bool
TCC_RAE_entry_block(S, B,  $\mathcal{A}_{AEA}$ , CBEP) =
  let
    ((vds, instrs), I) = S;
    (pid0, BP, BB, succBP, predB) = B
  in
    ∃ bid0 set gen kill, out in.
      BP(pid0) = Some(pid0, bid0, 1, 0, 0) ∧
      predB(pid0) = Some(set) ∧
      (bid0, FTYPE) ∈ set ∧
      ((bid0, bid0), (bid0, bid0), FTYPE) ∈ CBEP ∧
      gen = instr2gen(S, instrs!0) ∧
      kill = instr2kill(S, instrs!0) ∧
      in = {} ∧
      out = instr2gen(S, instrs!0) ∧
       $\mathcal{A}_{AEA}(0)$  = Some(gen, kill, in, out) ∧
      confaesets( $\mathcal{A}_{AEA}$ , 0, 0, mapof(I))
  end

```

Definition 7.94 defines the predicate  $\text{TCC\_RAE\_normal\_block}$  on a source program  $S$ , a target program  $T$ , a CFGB declaration  $B$ , a AEA result  $\mathcal{A}_{AEA}$ , a corresponding block edges relation  $CBEP$ , and a block  $bid$  which checks if the block  $bid$  in the CFGB  $(S, B)$  and the block  $bid$  in the CFGB  $(T, B)$  fulfill the following optimization correctness criterion for the block pair  $(bid, bid)$ :  $\text{TCC\_RAE\_normal\_block}(S, T, B, \mathcal{A}_{AEA}, CBEP, bid)$  holds true iff there exist  $pc$ -th program points  $pc$  in  $S$  and  $T$ ; and a block positions  $pid$  in  $(S, B)$  and  $(T, B)$  such that  $pc$  is allocated to  $pid$ ,  $pid$  is included by the block  $bid$ , and the pair  $(instrs!pc, instrs'!pc)$  consisting of the  $pc$ -th instructions of  $S$  and  $T$ , respectively, are in the RAE optimization relation  $\text{rae\_transrel\_instr}(S, T, B, \mathcal{A}_{AEA}, CBEP, bid, pid, pc)$ .

**Definition 7.94.**

```

TCC_RAE_normal_block : Program × Program × BlkPosEnv × AESetsEnv
                      × CorrespBlockEdgePairSet × BlkId → Bool
TCC_RAE_normal_block(S, T, B,  $\mathcal{A}_{AEA}$ , CBEP, bid) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BP, BB, succBP, predB) = B
  in
    ∃ pid pc.
      BB(bid) = Some(pid) ∧
      BP(pid) = Some(pid, bid, 1, 0, pc) ∧
      (instrs!pc, instrs'!pc) ∈ rae_transrel_instr(S, T, B,  $\mathcal{A}_{AEA}$ , CBEP, bid, pid, pc)
  end

```

Definition 7.95 defines our optimization correctness criterion for the RAE optimization on a source program  $S$ , a target program  $T$ , a CFGB declaration  $B$ , a AEA result  $\mathcal{A}_{AEA}$ , and corresponding block edges relation  $CBEP$  which checks if  $T$  is a correct RAE optimization of  $S$  w.r.t.  $VB$ ,  $CBEP$ , and  $\mathcal{A}_{AEA}$ . According to the definition of  $\text{TCC}_{RAE}$ , an IL program  $T$  is a correct RAE optimization of  $S$  iff

1. the entry blocks of IL<sup>2</sup> programs  $(S, B)$  and  $(T, B)$  fulfill the well-formedness criterion  $\text{TCC\_RAE\_entry\_block}$  w.r.t.  $\mathcal{A}_{AEA}$  and  $CBEP$ .
2. each block in  $(S, B)$  and  $(T, B)$  fulfills the criterion  $\text{TCC\_RAE\_normal\_block}(S, T, B, \mathcal{A}_{AEA}, CBEP)$ .

**Definition 7.95.**

```

TCCRAE : Program × Program × BlkPosEnv × AESetsEnv × CorrespBlockEdgePairSet
        → Bool
TCCRAE(S, T, B,  $\mathcal{A}_{AEA}$ , CBEP) = TCC_RAE_entry_block(S, B,  $\mathcal{A}_{AEA}$ , CBEP) ∧
                                ∀ bid. TCC_RAE_normal_block(S, T, B,  $\mathcal{A}_{AEA}$ , CBEP, bid)

```

#### 7.5.4 Verification of the optimization correctness criterion $\text{TCC}_{RAE}$

This section presents the theorems which we proved in order to verify the specification of the optimization correctness criterion  $\text{TCC}_{RAE}$  presented in the previous section. The main result in this section is a theorem which can be used directly in translation certificates generated by our compiler.

To verify the specification of the criterion  $\text{TCC}_{RAE}$ , we proved Theorem 7.96 which is a statement about a source and a target programs of a concrete RAE optimization,  $S$  and  $T$ , a program

type  $\Phi$ , a CFGB declaration  $B$ , a corresponding block edges relation  $CBEP$ , and a result of the AE analysis which was performed on  $S$  prior to that optimization,  $\mathcal{A}_{AEA}$ . The statement of the theorem says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B$  is well-formed w.r.t.  $S$  and  $T$ ; and  $S$  and  $T$  fulfill the optimization correctness criterion  $TCC_{RAE}$  w.r.t.  $B$ ,  $CBEP$ , and  $\mathcal{A}_{AEA}$ , then  $S$  and  $T$  fulfill the optimization independent translation correctness TCC w.r.t. the bisimulation relation  $\text{bisimrel}_{RAE}(S, T, B, CBEP, \mathcal{A}_{AEA})$ .

**Theorem 7.96.**

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge TCC_{RAE}(S, T, B, \mathcal{A}_{AEA}, CBEP) \\ & \implies \\ & TCC(S, T, \text{bisimrel}_{RAE}(S, T, B, CBEP, \mathcal{A}_{AEA})) \end{aligned}$$

□

Finally, we present the main result in this section, a theorem which is a statement about a source and a target programs of a concrete RAE optimization,  $S$  and  $T$ , a program type  $\Phi$ , a CFGB declaration  $B$ , a corresponding block edges relation  $CBEP$ , and a result of the AEA which was performed on  $S$  prior to that optimization,  $\mathcal{A}_{AEA}$ . The statement of the theorem says that if  $S$  and  $T$  are well-typed w.r.t.  $\Phi$ ; and  $B$  is well-formed w.r.t.  $S$  and  $T$ ; and  $S$  and  $T$  fulfill the optimization correctness criterion  $TCC_{RAE}$  w.r.t.  $B$ ,  $CBEP$ , and  $\mathcal{A}_{AEA}$ , then  $S$  and  $T$  fulfill the translation correctness predicate  $\text{corrTrans}$ .

**Theorem 7.97.**

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge TCC_{RAE}(S, T, B, \mathcal{A}_{AEA}, CBEP) \\ & \implies \\ & \text{corrTrans}(S, T) \end{aligned}$$

*Proof.*

$$\begin{aligned} & \text{wtp}(S, \Phi) \wedge \text{wtp}(T, \Phi) \wedge \text{wfB}(S, B) \wedge \text{wfB}(T, B) \wedge TCC_{RAE}(S, T, B, \mathcal{A}_{AEA}, CBEP) \\ & \implies \text{[by application of Theorem 7.96]} \\ & TCC(S, T, \text{bisimrel}_{RAE}(S, T, B, CBEP, \mathcal{A}_{AEA})) \\ & \implies \text{[by application of the existential introduction rule]} \\ & \exists \mathcal{R}. TCC(S, T, \mathcal{R}). \\ & \implies \text{[by application of Theorem 6.44]} \\ & \text{corrTrans}(S, T) \end{aligned}$$

□

Theorem 7.97 is an instance of the corollary (I) in Section 3.6 and is directly applicable in translation certificates which are generated by the compiler for each RAE optimization, see the end of Section 3.6 for a general application scheme of this theorem.



## Chapter 8

---

### Evaluation

This chapter presents evaluation of our FTV system. The evaluation is split into the following aspects the SVF:

1. the size of proof scripts generated by our compiler,
2. performance of proof script checking,
3. suitability of our SVF for certifying further optimizations.

#### 8.1 Proof script size

This section presents evaluation of the size of proof scripts generated by our compiler front-end. Our evaluation is based on the description of the proof script layout in Sections 3.7 and 3.6. As aforementioned in these sections, Layer 5 of the SVF provides for each optimization  $O$  an optimization correctness criterion  $\text{TCC}_O$  and an optimization correctness theorem that give raise to a uniform proof script layout for the optimization  $O$ . According to this layout, a proof script with the correctness proof of a concrete optimization  $O$  comprises thirteen parts that are divided in two sections: a constant definition section and a lemma section. The constant definition section comprises the following parts:

- Part 1.:* provides constant definition of the source program  $S$  of the optimization  $O$ ,
- Part 2.:* provides constant definition of the target program  $T$  of the optimization  $O$ ,
- Part 3.:* provides constant definition of a program type  $\Phi_S$  which is the program type of  $S$ ,
- Part 4.:* provides constant definition of a program type  $\Phi_T$  which is the program type of  $T$ .
- Part 5.:* provides constant definition of a CFGB declaration  $B_S$  for the program  $S$ .
- Part 6.:* provides constant definition of a CFGB declaration  $B_T$  for the program  $T$ .
- Part 7.:* provides constant definition of a result of data flow analysis  $\mathcal{A}_O$ ,

It holds for all optimizations supported by our SVF but the NI optimization that the source and the target program types  $\Phi_S$  and  $\Phi_T$  are equal. Therefore, the compiler generates proof scripts for these optimizations with Parts 3. and 4. merged into one part providing the constant definition of a program type  $\Phi$  which is declared for both  $S$  and  $T$ . The same holds for the source and the target CFGB declarations  $B_S$  and  $B_T$  which are equal for all optimizations but the NI optimization. Therefore, the compiler generates proof scripts for these optimizations with Parts 5. and 6. merged into one part providing the constant definition of a CFGB declaration  $B$  which is declared for both  $S$  and  $T$ . Further, it holds that Part 7. significantly differs from optimization to optimization and depends on the parameters of the optimization correctness criterion  $\text{TCC}_O$ . The part comprises the following for particular optimizations:

*For the CF optimization:* the constant definition of a CPA result  $\mathcal{A}_{CPA}$ ,

*For the DAE optimization:* the constant definition of a LA result  $\mathcal{A}_{LA}$ ,

*For the NI optimization:* the constant definition of a corresponding block edge relation  $CBEP$ ,

*For the RAI optimization:* the constant definition of a LA result  $\mathcal{A}_{LA}$ ,

*For the RAE optimization:* the constant definitions of a corresponding block edge relation  $CBEP$  and a AEA result  $\mathcal{A}_{AEA}$ ,

In the following, we estimate the order of the length of the constant definition part of a proof script as a function of the length of the instruction list of  $S$ ,  $n$ , and the number of variables of  $S$ ,  $m$ . The length of the instruction lists of  $S$  and  $T$  are equal for all optimizations but the NI optimization which inserts a limited number of nop instructions  $k$  which is negligible. Therefore, to simplify the estimation, we assume that the length of the instruction lists of  $S$  and  $T$  are equal for all optimizations and that the number of variables  $m$  is less than  $n$ .

The constant definition of an IL program  $S = ((vds, instrs), I)$  comprises four constant definitions:

1. The constant definition of the variable declaration list  $vds$  whose length is a function of the number of variables  $m$ . Each variable declaration tuple  $(v, \tau)$  is defined in one proof script line. Therefore, the size of  $(v, \tau)$  is equal 1 and the size of the constant definition of  $vds$  is equal  $1 * m = m$
2. The constant definition of the instruction list  $instrs$  whose length is a function of the length of  $instrs$ . Each instruction  $instr$  is defined in one proof script line. Therefore, the size of  $instrs$  is equal 1 and the size of the constant definition of  $instrs$  is equal  $1 * n = n$
3. The constant definition of the input list  $I$  whose length is a function of the number of variables  $m$ . Each pair  $(v, val)$  in the input list  $I$  is defined in one proof script line. Therefore, the size of  $(v, val)$  is equal 1 and the size of the input list  $I$  is equal  $1 * m = m$
4. The constant definition of the program  $S$  is given in one proof script line:

$$S = ((vds, instrs), I)$$

Therefore, the size of this constant definition is equal 1.

Altogether, the size of the constant definition of  $S$  has order of  $m + n + m + 1 = 2 * m + n + 1 < 3n + 1 \in O(n)$  and the size of  $S$  and  $T$  together also has order of  $(3n + 1) + (3n + 1) = 6n + 2 \in O(n)$ . As we assumed that the length of instruction lists in  $S$  and  $T$  are equal, the size of the constant definition of  $T$  also has order of  $O(n)$ .

The length of the constant definition of a program type  $\Phi$  is a function of the number of variables  $m$ . Each type identifier  $\tau$  is defined in one proof script line. Therefore, the size of  $\tau$  is equal 1 and the size of the constant definition of  $\Phi$  has order of  $1 * m = m < n \in O(n)$ .

The constant definition of a CFGB declaration  $B = (pid_0, BP, BB, succBP, predB)$  comprises five constant definitions.

1. The constant definition of the block position descriptor environment  $BP$  has the form

$$BP = \text{mapof}(l)$$

where  $l$  is a list of pairs  $(pid, bpos)$  consisting of a block position identifier  $pid$  and a block position descriptor  $bpos$ . Each pair  $(pid, bpos)$  is defined in one proof script line and the length of  $l$  is a function of the number of block positions. The number of block positions is equal  $n$  for all optimizations with exception of the NI optimization where this number is equal  $n + k$ . Therefore, the size of  $(pid, bpos)$  is equal 1 and the size of the constant definition of  $BP$  is equal  $1 * (n + k) = n + k$

2. The constant definitions of  $BB$ ,  $succBP$ , and  $predB$  has forms, which are analogous to the one of  $BP$ . Therefore, it can be roughly estimated that they have the same sizes as  $BP$ .

3. The constant definition of the CFGB declaration is given in one proof script line:

$$B = (pid_0, BP, BB, succBP, predB)$$

Therefore, the size of this constant definition is equal 1.

Altogether, the size of the constant definition of  $B$  has order of  $4 * (n + k) + 1 = 4n + 4k + 1 < 8n + 1 \in O(n)$  and the size of  $B_S$  and  $B_T$  together has order of  $2 * (8n + 1) = 16n + 2 \in O(n)$

The constant definition of a data flow analysis result  $\mathcal{A}_O$  has the form

$$\mathcal{A}_O = \text{mapof}(l)$$

where  $l$  is a list of data flow informations which were determined by the compiler for each program point of  $S$ . Each data flow information is defined in one proof script line and the length of  $l$  is equal to the length of the instruction list of  $S$ ,  $n$ . Therefore, the size of the constant definition of  $\mathcal{A}$  has order  $1 * n = n \in O(n)$

Altogether, the size of the constant definition part of a proof script has order  $O(n) + O(n) + O(n) + O(n) = 4 * O(n) \in O(4n) = O(n)$ . Thus, the size of this part of the proof script is a linear function of the length of the instruction lists of the source and the target programs which is comparable to the size of semantic abstractions of programs in the traditional approaches to certifying compilers like the PCC and the TV.

Now, we estimate the size order of the lemma section of a proof script as a function of the length of the instruction list of  $S$ ,  $n$ , and the number of variables of  $S$ ,  $m$ . To simplify the estimation, we assume that the size of one lemma has order of  $O(1)$ . This assumption follows from the following facts:

- It holds for each lemma that its statement is written in one proof script line.
- Each lemma is proved by a tactic call that is written in one proof script line.
- Each tactic call line is followed by two proof script lines that close the proof and call an ML function that updates an appropriate lookup table with lemma names.

The lemma section of the proof script comprises the following parts:

*Part 8.:* provides a sequence of lemmas (*lookup lemmas*) proving an equation of the form:  $instrs!pc = instr$ , where  $instrs$  and  $instr$  are a instruction list and and a instruction in the program  $S$  or the program  $T$ .

*Part 9.:* provides a sequence of lookup lemmas proving an equation of the form:  $BP(pid) = \text{Some}(bpos)$ , where  $BP$ ,  $pid$  and  $bpos$  are a block position descriptor environment defined in the constant definition part of the proof script, a block position identifier, and a block position descriptor, respectively.

*Parts 10. through 12.:* provide sequences of lookup lemmas proving equations for the remaining mappings of the block position environment  $B_S$  and  $B_T$ . The forms of those equations are analogous to those in the previous part.

*Part 13.:* provides a sequence of lookup lemmas proving an equation of the form:  $\mathcal{A}_O(pc) = \text{Some}(x)$ , where  $\mathcal{A}_O$ ,  $pc$  and  $x$  are a result of data flow analysis defined in the constant definition part of the proof script, a program point, and a data flow information which was determined by the compiler for the program point  $pc$ , respectively.

*Part 14.:* provides proof of the predicate  $\text{wtp}(S, \Phi_S)$ ,

*Part 15.:* provides proof of the predicate  $\text{wtp}(T, \Phi_t)$ ,

*Part 16.:* provides proof of the predicate  $\text{wfB}(S, B_S)$ ,

*Part 17.:* provides proof of the predicate  $\text{wfB}(T, B_T)$ ,

*Part 18.:* provides proof of the predicate  $\text{TCC}_O((S, B_S), (T, B_T), \mathcal{A}_O)$ ,

*Part 19.:* provides proof of the predicate  $\text{corrTrans}(S, T)$ .

As Part 8. comprises  $n$  lookup lemmas for  $S$  and  $n$  lookup lemmas for  $T$ , the size of Part 8. has order of  $O(n) + O(n) = O(n)$ .

Analogously, the sizes of Parts 9. through 13. have orders of  $O(n)$ , respectively.

The predicates in Parts 14. through 18. are proved in single lemmas. Thus, the size of each of them has order of  $O(1)$ ,

The size of Part 18. has order of  $O(n)$ . We exemplify the estimation by the optimization correctness criterion  $\text{TCC}_{\text{CF}}$ , cf. Definition 7.25. The definition of  $\text{TCC}_{\text{CF}}$  has the form of a conjunction  $P \wedge Q$ . The first conjunct,  $P$ , is

$$\text{TCC\_CF\_entry\_block}(B, \mathcal{A}_{\text{CPA}}, I)$$

which is proved in a single lemma. The second conjunct,  $Q$ , is

$$\forall s \text{ bid. } \text{TCC\_CF\_normal\_block}(S, T, B, \mathcal{A}_{\text{CPA}}, s, \text{bid})$$

where the  $\forall$ -quantified variable  $\text{bid}$  is a block identifier. This predicate is proved in  $n + 1$  lemmas.  $n$  lemmas prove  $n$  cases of  $\text{bid}$  and one lemma proves  $Q$  by cases over  $\text{bid}$ . Therefore, the size of Part 18. has order of  $(O(n) * O(1)) = O(n)$ .

Part 19. contains a single lemma whose statement is the predicate  $\text{corrTrans}(S, T)$ . Therefore, the size of Part 19. has order of  $O(1)$ .

Altogether, the size of the lemma section of a proof script has order of

$$O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(1) + O(1) + O(1) + O(1) + O(1) + O(n) + O(1) = O(n).$$

The size order of this section is the result of the trade-off decision between the small proof script size and facility of inspection. The former makes our approach interesting for real life applications. However, it turned out during the work on the SVF that debugging complex tactics proving complex predicates is extremely time consuming. Therefore, our tactics applied in proof scripts are the result of some experimentation and design decisions:

- Well-typedness of  $S$  and  $T$  is proved in single lemmas in Parts 14. and 15.. The lemmas are proved by single tactic calls which *do not* apply any auxiliary lookup lemmas proved in the script.
- Well-formedness of  $B_S$  and  $B_T$  is proved in single lemmas in Parts 16. and 17.. The lemmas are proved by single tactic calls which *apply* auxiliary lookup lemmas proved in the script.
- The lemma proving optimization correctness criterion is proved in  $O(n)$  lemmas in Part 18.. Each of these is proved by single tactic calls which *apply* auxiliary lookup lemmas proved in the script.

Further, it turned out that complex predicates proved in single lemma by a single tactic call tend to get a bottleneck during the proof checking. This results from the fact that the size of the proof state which has to be managed by the theorem prover grows quadratically with the size of the program. We anticipate that the successor FTV systems will split lemmas in the proof script in a large number of small auxiliary lemmas such that the theorem prover never has to manage large proof states during the proofs.

Concluding this section, we note that our implementation of tactics proving well-typedness in the proof script sections 14. and 15. demonstrates that within our approach there exists, at least potentially, a possibility to generate proof scripts with the lemma part of the size which has order of  $O(1)$ .

## 8.2 Performance

This section presents the evaluation of efficiency of checking of translation certificates.

In order to evaluate efficiency, we conducted two tests:

The aim of the first test was to investigate the overall time needed to check translation certificates for individual optimizations. The table in Figure 8.1 shows the runtimes<sup>1</sup> required to verify translation certificates generated by our compiler front-end for a program `example_in_thesis` that is a running example in this thesis and four other programs: `insert` takes an integer array  $a$ , an array index  $i$ , and integer  $x$ , and replaces the array element  $a[i]$  by  $x$ . `fibonacci_number` takes an integer  $n$  and computes the  $n$ -th fibonacci number. `straight_merge_sort` takes an integer array  $a$  and sorts  $a$  applying the straight merge sort algorithm. `example_in_thesis2` is a simple benchmark program. We created this program by copying-and-pasting the while-loop of the program `example_in_thesis` four times and giving it as input to our compiler. An IL program that is the result of translation of `example_in_thesis2` has four times more lines of code that is loop invariant (8 lines) then `example_in_thesis` (2 lines). The numbers in the `length` column of the table indicates how many lines of code do the results of translation of respective  $\mu$ C programs have. The times in the columns CF, DAE, NI, RAI, and RAE show how many seconds it takes to check a translation certificate that is generated for the optimization CF, DAE, NI, RAI, and RAE, respectively. The results of the tests are slightly better than the results achieved by Blech and Poetzsch-Heffter in [19]. This shows that

- the FTV approach is in general as feasible for realistic compilers as the TV approach applied to code generation presented in [19],
- the FTV approach has similar limitations as the TV approach when it comes to the efficiency of proof checking. Namely, the main limitation factor is the efficiency of the theorem prover. As aforementioned in Section 1.6, Blech and Gregoire in [18] used the theorem prover Coq and investigated the efficiency and the suitability of the TV approach for certification of the code generation phase. The results they achieved are very promising and they suggest that using the theorem prover Coq to implement our framework would yield a similar result.

program	length	CF	DAE	NI	RAI	RAE
<code>insert</code>	29	286 s	254 s	271 s	284 s	312 s
<code>example_in_thesis</code>	30	314 s	273 s	295 s	322 s	301 s
<code>fibonacci_number</code>	41	721 s	596 s	595 s	641 s	648 s
<code>example_in_thesis2</code>	98	6175 s	4290 s	5025 s	4313 s	4826 s
<code>straight_merge_sort</code>	101	4759 s	4487 s	4514 s	4629 s	5381 s

**Fig. 8.1.** Overall proof checking time

The aim of the second test was to investigate which parts of translation certificates take the most time to check the proofs of lemmas they comprise. We investigated this by choosing the CF optimization and measuring the time it took to check lemmas in all parts of proof scripts generated of optimizations CF. The table in Figure 8.2 shows the results of the experiment achieved with the programs from the first part of our evaluation. The numbers in the columns of the tables denote the following:

- the numbers in the column "`auxiliary lemmas`" indicate how many time in seconds does it take to prove auxiliary lemmas in a translation certificate generated for an optimization CF

<sup>1</sup> Experiments were conducted on Linux PC with 1.4 GHz.

and what part does it have in the overall time needed to check the certificate completely. Some of the auxiliary lemmas in the translation certificates are "glue code" lemmas explained in Section 3.7.

- The number in the column "lookup lemmas from Parts 8. through 13." indicate how many time in seconds does it take to prove lookup lemmas in the translation certificate generated for the optimization CF and what part does it have in the overall time needed to check the certificate completely.
- The number in the column "lemmas from Parts 14. through 15." indicate how many time in seconds does it take to prove well-typedness the source and the target programs in a translation certificate generated for the optimization CF and what part does it have in the overall time needed to check the certificate completely.
- The number in the column "lemmas from Parts 16. through 17." indicate how many time in seconds does it take to prove well-formedness of the CFGB declaration in a translation certificate generated for the optimization CF and what part does it have in the overall time needed to check the certificate completely.
- The number in the column "lemmas from Parts 18. through 19." indicate how many time in seconds does it take to prove that the source and the target programs fulfill the optimization correctness criterion  $TCC_{CF}$  and the main lemma in a translation certificate generated for the optimization CF and what part does it have in the overall time needed to check the certificate completely.

The results of the test show that, besides the optimization correctness criterion  $TCC_{CF}$ , the most complex proof tasks in translation certificates are those of proving lookup lemmas and well-formedness of CFGB declarations. There are two reasons for this behavior: The first reason is the aforementioned inefficiency of the theorem prover Isabelle/HOL in dealing with proof goals containing assumptions or conclusions of the list type. The second reason is that well-typedness and CFGB-well-formedness predicates are fairly complex and proof states that emerge during their proofs are very large. In order to avoid that the Isabelle/HOL has to manage large stacks of proof subgoals, one should split these proofs in a sequence of proofs of auxiliary lemmas that would then be applied in a main lemma proving the respective predicate. However, in our prototype framework, we implemented tactics that prove these predicate by applying as little auxiliary lemmas as possible (cf. the discussion on this topic in the previous section).

program	length	auxiliary lemmas	lookup lemmas from Parts 8. through 13.	lemmas from Parts 14. and 15.	lemmas from Parts 16. and 17.	lemmas from Parts 18. and 19.	overall time (100%)
insert	29	3 s 1.04 %	42 s 14.68 %	15 s 5.24 %	168 s 58.74 %	55 s 19.23 %	286 s
example_in_thesis	30	5 s 1.59 %	44 s 14.01 %	16 s 5.09 %	189 s 60.19 %	58 s 18.47 %	314 s
fibonacci_number	41	2 s 0.27 %	102 s 14.14 %	35 s 4.85 %	416 s 57.69 %	163 s 22.60 %	721 s
example_in_thesis2	98	38 s 0.61 %	556 s 9.0 %	288 s 4.66 %	3030 s 49.06 %	2260 s 36.59 %	6175 s
straight_merge_sort	101	38 s 0.79 %	572 s 12.01 %	321 s 6.74 %	2999 s 63.01 %	827 s 17.37 %	4759 s

Fig. 8.2. Relative proof checking times for the CF optimization

### 8.3 Framework evaluation

This section presents the evaluation of the formal framework described in this thesis. Our evaluation is focused on the following aspects of the SVF:

- *Is our formal framework suitable for certification of complete program transformation chains performed by the compiler?*

The layer architecture of the SVF we propose in this thesis provides a simple and elegant means for hierarchical structuring a set of translation correctness criteria which are associated with individual program transformations and phases of the compiler. As our SVF is structured in this way, it is easily extensible and we explained in Chapter 3 that integrating the extensions capturing further optimizations or compiler phases in our SVF can be done with the minimal effort.

- *Is it possible to augment the intermediate language IL supported by our SVF by further imperative programming language features?*

Augmenting the formalization of the language IL in the translation contract by the standard imperative language features like procedure calls, pointers, and recursion should not be problematic. The same holds for the effort needed for adapting the formalizations of optimization correctness criteria to the new formalization of the language IL. However, at the current stage of the development of our SVF, it is difficult to anticipate what formalizations have to be implemented to make translation certificates automatically checkable.

- *Is it possible to augment our formal framework such that it supports certifying further intermediate language optimizations?*

In general, our framework is suitable for certifying all structure preserving optimizations which can be justified using data flow analysis results. Further, we anticipate that it should be also possible to formalize optimization criteria for at least some of the structure modifying optimizations like the code motion optimizations: branch elimination and partial redundancy elimination. At the current stage of the development of our SVF, it is unknown how our framework has to be adapted such that it supports other structure modifying optimizations like loop optimizations.



## Chapter 9

---

### Conclusions and future work

This chapter summarizes the achievements of this thesis and considers future directions of research.

#### 9.1 Contributions

Most software systems are described in high-level model or programming languages. Their runtime behavior, however, is controlled by the compiled code. For uncritical software, it may be sufficient to test the runtime behavior of the code. For safety-critical software, there is an additional aggravating factor resulting from the fact that the code must satisfy the formal specification which reflects the *safety policy* of the *software consumer* and that the *software producer* is obliged to demonstrate that the code is correct with respect to the specification using formal verification techniques. In this scenario, it is of great importance that static analyses and formal methods can be applied on the source code level, because this level is more abstract and better suited for such techniques. However, the results of the analyses and the verification can only be carried over to the machine-code level, if we can establish the correctness of the translation. Thus, compilation is a crucial step in the development of software systems and formally verified *translation correctness* is essential to close the formalization chain from high-level formal methods to the machine-code level.

In this thesis, we propose an approach to certifying compilers which achieves the aim of closing the formalization chain from high-level formal methods to the machine-code level by applying techniques from mathematical logic and programming language semantics, a *foundational translation validation* (FTV).

The FTV approach has several novel features that, in combination, give it an advantage over traditional approach to certifying compilers, the TV approach:

*The presence of an explicit translation contract formalized in HOL:* The traditional TV approach does not formalizes a translation contract explicitly. Instead of this, the operational semantics and translation correctness criterion are implicit, i.e. they are incorporated in the translation validator program on the programming language level. Here, the explicit formalization of the translation contract entails the following advantages over the TV approach: Firstly, the formalizations of the source and the target languages in the translation contract makes the verification infrastructure of the SVF more flexible as novel programming language features, such as type systems or program semantics definitions, can be introduced to the software consumer without reimplementing of the type system or the operational semantics incorporated in the translator validator. Secondly, the formalization of the translation correctness predicate increases software consumer's confidence and makes the verification infrastructure of the SVF more flexible similarly to the formalizations of the source and the target languages as novel optimization procedures or further intermediate program transformations can be introduced

without reimplementing the VCG, which is a part of the translation validator program in the TV approach.

*Representation of programs in correctness proofs as logic constants:* The TV approach translates programs into their semantic abstractions on the programming language level. These abstractions serve as input for the translation validator. Translating the programs into their representations as HOL constants entails smaller size of the TCB that is associated with the FTV approach in comparison to the one that associated with the TV approach. This is due to the fact that, in the FTV approach, program representations as HOL constants are just other syntactic representations of original programs on the programming language level. Thus, in general, the software consumer is able to check if a logic constant and a program are corresponding by translating the constant back and check if the program and the result of translation are equal. In the TV approach, translating the semantic abstractions back into programs is, in general, not possible.

*Certification of program transformation chains:* Unlike the TV approach, which certifies single program transformations, the FTV approach achieves the aim of certifying whole chains of program transformations. This is possible due to the following three facts: Firstly, the translation contract provides, for all programming languages involved in the program transformation chain, definitions of program semantics functions which map programs to mathematical objects that are elements of a set with an (at least) partial order " $\leq$ ". Secondly, the definition of the translation correctness predicate is based on the definition of " $\leq$ ". Thirdly, for each translation run, which performs a chain of program transformations, the compiler generates a proof that the input and output of the transformation chain fulfill the translation correctness predicate. The proof makes use of the fact that the relation " $\leq$ " is transitive.

We see our work as a study of feasibility of the FTV approach. To investigate this approach, we implemented a small proof generating compiler and formalized a prototype SVF which accompanies the compiler. Our work resulted in a novel formal method for certifying compilers following the FTV approach which can serve as a prototype for further investigations.

The main contributions of this thesis are:

- To our knowledge, our implementation is the first certifying compiler which completely follows the FTV approach.
- Formalization of a translation contract for a small intermediate language.
- Specification and verification of the notion of declarations of control flow graphs with blocks.
- Formalization and verification of the notion of optimization patterns for selected structure preserving optimizations (i.e. optimizations which transforms programs locally). The optimization patterns allow for proving optimizations correct without reasoning about symbolic executions of programs.
- Techniques to combine and to reuse selected formalizations of the optimization patterns. Using our techniques, we have shown how to certify a nontrivial transformation such as loop invariant hoisting.
- Techniques for structuring the verification of optimizations into program dependent and independent parts.
- Implementation of techniques to automate verification of optimizations which applies dedicated proof tactics implemented within the theorem prover.
- First experimental results, experiences, and technical propositions on how to run proofs more efficiently.

## 9.2 Future work

This section details some possible future directions of research based on the work presented in this thesis. The possibilities are split into three categories:

1. improving the efficiency of proof checking,
2. augmenting the programming language features,
3. augmenting the set of optimizations supported by the SVF, and
4. researching into possible applications of the FTV approach.

*Efficiency:* In Chapter 8, we pointed out that our framework comprises two formalisms which proved to be bottlenecks during proof checking:

1. Our formalizations of the notions of well-typedness, well-formedness of block position environments, and optimization correctness criterions  $TCC_O$  formulate their requirements in terms of *equations with lookup expressions on their left-hand side*, e.g. the equations  $\mathcal{P}!(\text{idxof}(v, 0, vds)) = \tau$  and  $BP(pid) = \text{Some}(pid, bid, bsize, bidx, pc)$  in Definitions 5.10 and 6.13, respectively. As it turned out that each such equation has to be proved at least one time during checking of a translation certificate, these equations are proved in separate lemmas. As the number of these lemmas has order of  $O(n)$ , where  $n$  is a function of the length of the instruction list of a program, proving all lookup equations in a proof script requires time which has order of  $O(n^2)$ . One way to improve the efficiency here would be to implement appropriate Isabelle/HOL oracles, which are ordinary ML functions provided by the programming interface of Isabelle/HOL, and replace the tactic calls which prove the lookup equations by the respective oracle calls. This would increase the size of the trusted computing base of the FTV system but the number of ML code lines which would have to be trusted would be very small. In our work on the SVF, we already have made some experience in that matter as we wrote ML functions which convert constant definitions in the proof scripts generated by our compiler front-end into ML data structures which serve as lookup tables for functions which compute tactics solving, among other things, lemmas with equations involving lookup expressions. Therefore, we can report that in our SVF an ML function which converts a block position descriptor environment  $BP$  into a corresponding data structure needs 15 lines of ML code. Converting is done only one time at the beginning of the proof script. Given that a call to such a function would result in converting the definition of  $BP$  into a list  $l$ , then an oracle would have to convert a lemma's statement of the form

$$BP("pid") = \text{Some}("pid", "bid", bsize, "bidx", pc)$$

into an ML expression of the form

$$\text{lkp\_BP}("pid", l) = ("pid", "bid", bsize, "bidx", pc)$$

where  $\text{lkp\_BP}$  denotes an ML function in our SVF which performs lookup in  $l$  and needs 2 lines of ML code. We anticipate that the implementation of such a function would need no more than another 15 lines of ML code. Hence, the increase of the size of the TCB would be very small.

2. Another factor which extremely slows down checking proof scripts is *proofs of well-formedness of block position environments*. Proving that  $\text{wfB}(P, B)$  holds true for a program  $P$  and a block position environment  $B$  is time-consuming due to its definition which is given in a declarative style. We have roughly estimated that currently the time complexity of proving a statement of the form  $\text{wfB}(P, B)$  has order of  $O(n^2)$ , where  $n$  is a function of the length of the instruction list of a program. As for most optimizations the length of blocks in the CFGB declaration is

equal 1 and the allocation relation between program points and block positions is defined as a one-to-one correspondence, it is conceivable to formalize for each optimization  $O$  a predicate expressing well-formedness of  $B$ ,  $\text{wfB}'$ , which is specific to the optimization  $O$  and has the following properties:

- a) the SVF provides a proof of the lemma of the form

$$\text{wfB}'(P, B) \implies \text{wfB}(P, B)$$

- b) proving that  $\text{wfB}'(P, B)$  holds true for a program  $P$  and a block position environment  $B$  requires time which has order of  $O(n)$ , where  $n$  is a function of the length of the instruction list of  $P$ .

Then, proving the lemma

$$\text{wfB}(P, B)$$

would require  $O(n)$  time and would be done in two steps: The first step would apply the lemma in a). The second lemma would prove

$$\text{wfB}(P, B)$$

in  $O(n)$  time.

Another direction of research which should be considered is investigating how much efficiency would be gained and how the size of the TCB would be affected by applying the program generation mechanism in Isabelle/HOL which generates executable ML programs from Isabelle/HOL specifications.

*The augmentation of the supported source language:* In this thesis, we presented a prototype FTV system. The FTV system comprises two parts: a small certifying compiler front-end and an accompanying SVF. Our compiler front-end translates a  $\mu\text{C}$  language and performs three optimizations. As the language  $\mu\text{C}$  is a small subset of the language  $\text{C}$ , our FTV system is not applicable yet in developing real life applications. To improve this, the starting point for further work would be to make the implementation of our FTV system complete in the sense that it would be able to certify complete translation runs of the compiler. The work on this would include the following steps:

- implementing the back-end of our compiler,
- re-defining the translation correctness predicate `corrTrans` in the translation contract provided by Layer 3 in the SVF such that `corrTrans` is a predicate on  $\mu\text{C}$  and machine code programs,
- augmenting the implementation of Layer 4 in the SVF with phase independent formal frameworks which enable certifying complete translation runs of the compiler.
- augmenting the implementation of Layer 5 in the SVF with phase dependent formal frameworks which enable certifying the translation and the code generation phases.
- augmenting the implementation of Layer 6 in the SVF with proof environments providing tactics specific to the translation and the code generation phases.

Then, one direction of the research with such FTV system would be to extend the  $\mu\text{C}$  and IL languages by features which are standard in imperative languages, like procedures and pointers, and to re-implement the SVF in order to support this changes. Another interesting direction of the research would be to investigate the suitability of the FTV approach to certify translations of languages which support other paradigms like object-oriented languages.

*The augmentation of the SVF:* Currently, our SVF supports three optimizations: constant folding, dead assignment elimination, and loop invariant hoisting, which is certainly not enough to

make our FTV system suitable for real-life applications. Here, one direction of the research would be to augment the compiler by additional optimizations and to augment Layer 5 in the SVF by formalizations of new optimization correctness criteria which enable certifying these optimizations.

This thesis demonstrated that our SVF provides formal means which are suitable for certifying structure preserving optimizations such as constant folding and dead assignment elimination. Therefore, extending the SVF by the verification infrastructure for other optimizations of this kind should not pose any serious problems. This thesis also exemplified certification of one structure modifying transformation, loop invariant hoisting. However, it is unknown at the current stage of the development of our FTV system how much the FTV approach is suitable for certifying such optimizations. We anticipate that the formal method we developed within this thesis is directly applicable to at least some of them such as the code motion optimizations: branch elimination and partial redundancy elimination.

*Applications:* The FTV system described in this thesis provides an elegant and flexible SVF to specify and verify program semantics equivalence. However, the SVF can also serve as a good starting point for the research of other applications of the FTV approach. Here, possible directions of the research would be to investigate the suitability of the approach for the certification of the following:

- preservation of safety properties by program transformations which are performed during the development of embedded systems, e.g. preservation of real-time computing constraints,
- preservation of safety and liveness properties by program transformations which are performed during the development of embedded systems, e.g. preservation of deadlock-freeness property.

To conclude this thesis, we note that the results of our research described here demonstrates that ideas and techniques from mathematical logic and programming languages can and should be used for solving problems posed by software systems as complex as compilers. The design and implementation of our FTV system demonstrates the path one should follow when implementing a complex software system delivering solution to a sophisticated problem. Namely, the path goes from analysing the problem and identifying subproblems through a hierarchy of subproblems and their solutions differing from each other by the level of abstraction to a software system whose architecture is a hierarchy of subsystems which complies with the hierarchy of the solutions and builds solutions of the high abstraction-level problems from solutions of problems on the lower abstraction-level.



## Chapter A

---

Intermediate programs illustrating work-flow of our front-end

<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: _tI_1 = 0; [1]: i = _tI_1; [2]: _tI_2 = 0; [3]: res = _tI_2; [4]: _tI_3 = 2; [5]: _tB_1 = n &lt; _tI_3; [6]: BRANCH ~_tB_1 [11]; [7]: _tI_4 = 1; [8]: _tI_5 = n + _tI_4; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: _tI_6 = n * n; [13]: tmp = _tI_6; [14]: _tI_7 = res + tmp; [15]: _tI_8 = a[i]; [16]: _tI_9 = _tI_7 + _tI_8; [17]: res = _tI_9; [18]: _tI_10 = a[i]; [19]: _tI_11 = tmp + _tI_10; [20]: PRINTI _tI_11; [21]: _tI_12 = 1; [22]: _tI_13 = i + _tI_12; [23]: i = _tI_13; [24]: _tI_14 = 4; [25]: _tB_2 = i &lt; _tI_14; [26]: BRANCH _tB_2 [11]; [27]: PRINTI res [28]; [28]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>	<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: _tI_1 = 0; [1]: i = 0; [2]: _tI_2 = 0; [3]: res = 0; [4]: _tI_3 = 2; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: _tI_4 = 1; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: _tI_6 = n * n; [13]: tmp = _tI_6; [14]: _tI_7 = res + tmp; [15]: _tI_8 = a[i]; [16]: _tI_9 = _tI_7 + _tI_8; [17]: res = _tI_9; [18]: _tI_10 = a[i]; [19]: _tI_11 = tmp + _tI_10; [20]: PRINTI _tI_11; [21]: _tI_12 = 1; [22]: _tI_13 = i + 1; [23]: i = _tI_13; [24]: _tI_14 = 4; [25]: _tB_2 = i &lt; 4; [26]: BRANCH _tB_2 [11]; [27]: PRINTI res; [28]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>
---	--

**Fig. A.1.** The programs  $IL_0$ , left, and  $IL_1$ , right, which are the source and the target programs of the  $CF$  transformation.

<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: _tI_1 = 0; [1]: i = 0; [2]: _tI_2 = 0; [3]: res = 0; [4]: _tI_3 = 2; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: _tI_4 = 1; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: _tI_6 = n * n; [13]: tmp = _tI_6; [14]: _tI_7 = res + tmp; [15]: _tI_8 = a[i]; [16]: _tI_9 = _tI_7 + _tI_8; [17]: res = _tI_9; [18]: _tI_10 = a[i]; [19]: _tI_11 = tmp + _tI_10; [20]: PRINTI _tI_11; [21]: _tI_12 = 1; [22]: _tI_13 = i + 1; [23]: i = _tI_13; [24]: _tI_14 = 4; [25]: _tB_2 = i &lt; 4; [26]: BRANCH _tB_2 [11]; [27]: PRINTI res; [28]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>	<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: _tI_6 = n * n; [13]: tmp = _tI_6; [14]: _tI_7 = res + tmp; [15]: _tI_8 = a[i]; [16]: _tI_9 = _tI_7 + _tI_8; [17]: res = _tI_9; [18]: _tI_10 = a[i]; [19]: _tI_11 = tmp + _tI_10; [20]: PRINTI _tI_11; [21]: GOTO [22]; [22]: _tI_13 = i + 1; [23]: i = _tI_13; [24]: GOTO [25]; [25]: _tB_2 = i &lt; 4; [26]: BRANCH _tB_2 [11]; [27]: PRINTI res; [28]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>
--	--

**Fig. A.2.** The programs  $IL_1$ , left, and  $IL_2$ , right, which are the source and the target programs of the DAE transformation.

<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: _tI_6 = n * n; [13]: tmp = _tI_6; [14]: _tI_7 = res + tmp; [15]: _tI_8 = a[i]; [16]: _tI_9 = _tI_7 + _tI_8; [17]: res = _tI_9; [18]: _tI_10 = a[i]; [19]: _tI_11 = tmp + _tI_10; [20]: PRINTI _tI_11; [21]: GOTO [22]; [22]: _tI_13 = i + 1; [23]: i = _tI_13; [24]: GOTO [25]; [25]: _tB_2 = i &lt; 4; [26]: BRANCH _tB_2 [11]; [27]: PRINTI res; [28]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>	<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: GOTO [13]; [13]: _tI_6 = n * n [14]: tmp = _tI_6; [15]: _tI_7 = res + tmp; [16]: _tI_8 = a[i]; [17]: _tI_9 = _tI_7 + _tI_8; [18]: res = _tI_9; [19]: _tI_10 = a[i]; [20]: _tI_11 = tmp + _tI_10; [21]: PRINTI _tI_11; [22]: GOTO [23]; [23]: _tI_13 = i + 1; [24]: i = _tI_13 [25]: GOTO [26]; [26]: _tB_2 = i &lt; 4; [27]: BRANCH _tB_2 [12]; [28]: PRINTI res; [29]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>
--	---

**Fig. A.3.** The programs  $IL_2$ , left, and  $IL_3$ , right, which are the source and the target programs of the  $NI$  transformation.

<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: GOTO [12]; [12]: GOTO [13]; [13]: _tI_6 = n * n [14]: tmp = _tI_6; [15]: _tI_7 = res + tmp; [16]: _tI_8 = a[i]; [17]: _tI_9 = _tI_7 + _tI_8; [18]: res = _tI_9; [19]: _tI_10 = a[i]; [20]: _tI_11 = tmp + _tI_10; [21]: PRINTI _tI_11; [22]: GOTO [23]; [23]: _tI_13 = i + 1; [24]: i = _tI_13 [25]: GOTO [26]; [26]: _tB_2 = i &lt; 4; [27]: BRANCH _tB_2 [12]; [28]: PRINTI res; [29]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>	<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0 [2]; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11] [7]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: _tI_6 = n * n; [12]: GOTO [13]; [13]: _tI_6 = n * n; [14]: tmp = _tI_6; [15]: _tI_7 = res + tmp; [16]: _tI_8 = a[i]; [17]: _tI_9 = _tI_7 + _tI_8; [18]: res = _tI_9; [19]: _tI_10 = a[i]; [20]: _tI_11 = tmp + _tI_10; [21]: PRINTI _tI_11; [22]: GOTO [23]; [23]: _tI_13 = i + 1; [24]: i = _tI_13; [25]: GOTO [26]; [26]: _tB_2 = i &lt; 4; [27]: BRANCH _tB_2 [12] [28]; [28]: PRINTI res; [29]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>
---	--

**Fig. A.4.** The programs  $IL_3$ , left, and  $IL_4$ , right, which are the source and the target programs of the *RAI* transformation.

<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0 [2]; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11] [7]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: _tI_6 = n * n; [12]: GOTO [13]; [13]: _tI_6 = n * n; [14]: tmp = _tI_6; [15]: _tI_7 = res + tmp; [16]: _tI_8 = a[i]; [17]: _tI_9 = _tI_7 + _tI_8; [18]: res = _tI_9; [19]: _tI_10 = a[i]; [20]: _tI_11 = tmp + _tI_10; [21]: PRINTI _tI_11; [22]: GOTO [23]; [23]: _tI_13 = i + 1; [24]: i = _tI_13; [25]: GOTO [26]; [26]: _tB_2 = i &lt; 4; [27]: BRANCH _tB_2 [12] [28]; [28]: PRINTI res; [29]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>	<pre> vardecls n int; a int [4]; i int; tmp int; res int; _tI_1 int; _tI_2 int; _tI_3 int; _tB_1 bool; _tI_4 int; _tI_5 int; _tI_6 int; _tI_7 int; _tI_8 int; _tI_9 int; _tI_10 int; _tI_11 int; _tI_12 int; _tI_13 int; _tI_14 int; _tB_2 bool; begin [0]: GOTO [1]; [1]: i = 0; [2]: GOTO [3]; [3]: res = 0; [4]: GOTO [5]; [5]: _tB_1 = n &lt; 2; [6]: BRANCH ~_tB_1 [11]; [7]: GOTO [8]; [8]: _tI_5 = n + 1; [9]: n = _tI_5; [10]: GOTO [4]; [11]: _tI_6 = n * n; [12]: GOTO [13]; [13]: GOTO [14]; [14]: tmp = _tI_6; [15]: _tI_7 = res + tmp; [16]: _tI_8 = a[i]; [17]: _tI_9 = _tI_7 + _tI_8; [18]: res = _tI_9; [19]: _tI_10 = a[i]; [20]: _tI_11 = tmp + _tI_10; [21]: PRINTI _tI_11; [22]: GOTO [23]; [23]: _tI_13 = i + 1; [24]: i = _tI_13; [25]: GOTO [26]; [26]: _tB_2 = i &lt; 4; [27]: BRANCH _tB_2 [12]; [28]: PRINTI res; [29]: EXIT; end  INPUT begin: (n, 1), (a, [9, 7, 5, 0]), (i, 0), (tmp, 0), (res, 0), (_tI_1, 0), (_tI_2, 0), (_tI_3, 0), (_tB_1, false), (_tI_4, 0), (_tI_5, 0), (_tI_6, 0), (_tI_7, 0), (_tI_8, 0), (_tI_9, 0), (_tI_10, 0), (_tI_11, 0), (_tI_12, 0), (_tI_13, 0), (_tI_14, 0), (_tB_2, false) INPUT end </pre>
--	---

**Fig. A.5.** The programs  $IL_4$ , left, and  $IL_5$ , right, which are the source and the target programs of the *RAE* transformation.

## Chapter B

### Specification of optimization relation for constant folding

#### B.1 Optimization patterns for expressions

**Definition B.1.** (*Translation relation over operand expression pairs*)

$$\begin{aligned} \text{cf\_transrel\_expr\_operand} &: \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF} \\ \text{cf\_transrel\_expr\_operand}(inv) &= \\ &\{ (e, e') \mid \exists i. (e, e') = (i, i) \vee \\ &\quad \exists b. (e, e') = (b, b) \vee \\ &\quad \exists v. (e, e') = (v, v) \wedge inv(v) = \text{None} \vee \\ &\quad \exists v i. (e, e') = (v, i) \wedge inv(v) = \text{Some}(i) \vee \\ &\quad \exists v b. (e, e') = (v, b) \wedge inv(v) = \text{Some}(b) \vee \\ &\quad \exists a i. (e, e') = (a[i], a[i]) \vee \\ &\quad \exists a v. (e, e') = (a[v], a[v]) \wedge inv(v) = \text{None} \vee \\ &\quad \exists a v i. (e, e') = (a[v], a[i]) \wedge inv(v) = \text{Some}(i) \} \end{aligned}$$

**Definition B.2.** (*Translation relation over pairs of expressions of the form  $\neg o$* )

$$\begin{aligned} \text{cf\_transrel\_expr\_not} &: \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF} \\ \text{cf\_transrel\_expr\_not}(inv) &= \{ (e, e') \mid \\ &\quad \exists b_1 b_2. (e, e') = (\neg b_1, b_2) \wedge \neg b_1 = b_2 \vee \\ &\quad \exists v. (e, e') = (\neg v, \neg v) \wedge inv(v) = \text{None} \vee \\ &\quad \exists v_1 b_1 b_2 b_3. (e, e') = (\neg v, b_2) \wedge inv(v) = \text{Some}(b_1) \wedge \neg b_1 = b_2 \vee \\ &\quad \exists a i. (e, e') = (\neg a[i], \neg a[i]) \vee \\ &\quad \exists a v. (e, e') = (\neg a[v], \neg a[v]) \wedge inv(v) = \text{None} \vee \\ &\quad \exists a v i. (e, e') = (\neg a[v], \neg a[i]) \wedge inv(v) = \text{Some}(i) \} \end{aligned}$$

**Definition B.3.** (*Translation relation over pairs of expressions of the syntactic form  $\neg o$* )

$$\begin{aligned} \text{cf\_transrel\_expr\_unmin} &: \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF} \\ \text{cf\_transrel\_expr\_unmin}(inv) &= \{ (e, e') \mid \\ &\quad \exists i_1 i_2 i_3. (e, e') = (\neg i_1, i_2) \wedge \neg i_1 = i_2 \vee \\ &\quad \exists v_1 i_2. (e, e') = (\neg v, \neg v) \wedge inv(v) = \text{None} \vee \\ &\quad \exists v i_1 i_2. (e, e') = (\neg v, i_2) \wedge inv(v) = \text{Some}(i_1) \wedge \neg i_1 = i_2 \vee \\ &\quad \exists a i. (e, e') = (\neg a[i], \neg a[i]) \vee \\ &\quad \exists a v. (e, e') = (\neg a[v], \neg a[v]) \wedge inv(v) = \text{None} \vee \\ &\quad \exists a v i. (e, e') = (\neg a[v], \neg a[i]) \wedge inv(v) = \text{Some}(i) \vee \end{aligned}$$

**Definition B.4.** (Translation relation over pairs of expressions of the syntactic form  $o_1 + o_2$ )

$\text{cf\_transrel\_expr\_plus} : \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF}$   
 $\text{cf\_transrel\_expr\_plus}(inv) = \{ (e, e') \mid$   
 $\exists i_1 i_2 i_3. (e, e') = (i_1 + i_2, i_3) \wedge i_1 + i_2 = i_3 \vee$   
 $\exists v_1 i_2. (e, e') = (v_1 + i_2, v_1 + i_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists v_1 i_1 i_2 i_3. (e, e') = (v_1 + i_2, i_3) \wedge inv(v_1) = \text{Some}(i_1) \wedge i_1 + i_2 = i_3 \vee$   
 $\exists a i_1 i_2. (e, e') = (a[i_1] + i_2, a[i_1] + i_2) \vee$   
 $\exists a v_1 i_2. (e, e') = (a[v_1] + i_2, a[v_1] + i_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (a[v_1] + i_2, a[i_1] + i_2) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists v_1 i_2. (e, e') = (i_1 + v_2, i_1 + v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 i_2 i_3. (e, e') = (i_1 + v_2, i_3) \wedge inv(v_2) = \text{Some}(i_2) \wedge i_1 + i_2 = i_3 \vee$   
 $\exists v_1 v_2. (e, e') = (v_1 + v_2, v_1 + v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 i_2. (e, e') = (v_1 + v_2, v_1 + i_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists v_1 v_2 i_1. (e, e') = (v_1 + v_2, i_1 + v_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 i_1 i_2 i_3. (e, e') = (v_1 + v_2, i_3) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \wedge i_1 + i_2 = i_3 \vee$   
 $\exists a i_1 v_2. (e, e') = (a[i_1] + v_2, a[i_1] + v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (a[i_1] + v_2, a[i_1] + i_2) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (a[v_1] + v_2, a[v_1] + v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (a[v_1] + v_2, a[v_1] + i_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (a[v_1] + v_2, a[i_1] + v_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (a[v_1] + v_2, a[i_1] + i_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a i_1 i_2. (e, e') = (i_1 + a[i_2], i_1 + a[i_2]) \vee$   
 $\exists a v_1 i_2. (e, e') = (v_1 + a[i_2], v_1 + a[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (v_1 + a[i_2], i_1 + a[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists a_1 i_1 a_2 i_2. (e, e') = (a_1[i_1] + a_2[i_2], a_1[i_1] + a_2[i_2]) \vee$   
 $\exists a_1 v_1 a_2 i_2. (e, e') = (a_1[v_1] + a_2[i_2], a_1[v_1] + a_2[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 i_2. (e, e') = (a_1[v_1] + a_2[i_2], a_1[i_1] + a_2[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists a i_1 v_2. (e, e') = (i_1 + a[v_2], i_1 + a[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (i_1 + a[v_2], i_1 + a[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (v_1 + a[v_2], v_1 + a[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (v_1 + a[v_2], i_1 + a[v_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (v_1 + a[v_2], v_1 + a[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (v_1 + a[v_2], i_1 + a[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 i_1 a_2 v_2. (e, e') = (a_1[i_1] + a_2[v_2], a_1[i_1] + a_2[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[i_1] + a_2[v_2], a_1[i_1] + a_2[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 a_2 v_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[v_1] + a_2[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[i_1] + a_2[v_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[v_1] + a_2[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] + a_2[v_2], a_1[i_1] + a_2[i_2]) \wedge$   
 $inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \}$

**Definition B.5.** (Translation relation over pairs of expressions of the syntactic form  $o_1 - o_2$ )

$\text{cf\_transrel\_expr\_binmin} : \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF}$   
 $\text{cf\_transrel\_expr\_binmin}(inv) = \{(e, e') \mid$   
 $\exists i_1 i_2 i_3. (e, e') = (i_1 - i_2, i_3) \wedge i_1 - i_2 = i_3 \vee$   
 $\exists v_1 i_2. (e, e') = (v_1 - i_2, v_1 - i_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists v_1 i_1 i_2 i_3. (e, e') = (v_1 - i_2, i_3) \wedge inv(v_1) = \text{Some}(i_1) \wedge i_1 - i_2 = i_3 \vee$   
 $\exists a i_1 i_2. (e, e') = (a[i_1] - i_2, a[i_1] - i_2) \vee$   
 $\exists a v_1 i_2. (e, e') = (a[v_1] - i_2, a[v_1] - i_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (a[v_1] - i_2, a[i_1] - i_2) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists v_1 i_2. (e, e') = (i_1 - v_2, i_1 - v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 i_2 i_3. (e, e') = (i_1 - v_2, i_3) \wedge inv(v_2) = \text{Some}(i_2) \wedge i_1 - i_2 = i_3 \vee$   
 $\exists v_1 v_2. (e, e') = (v_1 - v_2, v_1 - v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 i_2. (e, e') = (v_1 - v_2, v_1 - i_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists v_1 v_2 i_1. (e, e') = (v_1 - v_2, i_1 - v_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 i_1 i_2 i_3. (e, e') = (v_1 - v_2, i_3) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \wedge i_1 - i_2 = i_3 \vee$   
 $\exists a i_1 v_2. (e, e') = (a[i_1] - v_2, a[i_1] - v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (a[i_1] - v_2, a[i_1] - i_2) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (a[v_1] - v_2, a[v_1] - v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (a[v_1] - v_2, a[v_1] - i_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (a[v_1] - v_2, a[i_1] - v_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (a[v_1] - v_2, a[i_1] - i_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a i_1 i_2. (e, e') = (i_1 - a[i_2], i_1 - a[i_2]) \vee$   
 $\exists a v_1 i_2. (e, e') = (v_1 - a[i_2], v_1 - a[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (v_1 - a[i_2], i_1 - a[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists a_1 i_1 a_2 i_2. (e, e') = (a_1[i_1] - a_2[i_2], a_1[i_1] - a_2[i_2]) \vee$   
 $\exists a_1 v_1 a_2 i_2. (e, e') = (a_1[v_1] - a_2[i_2], a_1[v_1] - a_2[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 i_2. (e, e') = (a_1[v_1] - a_2[i_2], a_1[i_1] - a_2[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists a i_1 v_2. (e, e') = (i_1 - a[v_2], i_1 - a[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (i_1 - a[v_2], i_1 - a[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (v_1 - a[v_2], v_1 - a[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (v_1 - a[v_2], i_1 - a[v_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (v_1 - a[v_2], v_1 - a[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (v_1 - a[v_2], i_1 - a[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 i_1 a_2 v_2. (e, e') = (a_1[i_1] - a_2[v_2], a_1[i_1] - a_2[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[i_1] - a_2[v_2], a_1[i_1] - a_2[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 a_2 v_2. (e, e') = (a_1[v_1] - a_2[v_2], a_1[v_1] - a_2[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2. (e, e') = (a_1[v_1] - a_2[v_2], a_1[i_1] - a_2[v_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] - a_2[v_2], a_1[v_1] - a_2[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] - a_2[v_2], a_1[i_1] - a_2[i_2]) \wedge$   
 $inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \}$



**Definition B.7.** (Translation relation over pairs of expressions of the syntactic form  $o_1 \wedge o_2$ )

$\text{cf\_transrel\_expr\_and} : \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF}$   
 $\text{cf\_transrel\_expr\_and}(inv) = \{(e, e') \mid$   
 $\exists b_1 b_2 b_3. (e, e') = (b_1 \wedge b_2, b_3) \wedge (b_1 \wedge b_2 = b_3) \vee$   
 $\exists v_1 b_2. (e, e') = (v_1 \wedge b_2, v_1 \wedge b_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists v_1 b_1 b_2 b_3. (e, e') = (v_1 \wedge b_2, b_3) \wedge inv(v_1) = \text{Some}(b_1) \wedge (b_1 \wedge b_2 = b_3) \vee$   
 $\exists a i_1 b_2. (e, e') = (a[i_1] \wedge b_2, a[i_1] \wedge b_2) \vee$   
 $\exists a v_1 b_2. (e, e') = (a[v_1] \wedge b_2, a[v_1] \wedge b_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 i_1 b_2. (e, e') = (a[v_1] \wedge b_2, a[i_1] \wedge b_2) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists v_1 b_2. (e, e') = (b_1 \wedge v_2, b_1 \wedge v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 b_2 b_3. (e, e') = (b_1 \wedge v_2, b_3) \wedge inv(v_2) = \text{Some}(b_2) \wedge (b_1 \wedge b_2 = b_3) \vee$   
 $\exists v_1 v_2. (e, e') = (v_1 \wedge v_2, v_1 \wedge v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 b_2. (e, e') = (v_1 \wedge v_2, v_1 \wedge b_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(b_2) \vee$   
 $\exists v_1 v_2 b_1. (e, e') = (v_1 \wedge v_2, b_1 \wedge v_2) \wedge inv(v_1) = \text{Some}(b_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 b_1 b_2 b_3. (e, e') = (v_1 \wedge v_2, b_3) \wedge inv(v_1) = \text{Some}(b_1) \wedge inv(v_2) = \text{Some}(b_2) \wedge (b_1 \wedge b_2 = b_3) \vee$   
 $\exists a i_1 v_2. (e, e') = (a[i_1] \wedge v_2, a[i_1] \wedge v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a i_1 v_2 b_2. (e, e') = (a[i_1] \wedge v_2, a[i_1] \wedge b_2) \wedge inv(v_2) = \text{Some}(b_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (a[v_1] \wedge v_2, a[v_1] \wedge v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 b_2. (e, e') = (a[v_1] \wedge v_2, a[v_1] \wedge b_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(b_2) \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (a[v_1] \wedge v_2, a[i_1] \wedge v_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 i_1 v_2 b_2. (e, e') = (a[v_1] \wedge v_2, a[i_1] \wedge b_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(b_2) \vee$   
 $\exists a b_1 i_2. (e, e') = (b_1 \wedge a[i_2], b_1 \wedge a[i_2]) \vee$   
 $\exists a v_1 i_2. (e, e') = (v_1 \wedge a[i_2], v_1 \wedge a[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 b_1 i_2. (e, e') = (v_1 \wedge a[i_2], b_1 \wedge a[i_2]) \wedge inv(v_1) = \text{Some}(b_1) \vee$   
 $\exists a_1 i_1 a_2 i_2. (e, e') = (a_1[i_1] \wedge a_2[i_2], a_1[i_1] \wedge a_2[i_2]) \vee$   
 $\exists a_1 v_1 a_2 i_2. (e, e') = (a_1[v_1] \wedge a_2[i_2], a_1[v_1] \wedge a_2[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 i_2. (e, e') = (a_1[v_1] \wedge a_2[i_2], a_1[i_1] \wedge a_2[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists a i_1 v_2. (e, e') = (b_1 \wedge a[v_2], b_1 \wedge a[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a b_1 v_2 i_2. (e, e') = (b_1 \wedge a[v_2], b_1 \wedge a[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (v_1 \wedge a[v_2], v_1 \wedge a[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 b_1 v_2. (e, e') = (v_1 \wedge a[v_2], b_1 \wedge a[v_2]) \wedge inv(v_1) = \text{Some}(b_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (v_1 \wedge a[v_2], v_1 \wedge a[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 b_1 v_2 i_2. (e, e') = (v_1 \wedge a[v_2], b_1 \wedge a[i_2]) \wedge inv(v_1) = \text{Some}(b_1) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 i_1 a_2 v_2. (e, e') = (a_1[i_1] \wedge a_2[v_2], a_1[i_1] \wedge a_2[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[i_1] \wedge a_2[v_2], a_1[i_1] \wedge a_2[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 a_2 v_2. (e, e') = (a_1[v_1] \wedge a_2[v_2], a_1[v_1] \wedge a_2[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2. (e, e') = (a_1[v_1] \wedge a_2[v_2], a_1[i_1] \wedge a_2[v_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] \wedge a_2[v_2], a_1[v_1] \wedge a_2[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] \wedge a_2[v_2], a_1[i_1] \wedge a_2[i_2]) \wedge$   
 $\left. \begin{array}{l} inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \end{array} \right\}$



**Definition B.9.** (Translation relation over pairs of expressions of the syntactic form  $o_1 = o_2$ )

$\text{cf\_transrel\_expr\_eq} : \text{ConstantValueEnv} \rightarrow \text{ExpressionTransRel\_CF}$   
 $\text{cf\_transrel\_expr\_eq}(inv) = \{(e, e') \mid$   
 $\exists i_1 i_2 b. (e, e') = (i_1 = i_2, b) \wedge (i_1 = i_2) = b \vee$   
 $\exists v_1 i_2. (e, e') = (v_1 = i_2, v_1 = i_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists v_1 i_1 i_2 b. (e, e') = (v_1 = i_2, b) \wedge inv(v_1) = \text{Some}(i_1) \wedge (i_1 = i_2) = b \vee$   
 $\exists a i_1 i_2. (e, e') = (a[i_1] = i_2, a[i_1] = i_2) \vee$   
 $\exists a v_1 i_2. (e, e') = (a[v_1] = i_2, a[v_1] = i_2) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (a[v_1] = i_2, a[i_1] = i_2) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists v_1 i_2. (e, e') = (i_1 = v_2, i_1 = v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 i_2 b. (e, e') = (i_1 = v_2, b) \wedge inv(v_2) = \text{Some}(i_2) \wedge (i_1 = i_2) = b \vee$   
 $\exists v_1 v_2. (e, e') = (v_1 = v_2, v_1 = v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 i_2. (e, e') = (v_1 = v_2, v_1 = i_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists v_1 v_2 i_1. (e, e') = (v_1 = v_2, i_1 = v_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists v_1 v_2 i_1 i_2 i_3. (e, e') = (v_1 = v_2, b) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \wedge (i_1 = i_2) = b \vee$   
 $\exists a i_1 v_2. (e, e') = (a[i_1] = v_2, a[i_1] = v_2) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (a[i_1] = v_2, a[i_1] = i_2) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (a[v_1] = v_2, a[v_1] = v_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (a[v_1] = v_2, a[v_1] = i_2) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (a[v_1] = v_2, a[i_1] = v_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (a[v_1] = v_2, a[i_1] = i_2) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a i_1 i_2. (e, e') = (i_1 = a[i_2], i_1 = a[i_2]) \vee$   
 $\exists a v_1 i_2. (e, e') = (v_1 = a[i_2], v_1 = a[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (v_1 = a[i_2], i_1 = a[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists a_1 i_1 a_2 i_2. (e, e') = (a_1[i_1] = a_2[i_2], a_1[i_1] = a_2[i_2]) \vee$   
 $\exists a_1 v_1 a_2 i_2. (e, e') = (a_1[v_1] = a_2[i_2], a_1[v_1] = a_2[i_2]) \wedge inv(v_1) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 i_2. (e, e') = (a_1[v_1] = a_2[i_2], a_1[i_1] = a_2[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \vee$   
 $\exists a i_1 v_2. (e, e') = (i_1 = a[v_2], i_1 = a[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (i_1 = a[v_2], i_1 = a[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (v_1 = a[v_2], v_1 = a[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (v_1 = a[v_2], i_1 = a[v_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (v_1 = a[v_2], v_1 = a[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (v_1 = a[v_2], i_1 = a[i_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 i_1 a_2 v_2. (e, e') = (a_1[i_1] = a_2[v_2], a_1[i_1] = a_2[v_2]) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[i_1] = a_2[v_2], a_1[i_1] = a_2[i_2]) \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 a_2 v_2. (e, e') = (a_1[v_1] = a_2[v_2], a_1[v_1] = a_2[v_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2. (e, e') = (a_1[v_1] = a_2[v_2], a_1[i_1] = a_2[v_2]) \wedge inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{None} \vee$   
 $\exists a_1 v_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] = a_2[v_2], a_1[v_1] = a_2[i_2]) \wedge inv(v_1) = \text{None} \wedge inv(v_2) = \text{Some}(i_2) \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] = a_2[v_2], a_1[i_1] = a_2[i_2]) \wedge$   
 $inv(v_1) = \text{Some}(i_1) \wedge inv(v_2) = \text{Some}(i_2) \}$



**Definition B.11.** (Translation relation over pairs of expressions of the syntactic form  $o_1 < o_2$ )

$cf\_transrel\_expr\_lt : \mathbf{ConstantValueEnv} \rightarrow \mathbf{ExpressionTransRel\_CF}$   
 $cf\_transrel\_expr\_lt(inv) = \{(e, e') \mid$   
 $\exists i_1 i_2 b. (e, e') = (i_1 < i_2, b) \wedge (i_1 < i_2) = b \vee$   
 $\exists v_1 i_2. (e, e') = (v_1 < i_2, v_1 < i_2) \wedge inv(v_1) = \mathbf{None} \vee$   
 $\exists v_1 i_1 i_2 b. (e, e') = (v_1 < i_2, b) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge (i_1 < i_2) = b \vee$   
 $\exists a i_1 i_2. (e, e') = (a[i_1] < i_2, a[i_1] < i_2) \vee$   
 $\exists a v_1 i_2. (e, e') = (a[v_1] < i_2, a[v_1] < i_2) \wedge inv(v_1) = \mathbf{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (a[v_1] < i_2, a[i_1] < i_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \vee$   
 $\exists v_1 i_2. (e, e') = (i_1 < v_2, i_1 < v_2) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists v_1 i_2 b. (e, e') = (i_1 < v_2, b) \wedge inv(v_2) = \mathbf{Some}(i_2) \wedge (i_1 < i_2) = b \vee$   
 $\exists v_1 v_2. (e, e') = (v_1 < v_2, v_1 < v_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists v_1 v_2 i_2. (e, e') = (v_1 < v_2, v_1 < i_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists v_1 v_2 i_1. (e, e') = (v_1 < v_2, i_1 < v_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists v_1 v_2 i_1 i_2 i_3. (e, e') = (v_1 < v_2, b) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \wedge (i_1 < i_2) = b \vee$   
 $\exists a i_1 v_2. (e, e') = (a[i_1] < v_2, a[i_1] < v_2) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (a[i_1] < v_2, a[i_1] < i_2) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (a[v_1] < v_2, a[v_1] < v_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (a[v_1] < v_2, a[v_1] < i_2) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (a[v_1] < v_2, a[i_1] < v_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (a[v_1] < v_2, a[i_1] < i_2) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a i_1 i_2. (e, e') = (i_1 < a[i_2], i_1 < a[i_2]) \vee$   
 $\exists a v_1 i_2. (e, e') = (v_1 < a[i_2], v_1 < a[i_2]) \wedge inv(v_1) = \mathbf{None} \vee$   
 $\exists a v_1 i_1 i_2. (e, e') = (v_1 < a[i_2], i_1 < a[i_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \vee$   
 $\exists a_1 i_1 a_2 i_2. (e, e') = (a_1[i_1] < a_2[i_2], a_1[i_1] < a_2[i_2]) \vee$   
 $\exists a_1 v_1 a_2 i_2. (e, e') = (a_1[v_1] < a_2[i_2], a_1[v_1] < a_2[i_2]) \wedge inv(v_1) = \mathbf{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 i_2. (e, e') = (a_1[v_1] < a_2[i_2], a_1[i_1] < a_2[i_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \vee$   
 $\exists a i_1 v_2. (e, e') = (i_1 < a[v_2], i_1 < a[v_2]) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a i_1 v_2 i_2. (e, e') = (i_1 < a[v_2], i_1 < a[i_2]) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a v_1 v_2. (e, e') = (v_1 < a[v_2], v_1 < a[v_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a v_1 i_1 v_2. (e, e') = (v_1 < a[v_2], i_1 < a[v_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a v_1 v_2 i_2. (e, e') = (v_1 < a[v_2], v_1 < a[i_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a v_1 i_1 v_2 i_2. (e, e') = (v_1 < a[v_2], i_1 < a[i_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a_1 i_1 a_2 v_2. (e, e') = (a_1[i_1] < a_2[v_2], a_1[i_1] < a_2[v_2]) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[i_1] < a_2[v_2], a_1[i_1] < a_2[i_2]) \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a_1 v_1 a_2 v_2. (e, e') = (a_1[v_1] < a_2[v_2], a_1[v_1] < a_2[v_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2. (e, e') = (a_1[v_1] < a_2[v_2], a_1[i_1] < a_2[v_2]) \wedge inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{None} \vee$   
 $\exists a_1 v_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] < a_2[v_2], a_1[v_1] < a_2[i_2]) \wedge inv(v_1) = \mathbf{None} \wedge inv(v_2) = \mathbf{Some}(i_2) \vee$   
 $\exists a_1 v_1 i_1 a_2 v_2 i_2. (e, e') = (a_1[v_1] < a_2[v_2], a_1[i_1] < a_2[i_2]) \wedge$   
 $inv(v_1) = \mathbf{Some}(i_1) \wedge inv(v_2) = \mathbf{Some}(i_2) \}$



**Definition B.13.** (*Translation relation over expression pairs* `cf_transrel_expr`)

$$\begin{aligned} & \text{cf\_transrel\_expr} : \mathbf{ConstantValueEnv} \rightarrow \mathbf{ExpressionTransRel\_CF} \\ & \text{cf\_transrel\_expr}(inv) = \{(e, e') \mid (e, e') \in \text{cf\_transrel\_expr\_operand}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_plus}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_binmin}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_mult}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_unmin}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_and}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_not}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_or}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_eq}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_neq}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_lt}(inv) \vee \\ & \quad (e, e') \in \text{cf\_transrel\_expr\_le}(inv) \} \end{aligned}$$

## B.2 Optimization patterns for l-value

**Definition B.14.** (*Translation relation over l-value pairs, case 1/19*)

$$\begin{aligned}
 & \text{cf\_transrel\_lv\_case1} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
 & \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
 & \text{cf\_transrel\_lv\_case1}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
 & \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set. \\
 & \quad \quad pc' = \mathbf{Suc}(pc) \wedge \\
 & \quad \quad succBP(pid, pc') = \mathbf{Some}(pid') \wedge \\
 & \quad \quad BP(pid') = \mathbf{Some}(pid', bid', 1, 0, pc') \wedge \\
 & \quad \quad BB(bid') = \mathbf{Some}(pid') \wedge \\
 & \quad \quad predB(pid') = \mathbf{Some}(set) \wedge \\
 & \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
 & \quad \quad \mathcal{A}_{CPA}(pc) = \mathbf{Some}(inv) \wedge \\
 & \quad \quad \mathcal{A}_{CPA}(pc') = \mathbf{Some}(inv') \wedge \\
 & \quad \quad e' = i \wedge \\
 & \quad \quad (e, i) \in \text{cf\_transrel\_expr}(inv) \wedge \\
 & \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s(v \mapsto i)) \}
 \end{aligned}$$

**Definition B.15.** (*Translation relation over l-value pairs, case 2/19*)

$$\begin{aligned}
 & \text{cf\_transrel\_lv\_case2} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
 & \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
 & \text{cf\_transrel\_lv\_case2}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
 & \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set. \\
 & \quad \quad pc' = \mathbf{Suc}(pc) \wedge \\
 & \quad \quad succBP(pid, pc') = \mathbf{Some}(pid') \wedge \\
 & \quad \quad BP(pid') = \mathbf{Some}(pid', bid', 1, 0, pc') \wedge \\
 & \quad \quad BB(bid') = \mathbf{Some}(pid') \wedge \\
 & \quad \quad predB(pid') = \mathbf{Some}(set) \wedge \\
 & \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
 & \quad \quad \mathcal{A}_{CPA}(pc) = \mathbf{Some}(inv) \wedge \\
 & \quad \quad \mathcal{A}_{CPA}(pc') = \mathbf{Some}(inv') \wedge \\
 & \quad \quad e' = b \wedge \\
 & \quad \quad (e, b) \in \text{cf\_transrel\_expr}(inv) \wedge \\
 & \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s(v \mapsto b)) \}
 \end{aligned}$$

**Definition B.16.** (Translation relation over l-value pairs, case 3/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case3} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case3}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ v'. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v) = \text{None} \wedge \\
& \quad \quad e' = v' \wedge \\
& \quad \quad (e, v') \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.17.** (Translation relation over l-value pairs, case 4/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case4} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case4}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ a\ i. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v) = \text{None} \wedge \\
& \quad \quad e' = a[i] \wedge \\
& \quad \quad (e, a[i]) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.18.** (*Translation relation over l-value pairs, case 5/19*)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case5} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case5}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ a\ v'. \\
& \quad \quad pc' = \mathbf{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \mathbf{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \mathbf{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \mathbf{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \mathbf{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \mathbf{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \mathbf{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \mathbf{None} \wedge \\
& \quad \quad e' = a[v'] \wedge \\
& \quad \quad (e, a[v']) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.19.** (*Translation relation over l-value pairs, case 6/19*)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case6} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case6}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \mathbf{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \mathbf{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \mathbf{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \mathbf{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \mathbf{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \mathbf{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \mathbf{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \mathbf{None} \wedge \\
& \quad \quad e' = o_1 + o - 2 \wedge \\
& \quad \quad (e, o_1 + o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.20.** (Translation relation over l-value pairs, case 7/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case7} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case7}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 - o - 2 \wedge \\
& \quad \quad (e, o_1 - o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.21.** (Translation relation over l-value pairs, case 8/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case8} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case8}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 * o - 2 \wedge \\
& \quad \quad (e, o_1 * o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.22.** (Translation relation over l-value pairs, case 9/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case9} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case9}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = -o \wedge \\
& \quad \quad (e, -o) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.23.** (Translation relation over l-value pairs, case 10/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case10} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case10}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 \wedge o - 2 \wedge \\
& \quad \quad (e, o_1 \wedge o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.24.** (Translation relation over l-value pairs, case 11/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case11} : \mathbf{BlkPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlkPosId} \times \mathbf{BlkId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case11}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = \neg o \wedge \\
& \quad \quad (e, \neg o) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.25.** (Translation relation over l-value pairs, case 12/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case12} : \mathbf{BlkPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlkPosId} \times \mathbf{BlkId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case12}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 \vee o - 2 \wedge \\
& \quad \quad (e, o_1 \vee o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.26.** (Translation relation over l-value pairs, case 13/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case13} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case13}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 = o - 2 \wedge \\
& \quad \quad (e, o_1 = o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.27.** (Translation relation over l-value pairs, case 14/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case14} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case14}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 \neq o - 2 \wedge \\
& \quad \quad (e, o_1 \neq o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.28.** (Translation relation over l-value pairs, case 15/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case15} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case15}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 < o - 2 \wedge \\
& \quad \quad (e, o_1 < o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.29.** (Translation relation over l-value pairs, case 16/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case16} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case16}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(v, v) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(v') = \text{None} \wedge \\
& \quad \quad e' = o_1 \leq o - 2 \wedge \\
& \quad \quad (e, o_1 \leq o_2) \in \text{cf\_transrel\_expr}(inv) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.30.** (Translation relation over l-value pairs, case 17/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case17} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case17}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(a[i], a[i]) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv'(a) = \text{None} \wedge \\
& \quad \quad \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s) \}
\end{aligned}$$

**Definition B.31.** (Translation relation over l-value pairs, case 18/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case18} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case18}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(a[v], a[v]) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv(v) = \text{None} \wedge \\
& \quad \quad inv'(a) = \text{None} \wedge \\
& \quad \quad \text{confcpares}(inv, s) \longrightarrow \text{confcpares}(inv', s) \}
\end{aligned}$$

**Definition B.32.** (Translation relation over l-value pairs, case 19/19)

$$\begin{aligned}
& \text{cf\_transrel\_lv\_case18} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv\_case19}((pid_0, BP, BB, succBP, predB), \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(a[v], a[i]) \mid \exists inv\ inv'\ pc'\ pid'\ bid'\ set\ o_1\ o_2. \\
& \quad \quad pc' = \text{Suc}(pc) \wedge \\
& \quad \quad succBP(pid, pc') = \text{Some}(pid') \wedge \\
& \quad \quad BP(pid') = \text{Some}(pid', bid', 1, 0, pc') \wedge \\
& \quad \quad BB(bid') = \text{Some}(pid') \wedge \\
& \quad \quad predB(pid') = \text{Some}(set) \wedge \\
& \quad \quad (bid, \mathbf{FTYPE}) \in set \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc) = \text{Some}(inv) \wedge \\
& \quad \quad \mathcal{A}_{CPA}(pc') = \text{Some}(inv') \wedge \\
& \quad \quad inv(a) = \text{None} \wedge \\
& \quad \quad inv(v) = \text{Some}(i) \wedge \\
& \quad \quad \text{confcpare}(inv, s) \longrightarrow \text{confcpare}(inv', s) \}
\end{aligned}$$

**Definition B.33.** (Translation relation over l-value pairs)

$$\begin{aligned}
& \text{cf\_transrel\_lv} : \mathbf{BlckPosEnv} \times \mathbf{InvariantEnv} \times \mathbf{Pc} \times \mathbf{BlckPosId} \times \mathbf{BlckId} \times \mathbf{State} \\
& \quad \times \mathbf{Expression} \times \mathbf{Expression} \rightarrow \mathbf{LValueTransRel\_CF} \\
& \text{cf\_transrel\_lv}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') = \\
& \quad \{(l, l') \mid (l, l') \in \text{cf\_transrel\_lv\_case1}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case2}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case3}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case4}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case5}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case6}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case7}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case8}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case9}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case10}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case11}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case12}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case13}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case14}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case15}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case16}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case17}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case18}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \vee \\
& \quad \quad (l, l') \in \text{cf\_transrel\_lv\_case19}(B, \mathcal{A}_{CPA}, pc, pid, bid, s, e, e') \}
\end{aligned}$$



## Chapter C

---

## Specification of optimization relation for nop insertion

### C.1 Assignment

#### Definition C.1.

```

ni_transrel_assign_1_1 : Program × Program × BlckPosEnv × BlckPosEnv
                        × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_assign_1_1(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_succbid trg_set.
      instrs!src_pc1 = (lval:=e) ∧
      instrs'!trg_pc1 = (val:=e) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧
      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧

      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, FTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 1, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_succbid, succbsize, 0, trg_pc2) ∧

      trg_BB(trg_succbid) = Some(trg_pid2) ∧
      trg_predB(trg_pid2) = Some(trg_set) ∧
      (trg_bid1, FTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }
end

```

**Definition C.2.**

```

ni_transrel_assign_1_2 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_assign_1_2(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_bid2 trg_pc3 trg_pid3
      trg_succbid trg_set.
      instrs!src_pc1 = (lval:=e) ∧
      instrs'!trg_pc1 = (goto(trg_pc2)) ∧
      instrs'!trg_pc2 = (val:=e) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧

      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, FTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 2, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 2, 1, trg_pc2) ∧

      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_succbid, succbsize, 0, trg_pc3) ∧

      trg_BB(trg_succbid) = Some(trg_pid3) ∧
      trg_predB(trg_pid3) = Some(trg_set) ∧
      (trg_bid1, FTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }
end

```

**Definition C.3.**

```

ni_transrel_assign_1_3 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_assign_1_3(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_bid2 trg_pc3 trg_pid3
      trg_succbid trg_set.
      instrs!src_pc1 = lval := e ∧
      instrs!trg_pc1 = goto(trg_pc2) ∧
      instrs!trg_pc2 = goto(trg_pc3) ∧
      instrs!trg_pc3 = val := e) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧

      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, FTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 3, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 3, 1, trg_pc2) ∧

      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_bid1, 3, 2, trg_pc3) ∧

      trg_pc4 = Suc(trg_pc3) ∧
      trg_succBP(trg_pid3, trg_pc4) = Some(trg_pid4) ∧
      trg_BP(trg_pid4) = Some(trg_pid4, trg_succbid, succbssize, 0, trg_pc4) ∧

      trg_BB(trg_succbid) = Some(trg_pid4) ∧
      trg_predB(trg_pid4) = Some(trg_set) ∧
      (trg_bid1, FTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }
end

```

## C.2 Printi

### Definition C.4.

```

ni_transrel_printi_1_1 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_printi_1_1(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_succbid trg_set.
      instrs!src_pc1 = (printi(e)) ∧
      instrs!trg_pc1 = (printi(e)) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧

      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, OTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 1, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_succbid, succbsize, 0, trg_pc2) ∧

      trg_BB(trg_succbid) = Some(trg_pid2) ∧
      trg_predB(trg_pid2) = Some(trg_set) ∧
      (trg_bid1, OTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), OTYPE) ∈ CBEP }
end

```

**Definition C.5.**

```

ni_transrel_printi_1_2 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_printi_1_2(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_bid2 trg_pc3 trg_pid3
      trg_succbid trg_set.
      instrs!src_pc1 = (printi(e)) ∧
      instrs'!trg_pc1 = (goto(trg_pc2)) ∧
      instrs'!trg_pc2 = (printi(e)) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧

      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, OTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 2, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 2, 1, trg_pc2) ∧

      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_succbid, succbsize, 0, trg_pc3) ∧

      trg_BB(trg_succbid) = Some(trg_pid3) ∧
      trg_predB(trg_pid3) = Some(trg_set) ∧
      (trg_bid1, OTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), OTYPE) ∈ CBEP }
end

```

**Definition C.6.**

```

ni_transrel_printi_1_3 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI

ni_transrel_printi_1_3(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ lval e src_pc2 src_pid2 src_succbid src_set trg_pc2 trg_pid2 trg_bid2 trg_pc3 trg_pid3
      trg_succbid trg_set.
      instrs!src_pc1 = printi(e) ∧
      instrs'!trg_pc1 = goto(trg_pc2) ∧
      instrs'!trg_pc2 = goto(trg_pc3) ∧
      instrs'!trg_pc3 = printi(e) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_pc2 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc2) = Some(src_pid2) ∧
      src_BP(src_pid2) = Some(src_pid2, src_succbid, 1, 0, src_pc2) ∧

      src_BB(src_succbid) = Some(src_pid2) ∧
      src_predB(src_pid2) = Some(src_set) ∧
      (src_bid1, OTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 3, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 3, 1, trg_pc2) ∧

      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_bid1, 3, 2, trg_pc3) ∧

      trg_pc4 = Suc(trg_pc3) ∧
      trg_succBP(trg_pid3, trg_pc4) = Some(trg_pid4) ∧
      trg_BP(trg_pid4) = Some(trg_pid4, trg_succbid, succbsize, 0, trg_pc4) ∧

      trg_BB(trg_succbid) = Some(trg_pid4) ∧
      trg_predB(trg_pid4) = Some(trg_set) ∧
      (trg_bid1, OTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), OTYPE) ∈ CBEP }
end

```

### C.3 Branch

#### Definition C.7.

```

ni_transrel_branch_1_1 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_branch_1_1(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ e src_dst src_pc1 src_pid1 src_succbid1 src_set1 src_pcr src_pidr src_succbidr src_setr
      trg_dst trg_pc2 trg_pid2 trg_pc1 trg_pid1 trg_succbid1 succbsize1 trg_set1
      trg_pidr trg_succbidr succbsizer trg_setr.
      instrs!src_pc1 = (branch(e, src_dst)) ∧
      instrs'!trg_pc1 = (branch(e, trg_dst)) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_pc1 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc1) = Some(src_pid1) ∧
      src_BP(src_pid1) = Some(src_pid1, src_succbid1, 1, 0, src_pc1) ∧
      src_BB(src_succbid1) = Some(src_pid1) ∧
      src_predB(src_pid1) = Some(src_set1) ∧ (src_bid1, FTYPE) ∈ src_set1 ∧

      src_succBP(src_pid1, src_dst) = Some(src_pidr) ∧
      src_BP(src_pidr) = Some(src_pidr, src_succbidr, 1, 0, src_pcr) ∧
      src_BB(src_succbidr) = Some(src_pidr) ∧
      src_predB(src_pidr) = Some(src_setr) ∧ (src_bid1, FTYPE) ∈ src_setr ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 1, 0, trg_pc1) ∧

      trg_pc1 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc1) = Some(trg_pid1) ∧
      trg_BP(trg_pid1) = Some(trg_pid1, trg_succbid1, succbsize1, 0, trg_pc1) ∧
      trg_BB(trg_succbid1) = Some(trg_pid1) ∧
      trg_predB(trg_pid1) = Some(trg_set1) ∧ (trg_bid1, FTYPE) ∈ trg_set1 ∧

      trg_succBP(trg_pid1, trg_dst) = Some(trg_pidr) ∧
      trg_BP(trg_pidr) = Some(trg_pidr, trg_succbidr, succbsizer, 0, trg_pcr) ∧
      trg_BB(trg_succbidr) = Some(trg_pidr) ∧
      trg_predB(trg_pidr) = Some(trg_setr) ∧ (trg_bid1, FTYPE) ∈ trg_setr ∧

      ((src_bid1, src_succbid1), (trg_bid1, trg_succbid1), FTYPE) ∈ CBEP ∧
      ((src_bid1, src_succbidr), (trg_bid1, trg_succbidr), FTYPE) ∈ CBEP }
end

```

**Definition C.8.**

```

ni_transrel_branch_1_2 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_branch_1_2(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ e src_dst src_pc1 src_pidi src_succbidi src_seti src_pcr src_pidr src_succbidr src_setr
      trg_pid2 trg_bid2 trg_pc2
      trg_dst trg_pc2 trg_pid2 trg_pc1 trg_pidi trg_succbidi succbsizei trg_seti
      trg_pidr trg_succbidr succbsizer trg_setr.
      instrs!src_pc1 = (branch(e, src_dst)) ∧
      instrs'!trg_pc1 = (branch(e, trg_dst)) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_pc1 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc1) = Some(src_pidi) ∧
      src_BP(src_pidi) = Some(src_pidi, src_succbidi, 1, 0, src_pc1) ∧
      src_BB(src_succbidi) = Some(src_pidi) ∧
      src_predB(src_pidi) = Some(src_seti) ∧ (src_bid1, FTYPE) ∈ src_seti ∧

      src_succBP(src_pid1, src_dst) = Some(src_pidr) ∧
      src_BP(src_pidr) = Some(src_pidr, src_succbidr, 1, 0, src_pcr) ∧
      src_BB(src_succbidr) = Some(src_pidr) ∧
      src_predB(src_pidr) = Some(src_setr) ∧ (src_bid1, FTYPE) ∈ src_setr ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid2, 2, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid2, 2, 1, trg_pc2) ∧

      trg_pc1 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid1, trg_pc1) = Some(trg_pidi) ∧
      trg_BP(trg_pidi) = Some(trg_pidi, trg_succbidi, succbsizei, 0, trg_pc1) ∧
      trg_BB(trg_succbidi) = Some(trg_pidi) ∧
      trg_predB(trg_pidi) = Some(trg_seti) ∧ (trg_bid1, FTYPE) ∈ trg_seti ∧

      trg_succBP(trg_pid1, trg_dst) = Some(trg_pidr) ∧
      trg_BP(trg_pidr) = Some(trg_pidr, trg_succbidr, succbsizer, 0, trg_pcr) ∧
      trg_BB(trg_succbidr) = Some(trg_pidr) ∧
      trg_predB(trg_pidr) = Some(trg_setr) ∧ (trg_bid1, FTYPE) ∈ trg_setr ∧

      ((src_bid1, src_succbidi), (trg_bid1, trg_succbidi), FTYPE) ∈ CBEP ∧
      ((src_bid1, src_succbidr), (trg_bid1, trg_succbidr), FTYPE) ∈ CBEP }
end

```

**Definition C.9.**

```

ni_transrel_branch_1_3 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_branch_1_3(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ e src_dst src_pc1 src_pidi src_succbidi src_seti src_pcr src_pidr src_succbidr src_setr
      trg_pid2 trg_bid2 trg_pc2 trg_pid3 trg_bid3 trg_pc3
      trg_dst trg_pc2 trg_pid2 trg_pc1 trg_pidi trg_succbidi succbsizei trg_seti
      trg_pidr trg_succbidr succbsizer trg_setr.
      instrs!src_pc1 = (branch(e, src_dst)) ∧
      instrs'!trg_pc1 = (branch(e, trg_dst)) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧
      src_pc1 = Suc(src_pc1) ∧
      src_succBP(src_pid1, src_pc1) = Some(src_pidi) ∧
      src_BP(src_pidi) = Some(src_pidi, src_succbidi, 1, 0, src_pci) ∧
      src_BB(src_succbidi) = Some(src_pidi) ∧
      src_predB(src_pidi) = Some(src_seti) ∧ (src_bid1, FTYPE) ∈ src_seti ∧
      src_succBP(src_pid1, src_dst) = Some(src_pidr) ∧
      src_BP(src_pidr) = Some(src_pidr, src_succbidr, 1, 0, src_pcr) ∧
      src_BB(src_succbidr) = Some(src_pidr) ∧
      src_predB(src_pidr) = Some(src_setr) ∧ (src_bid1, FTYPE) ∈ src_setr ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid2, 3, 0, trg_pc1) ∧
      trg_pc2 = Suc(trg_pc1) ∧ trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid2, 3, 1, trg_pc2) ∧
      trg_pc3 = Suc(trg_pc2) ∧ trg_succBP(trg_pid1, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_bid2, 3, 2, trg_pc3) ∧
      trg_pc1 = Suc(trg_pc3) ∧ trg_succBP(trg_pid1, trg_pc1) = Some(trg_pidi) ∧
      trg_BP(trg_pidi) = Some(trg_pidi, trg_succbidi, succbsizei, 0, trg_pci) ∧
      trg_BB(trg_succbidi) = Some(trg_pidi) ∧
      trg_predB(trg_pidi) = Some(trg_seti) ∧ (trg_bid1, FTYPE) ∈ trg_seti ∧
      trg_succBP(trg_pid1, trg_dst) = Some(trg_pidr) ∧
      trg_BP(trg_pidr) = Some(trg_pidr, trg_succbidr, succbsizer, 0, trg_pcr) ∧
      trg_BB(trg_succbidr) = Some(trg_pidr) ∧
      trg_predB(trg_pidr) = Some(trg_setr) ∧ (trg_bid1, FTYPE) ∈ trg_setr ∧

      ((src_bid1, src_succbidi), (trg_bid1, trg_succbidi), FTYPE) ∈ CBEP ∧
      ((src_bid1, src_succbidr), (trg_bid1, trg_succbidr), FTYPE) ∈ CBEP }
end

```

## C.4 Goto

## Definition C.10.

```

ni_transrel_goto_1_1 : Program × Program × BlckPosEnv × BlckPosEnv
                      × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_goto_1_1(S, T, BS, BT, CBEP) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BPS, BBS, succBPS, predBS) = BS;
    (pid'0, BPT, BBT, succBPT, predBT) = BT
  in
    {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
      ∃ src_dst src_piddst src_succbid src_set
        trg_dst trg_piddst trg_succbid succbsize trg_set.
        instrs!src_pc1 = (goto(src_dst)) ∧
        instrs'!trg_pc1 = (goto(trg_dst)) ∧

        src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

        src_succBP(src_pid1, src_dst) = Some(src_piddst) ∧
        src_BP(src_piddst) = Some(src_piddst, src_succbid, 1, 0, src_dst) ∧
        src_BB(src_succbid) = Some(src_piddst) ∧
        src_predB(src_piddst) = Some(src_set) ∧ (src_bid1, FTYPE) ∈ src_set ∧

        trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 1, 0, trg_pc1) ∧

        trg_succBP(trg_pid1, trg_dst) = Some(trg_piddst) ∧
        trg_BP(trg_piddst) = Some(trg_piddst, trg_succbid, succbsize, 0, trg_dst) ∧
        trg_BB(trg_succbid) = Some(trg_piddst) ∧
        trg_predB(trg_piddst) = Some(trg_set) ∧ (trg_bid1, FTYPE) ∈ trg_set ∧

        ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }
  end

```

**Definition C.11.**

```

ni_transrel_goto_1_2 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI

ni_transrel_goto_1_2(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ src_dst src_piddst src_succbid src_set
      trg_pid2 trg_bid2 trg_pc2
      trg_dst trg_piddst trg_succbid succbsize trg_set.
      instrs!src_pc1 = (goto(src_dst)) ∧
      instrs'!trg_pc1 = (goto(trg_pc2)) ∧
      instrs'!trg_pc2 = (goto(trg_dst)) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_succBP(src_pid1, src_dst) = Some(src_piddst) ∧
      src_BP(src_piddst) = Some(src_piddst, src_succbid, 1, 0, src_dst) ∧
      src_BB(src_succbid) = Some(src_piddst) ∧
      src_predB(src_piddst) = Some(src_set) ∧ (src_bid1, FTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 2, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 2, 1, trg_pc2) ∧

      trg_succBP(trg_pid2, trg_dst) = Some(trg_piddst) ∧
      trg_BP(trg_piddst) = Some(trg_piddst, trg_succbid, succbsize, 0, trg_dst) ∧
      trg_BB(trg_succbid) = Some(trg_piddst) ∧
      trg_predB(trg_piddst) = Some(trg_set) ∧ (trg_bid1, FTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }

end

```

**Definition C.12.**

```

ni_transrel_goto_1_3 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI

ni_transrel_goto_1_3(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ src_dst src_piddst src_succbid src_set
      trg_pid2 trg_bid2 trg_pc2
      trg_pid3 trg_bid3 trg_pc3
      trg_dst trg_piddst trg_succbid succbsize trg_set.
      instrs!src_pc1 = (goto(src_dst)) ∧
      instrs'!trg_pc1 = (goto(trg_pc2)) ∧
      instrs'!trg_pc2 = (goto(trg_pc3)) ∧
      instrs'!trg_pc3 = (goto(trg_dst)) ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      src_succBP(src_pid1, src_dst) = Some(src_piddst) ∧
      src_BP(src_piddst) = Some(src_piddst, src_succbid, 1, 0, src_dst) ∧
      src_BB(src_succbid) = Some(src_piddst) ∧
      src_predB(src_piddst) = Some(src_set) ∧ (src_bid1, FTYPE) ∈ src_set ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 3, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 3, 1, trg_pc2) ∧

      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_bid1, 3, 2, trg_pc3) ∧

      trg_succBP(trg_pid3, trg_dst) = Some(trg_piddst) ∧
      trg_BP(trg_piddst) = Some(trg_piddst, trg_succbid, succbsize, 0, trg_dst) ∧
      trg_BB(trg_succbid) = Some(trg_piddst) ∧
      trg_predB(trg_piddst) = Some(trg_set) ∧ (trg_bid1, FTYPE) ∈ trg_set ∧

      ((src_bid1, src_succbid), (trg_bid1, trg_succbid), FTYPE) ∈ CBEP }
end

```

## C.5 Exit

### Definition C.13.

```

ni_transrel_exit_1_1 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_exit_1_1(S, T, BS, BT, CBEP) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BPS, BBS, succBPS, predBS) = BS;
    (pid'0, BPT, BBT, succBPT, predBT) = BT
  in
    {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
      instrs!src_pc1 = exit ∧
      instrs'!trg_pc1 = exit ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 1, 0, trg_pc1) }
  end

```

### Definition C.14.

```

ni_transrel_exit_1_2 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_exit_1_2(S, T, BS, BT, CBEP) =
  let
    ((vds, instrs), I) = S;
    ((vds', instrs'), I') = T;
    (pid0, BPS, BBS, succBPS, predBS) = BS;
    (pid'0, BPT, BBT, succBPT, predBT) = BT
  in
    {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
      ∃ trg_pc2 trg_pid2 trg_bid2.
        instrs!src_pc1 = exit ∧
        instrs'!trg_pc1 = goto(trg_pc2) ∧
        instrs'!trg_pc2 = exit ∧

        src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

        trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 2, 0, trg_pc1) ∧

        trg_pc2 = Suc(trg_pc1) ∧
        trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
        trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 2, 1, trg_pc2) }
  end

```

**Definition C.15.**

```

ni_transrel_exit_1_3 : Program × Program × BlckPosEnv × BlckPosEnv
  × CorrespBlockEdgePairSet → EntryBlockPosTransRel_NI
ni_transrel_exit_1_3(S, T, BS, BT, CBEP) =
let
  ((vds, instrs), I) = S;
  ((vds', instrs'), I') = T;
  (pid0, BPS, BBS, succBPS, predBS) = BS;
  (pid'0, BPT, BBT, succBPT, predBT) = BT
in
  {((src_bid1, src_pid1, src_pc1), (trg_bid1, trg_pid1, trg_pc1)) |
    ∃ trg_pc2 trg_pid2 trg_bid2.
      instrs!src_pc1 = exit ∧
      instrs!trg_pc1 = goto(trg_pc2) ∧
      instrs!trg_pc2 = goto(trg_pc3) ∧
      instrs!trg_pc3 = exit ∧

      src_BP(src_pid1) = Some(src_pid1, src_bid1, 1, 0, src_pc1) ∧

      trg_BP(trg_pid1) = Some(trg_pid1, trg_bid1, 3, 0, trg_pc1) ∧

      trg_pc2 = Suc(trg_pc1) ∧
      trg_succBP(trg_pid1, trg_pc2) = Some(trg_pid2) ∧
      trg_BP(trg_pid2) = Some(trg_pid2, trg_bid1, 3, 1, trg_pc2) ∧

      trg_pc3 = Suc(trg_pc2) ∧
      trg_succBP(trg_pid2, trg_pc3) = Some(trg_pid3) ∧
      trg_BP(trg_pid3) = Some(trg_pid3, trg_bid1, 3, 2, trg_pc3) }
end

```



## Chapter D

---

### The companion CD

This chapter presents and explains the usage of the content of the dissertation companion CD.

#### D.1 The content

The root directory of the companion CD contains the following entries:

**README:** The `README` file contains the text, which is almost the same as the text in this chapter and describes the following:

1. The purpose and the content of each entry in the root directory of the companion CD.
2. How to install and compile the software on the CD.

**SVF:** The directory `SVF` contains all Isabelle/HOL theories that make up the implementation of the SVF presented in this dissertation. To develop these theories further, one needs the theorem prover Isabelle version 2003. Unfortunately, it is not possible to compile the scripts into a heap image using the batch mode. The reason for this is of the purely technical nature: Some constant definitions declared in the framework for the optimizations CF and NI are too large and the Isabelle/HOL is not able to deal with them properly. Fortunately, the theorem prover Isabelle/Isar, which is used in interactive mode, is able to compile these definitions. Therefore, one has to translate the theory scripts using the canonical user interface Proof General in the XEmacs, cf. Section D.4 for the details.

**compiler:** The directory `compiler` contains all SML source files, which make up the implementation of the frontend presented in this dissertation. To compile the frontend, one needs the SML compiler version 110.41, cf. Section D.2 for the details.

**diss.pdf:** The file `diss.pdf` contains the electronic version of this dissertation.

**lookup.pdf:** The file `lookup.pdf` contains the electronic version of a lemma lookup document. The purpose of this document is to enable an interested reader to find, for a definition or a theorem in the dissertation, a respective constant definition or lemma in a corresponding Isabelle/HOL theory in the `SVF` directory.

The document is created directly from the dissertation document by removing all text between definitions and theorems and keeping the structure of the original document untouched. For each entry Definition X.Y in Section X in the dissertation, the removed original text, which contained the explanation of Definition X.Y, has been replaced by the comment explaining in which Isabelle/HOL theory file in the `SVF` directory a constant definition, which formalizes Definition X.Y, can be found. The theorems in the dissertation document have been processed in an analogous manner. As a result, the lemma lookup document enables the reader, for a definition Definition X.Y in Section X in the dissertation, to quickly look up for its formalization in the `SVF`: He or she merely has to look up for Definition X.Y in Section X in the

lemma lookup document and to read the comment below the definition. The same applies to the theorems in the dissertation.

## D.2 Compiling the frontend

The frontend can be compiled as follows.

1. Copy the directory `compiler` to a directory of your choice in your site, say the directory `/home/xyz`.
2. Change working directory to `/home/xyz/compiler`.
3. Edit the variable `CURR_DIR` in the makefile `/home/xyz/compiler/Makefile` according to the situation on your site, i.e. `CURR_DIR="/home/xyz/compiler"`.
4. The makefile contains two targets, which provide the following alternatives of compilation outputs:
  - a) Type `make compiler` to create a stand-alone compiler application. The application will be accessible at the following path: `/home/xyz/compiler/bin/mikroC`.  
If you want to change the directory or the file name, you have to edit the variable `MIKROC_BINARY` in the makefile accordingly.
  - b) Type `make sml-mikroC` to create a heap image with the structure `MikroC` visible on the top level. If you want to use this structure as a starting point for a further development, you have to pass the name of the heap image file as a parameter to the SML compiler. Read the documentation of the SML compiler on how to pass the name of the heap image file as a parameter to the SML compiler. A concise description of the parameter syntax can be get by executing the command `sml -h` in the shell. In particular, you have to read the description on how to specify the argument `@SMLload`.  
The pathname of the heap image file is controlled by the variable `SML_MIKROC` in the makefile. If you want to analyse the source code of the frontend, the good starting point for this are the files `/home/xyz/compiler/root.sml` and `/home/xyz/compiler/sources.cm`.  
The file `root.sml` contains the declaration of the structure `MikroC`. The file `sources.cm` is a configuration file for the compilation manager and contains the list of all SML files in the ProGenCo project.

### D.3 Using the frontend

Edit a mikroC program. The examples of mikroC programs can be found in the directory `IL/test_programs`.

To list them, execute the following commands in the shell:

```
cd /home/xyz/compiler
find IL/test_programs/test* -name "t*.c".
```

Suppose that the filename of the file with your mikroC program is `prog.c`. Then, you have to type `/home/xyz/compiler/bin/mikroC prog.c`.

The frontend generates the following files:

1. `prog.il` contains an IL program, which is the result of translation of mikroC program contained in `prog.c` and five subsequent optimizations described in this thesis.
2. `prog.log` contains pretty printed output of some functions, which is useful for the purpose of debugging.
3. `CFcorrect.thy` contains constant definitions needed for the correctness proof of the CF optimization.
4. `CFcorrect.ML` contains the correctness proof of the CF optimization.
5. `DAEcorrect.thy` contains constant definitions needed for the correctness proof of the DAE optimization.
6. `DAEcorrect.ML` contains the correctness proof of the DAE optimization.
7. `DAIcorrect.thy` contains constant definitions needed for the correctness proof of the RAI optimization.
8. `DAIcorrect.ML` contains the correctness proof of the RAI optimization.
9. `NOPIINcorrect.thy` contains constant definitions needed for the correctness proof of the NI optimization.
10. `NOPIINcorrect.ML` contains the correctness proof of the NI optimization.
11. `RAEcorrect.thy` contains constant definitions needed for the correctness proof of the RAE optimization.
12. `RAEcorrect.ML` contains the correctness proof of the RAE optimization.

The reader should note that currently our frontend does not generate a theory file with a proof combining the results proved in the above theory files in order to derive a statement about the optimization correctness of the whole optimization chain that consists of the optimizations CF, DAE, RAI, NI, and RAE (as described in this dissertation). The reason for this is that our framework is in a highly experimental stage of development and due to this fact, and from the lack of time, there was no need to implement a proof generation module which performs this proof step. The proof of the aforementioned statement, however, is trivial and consists of only one proof step. Therefore, in our work, we always wrote it by hand working with Isabelle in the interactive mode.

The next section describes how to verify the Isabelle/HOL theories in the above files using the theorem prover Isabelle/HOL.

### D.4 Using the SVF

To verify the proofs in the theory files described in the previous section, one has to perform the following steps:

1. Copy the directory  
SVF

- to a directory of your choice in your site, say the directory `/home/xyz`.
2. Change working directory to `/home/xyz/SVF`.
  3. The directory contains the following entries:
    - a) `ROOT.ML`: This is a standard configuration file, which used by Isabelle when generating heap image file containing a logic. It can be used to create a logic, which comprises the theories from Layers 1 through 4, cf. the remark on generating heap image files using Isabelle in the batch mode in Section D.1.
    - b) `Setup.thy`: This theory is a workaround, which is used to set proper paths to all theories in the SVF for Isabelle. Always process this theory in Proof General first, if you want to work with the SVF in the interactive mode.
    - c) `Layer1_HOLBasics`: This directory contains theory files, which contain Isabelle/HOL formalizations concepts from Layer 1 in the SVF.
    - d) `Layer2_TranslationContract`: This directory contains theory files, which contain Isabelle/HOL formalizations concepts from Layer 2 in the SVF.
    - e) `Layer3_TypeSafety`: This directory contains theory files, which contain Isabelle/HOL formalizations concepts from Layer 3 in the SVF.
    - f) `Layer4_OptimizationIndependentTranslCorrCriterion`: This directory contains theory files, which contain Isabelle/HOL formalizations concepts from Layer 4 in the SVF.
    - g) `Layer5_OptimizationDependent_TranslCorrCriteria`: This directory contains theory files, which contain Isabelle/HOL formalizations concepts from Layer 5 in the SVF.
    - h) `Layer6_ProofEnvironmentsForParticularProofTasks`: This directory contains theory files, which contain Isabelle/HOL formalizations concepts from Layer 6 in the SVF.
  4. Edit `SVF/ROOT.ML`:
  5. Edit `SVF/Setup.thy`:
  6. Open the file `Setup.thy` in the user interface Proof General by executing the following command in the shell:
 

```
xemacs Setup.thy &
```
  7. Edit the variable `mypath2progenco` according to the situation on your site, i.e.
 

```
val mypath2progenco = "/home/xyz/SVF/".
```
  8. Start Isabelle/HOL (*not* Isabelle/Isar) using default logic HOL by opening an arbitrary proof script file with the extension ML, say the file `/home/xyz/SVF/ROOT.ML`.
  9. Open the file `Setup.thy`
  10. Process the whole theory `Setup` by clicking the button `Use`.
  11. Open file `Layer6_ProofEnvironmentsForParticularProofTasks/SVFramework_IL_main.thy`
  12. Process the theory `SVFramework_IL_main` by clicking the button `Use`. Processing this theory takes approximately 6 hours for the standard computer with the Linux, the microprocessor 1.6 MHz, and 512 MB RAM.
  13. Now, you are ready to verify proofs generated by the frontend. We explain how to perform this by example of the CF optimization.
    - a) Given that the frontend has generated the correctness proof in the scripts `/home/xyz/myproject/CFcorrect.thy` and `/home/xyz/myproject/CFcorrect.ML`, than you have to open the former file and to process the whole theory in this file.
    - b) Switch to the latter file by pressing the shortcut `ctrl-c`, `ctrl-b`.

- c) Process the lemmas in this buffer, for example by pressing repeatedly the shortcut `ctrl-c`, `ctrl-n`.

Analogous has to be done for other optimizations.



---

## References

1. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.
2. C.M. Angelo, L. Claesen, and H. De Man. Degrees of formality in shallow embedding hardware description languages in HOL. In J.J. Joyce and C.-J.H. Seger, editors, *Theorem Proving in Higher Order Logic (TPHOL)*, volume 780 of *LNCS*, pages 89–100, Vancouver, Canada, 1994. Springer.
3. Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
4. Andrew W. Appel. Foundational proof-carrying code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247, Washington, DC, USA, 2001. IEEE Computer Society.
5. Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 243–253, New York, NY, USA, 2000. ACM Press.
6. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
7. C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A translation validator for optimizing compilers. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, July 2005.
8. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. *SIGPLAN Not.*, 23(7):329–338, 1988.
9. Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–25, New York, NY, USA, 2004. ACM.
10. Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In Wolf Zimmermann Jens Knoop, editor, *COCV'03, Compiler Optimization Meets Compiler Verification*, volume 82 of *ENTCS*, pages 377–394. Elsevier, April 2004.
11. Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development (Coq'Art: The Calculus of Inductive Constructions)*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, 2004.
12. Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs, Workshop TYPES 2004*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2006.
13. W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
14. William R. Bevier, Warren A. Hunt, J. Strother Moore, and William Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4), 1989.
15. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

16. Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.
17. Jan Olaf Blech, Sabine Glesner, Johannes Leitner, and Steffen Mülling. A comparison between two formal correctness proofs in Isabelle/HOL. In *Proceedings of the COCV-Workshop (Compiler Optimization meets Compiler Verification)*, 8th European Conferences on Theory and Practice of Software (ETAPS 2005). Elsevier, April 2005.
18. Jan Olaf Blech and Benjamin Gregoire. Certifying code generation with coq. In *Proceedings of the 7th Workshop on Compiler Optimization meets Compiler Verification (COCV 2008)*, Budapest, Hungary, to appear in ENTCS, 2008.
19. Jan Olaf Blech and Arnd Poetsch-Heffter. A certifying code generation phase. In *Proceedings of the 6th Workshop on Compiler Optimization meets Compiler Verification (COCV 2007)*, Braga, Portugal, volume 190 of ENTCS, pages 65–82, November 2007.
20. Jan Olaf Blech, Ina Schaefer, and Arnd Poetsch-Heffter. Translation validation for system abstractions. In *7th Workshop on Runtime Verification (RV'07)*, Vancouver, Canada, volume 4839. Springer, March 2007.
21. Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing, Seattle, Washington, 15–17 May 1989*, pages 86–97, 1989.
22. Manuel Blum and Sampath Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.
23. Egon Börger. Ten years of Gurevich’s abstract state machines. *Journal of Universal Computer Science*, 3(4):230–232, 1997.
24. Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design: Proceedings of the IFIP TC10/WG 10.2 International Conference*, pages 129–156. North-Holland, 1992.
25. R.J. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J.P. van Tassel. Experience with embedding hardware description languages in HOL. In *Conference on Theorem Provers in Circuit Design (TPCD)*, pages 129–156, Nijmegen, 1992. North Holland.
26. R. S. Boyer, M. Kaufmann, and J.S. Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.
27. Robert S. Boyer and J. Strother Moore. *The Correctness Problem in Computer Science*. Academic Press, Inc., Orlando, FL, USA, 1982.
28. Manfred Broy, Ursula Hinkel, Tobias Nipkow, Christian Prehofer, and Birgit Schieder. Interpreter verification for a functional language. In P.S. Thiagarajan, editor, *Proc. 14th Conf. Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 77–88. Springer Verlag, 1994.
29. Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.
30. W. J. Cullyer. Implementing high integrity systems: the VIPER microprocessor. *Aerospace and Electronic Systems Magazine, IEEE*, 4(6):5–13, 1989.
31. Paul Curzon. A programming logic for a verified structured assembly language. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 403–408. Springer-Verlag, 1992.
32. Paul Curzon. A verified compiler for a structured assembly language. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*. IEEE Computer Society Press, 1992.
33. Paul Curzon. Compiler correctness and input/output. In C.E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications 3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 189–209. Springer-Verlag, 1993.
34. Paul Curzon. A verified Vista implementation. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993.
35. Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

36. A. Dold and V. Vialard. Mechanically verified compiling specification for a lisp compiler. In *Proc. FSTTCS 2001*, December 2001.
37. R. W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, American Math. Soc., pages 19–32, 1967.
38. S. Glesner. Using program checking to ensure the correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, 2003. [http://www.jucs.org/jucs\\_9\\_3/using\\_program\\_checking\\_to](http://www.jucs.org/jucs_9_3/using_program_checking_to).
39. S. Glesner, G. Goos, F. v. Henke, H. Langmaack, W. Goerigk, and W. Zimmermann. Abschlussbericht verifix. Technical report, Universitäten Karlsruhe, Kiel, Ulm, July 2004. Bericht zur Vorlage bei der Deutschen Forschungsgemeinschaft.
40. Sabine Glesner. Program checking with certificates: Separating correctness-critical code. In *Proceedings of the 12th International FME Symposium (Formal Methods Europe)*, volume 2805 of *Lecture Notes in Computer Science*, pages 758–777, Pisa, Italy, September 2003. Springer Verlag.
41. Sabine Glesner, Gerhard Goos, and Wolf Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). *it - Information Technology*, 46(Issue 5/2004):265 – 276, May 2004.
42. Wolfgang Goerigk and Ulrich Hoffmann. Rigorous compiler implementation correctness: How to prove the real thing correct. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pages 122–136, London, UK, 1999. Springer-Verlag.
43. Gerhard Goos and Wolf Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rüdiger Olderog, editors, *Correct System Design*, volume 1710, pages 201–230. Springer, Nov 1999.
44. M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 73–128. Springer-Verlag, 1989.
45. M.C.J. Gordon. HOL – a machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.
46. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
47. Michael J.C. Gordon. From LCF to HOL: a short history. In Mads Tofte (Editor) G. Plotkin (Editor), Colin P. Stirling (Editor), editor, *Proof, Language, and Interaction*. MIT Press, 2000.
48. Yuri Gurevich. *Specification and validation methods*, chapter Evolving algebras 1993: Lipari guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.
49. Yuri Gurevich and Marc Spielmann. Recursive abstract state machines. *Journal of Universal Computer Science*, 3(4):233–246, apr 1997. [http://www.jucs.org/jucs\\_3\\_4/recur\\_siv\\_abstract](http://www.jucs.org/jucs_3_4/recur_siv_abstract).
50. Joshua D. Guttman, John D. Ramsdell, and Vipin Swarup. The VLISP verified scheme system. *Lisp Symb. Comput.*, 8(1-2):33–110, 1995.
51. Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. Vlisip: a verified implementation of scheme. *Lisp Symb. Comput.*, 8(1-2):5–32, 1995.
52. F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: A case study. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 179–213. North-Holland, 1986.
53. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
54. W. A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.
55. Intel. *Intel Architecture Software Developer's Manual*. Intel Corporation, 1997.
56. J. Kershaw. Vista user's guide. technical report 40186. Technical report, The Royal Signals and Radar Establishment, 1986.
57. Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13:1133–1151, 2001.
58. Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298:583–626, 2003.
59. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

60. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006.
61. David Lacey. *Program Transformation using Temporal Logic Specifications*. PhD thesis, Oxford University Computing Laboratory, 2003.
62. David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. *ACM SIGPLAN Notices*, 37(1):283–294, January 2002.
63. David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order Symbol. Comput.*, 17(3):173–206, 2004.
64. Sorin Lerner. *Automatically Proving the Correctness of Program Analyses and Transformations*. PhD thesis, University of Washington, 2006.
65. Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*, San Diego, California, June 9-11 2003.
66. Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Conference Record of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, Long Beach, California, January 12-14 2005.
67. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
68. Xavier Leroy. The Compcert verified compiler, commented Coq development. Available at <http://compcert.inria.fr/doc/>, March 2008.
69. Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 2008. Accepted for publication, to appear.
70. Tim Lindholm and Frank Yellin. *The Java<sup>TM</sup> Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
71. John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, Providence, RI, 1967.
72. Arthur J. Milner Michael J. Gordon and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
73. J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning, Kluwer Academic Publishers*, 5(4):461–492, 1989.
74. F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 144–152, New York, NY, USA, 1973. ACM.
75. George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.
76. George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
77. George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–95, 2000.
78. George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 229–243, Seattle, Washington, October 28–31 1996.
79. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.
80. Tobias Nipkow. Term rewriting and beyond — theorem proving in Isabelle. *Formal Aspects of Computing*, 1:320–338, 1989.
81. Tobias Nipkow. Verified lexical analysis. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 1998. Invited talk.

82. Tobias Nipkow. Verified bytecode verifiers. In M. Miculan F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer Verlag, 2001.
83. Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proof Technology and Computation*, pages 247–277. IOS Press, 2006.
84. Tobias Nipkow, David von Oheimb, and Cornelia Pusch.  $\mu$ Java: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.
85. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
86. Tobias Nipkow and David von Oheimb.  $\text{Java}_{\text{light}}$  is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.
87. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. <http://www4.in.tum.de/~oheimb/diss/>.
88. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999. <http://isabelle.in.tum.de/Bali/papers/Springer98.html>.
89. Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The vliisp verified prescheme compiler. *Lisp Symb. Comput.*, 8(1-2):111–182, 1995.
90. Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computer Science 2. Cambridge University Press, 1987.
91. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, USA, 1994.
92. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151+, 1998.
93. Arnd Poetzsch-Heffter. Shades of translation correctness. Talk given at Dagstuhl Seminar 05311: "Verifying Optimizing Compilers", 31.07.-05.08.05, available in <http://kathrin.dagstuhl.de/-files/Materials/05/05311/05311.PoetzschHeffterArnd.Slides.ppt>.
94. Arnd Poetzsch-Heffter and Marek J. Gawkowski. Towards proof generating compilers. In W. Zimmermann J. Knoop, G. C. Necula, editor, *Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004)*, volume 132 of *Electronic Notes in Theoretical Computer Science*, pages 37–51. Elsevier B. V., May 2005.
95. W. M. McCracken R. A. DeMillo, R. J. Martin and J. S. Passafiume. *Software Testing and Evaluation*. The Benjamin Cummings Publishing Company, Redwood City, 1987.
96. R. Reetz. Deep embedding VHDL. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Theorem Proving in Higher Order Logic (TPHOL)*, volume 971 of *LNCS*, pages 277–292, Aspen Grove, Utah, USA, 1995. Springer.
97. R. Reetz and T. Kropf. Simplifying deep embedding: A formalised code generator. In T.F. Melham and J. Camilleri, editors, *Theorem Proving in Higher Order Logic (TPHOL)*, volume 859 of *LNCS*, pages 378–390, Valetta, Malta, 1994. Springer.
98. Erika Rice, Sorin Lerner, and Craig Chambers. Automatically inferring sound dataflow functions from dataflow fact schemas. In G. C. Necula J. Knoop and W. Zimmermann, editors, *Proceedings of the Fourth International Workshop on Compiler Optimization meets Compiler Verification (COCV 2005)*, volume 141, Issue 2, Edinburgh, UK, 02 April 2005. Elsevier B.V.
99. Laurence Rideau, Bernard P. Serpette, and Xavier Leroy. Tilting at windmills with Coq: formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.
100. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.
101. Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT Laboratory for Computer Science, March 1999.
102. Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

103. H. Samet. *Automatically proving the correctness of translations involving optimized code*. PhD thesis, Computer Science Department, Stanford University, 1975.
104. H. Samet. Compiler testing via symbolic interpretation. In *Proceedings of the ACM 29th Annual Conference*, pages 492–497, Houston, TX, October 1976.
105. H. Samet. A normal form for compiler testing. In *Proceedings of the SIGART SIGPLAN Symposium on Artificial Intelligence and Programming Languages*, pages 155–162, Rochester, NY, August 1977.
106. H. Samet. A canonical form algorithm for proving equivalence of conditional forms. *Information Processing Letters*, 7(2):103–106, February 1978.
107. Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.
108. James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. *Automata, Languages and Programming*, chapter More on advice on structuring compilers and proving them correct. Springer Berlin / Heidelberg, 1979.
109. Robert van Engelen, David Whalley, and Xin Yuan. Validation of code-improving transformations for embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 684–691, New York, NY, USA, 2003. ACM.
110. Robert van Engelen, David Whalley, and Xin Yuan. Automatic validation of code-improving transformations on low-level program representations. *Sci. Comput. Program.*, 52(1-3):257–280, 2004.
111. Robert van Engelen, Lex Wolters, and Gerard Cats. CTADEL: A generator of multi-platform high performance codes for PDE-based scientific applications. In *International Conference on Supercomputing*, pages 86–93, 1996.
112. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001. <http://isabelle.in.tum.de/Bali/papers/CPE01.html>.
113. Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.
114. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logic (TPHOL)*, volume 3223 of *LNCS*, pages 305–320, Park City, Utah, USA, 2004. Springer.
115. Martin Wildmoser, Amine Chaieb, and Tobias Nipkow. Bytecode analysis for proof carrying code. In *Proceedings of the 1st Workshop on Bytecode Semantics, Verification and Transformation*, volume 141 of *Electronic Notes in Computer Science*, pages 19–34, 2005.
116. Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer-Verlag, 2004.
117. Martin Wildmoser and Tobias Nipkow. Asserting bytecode safety. In Mooly Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, volume 3444 of *LNCS*, pages 326–341. Springer Verlag, 2005.
118. Martin Wildmoser, Tobias Nipkow, Gerwin Klein, and Sebastian Nanz. Prototyping proof carrying code. In J.-J. Levy, E. Mayer, and J. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 333–347. Kluwer, 2004.
119. R. Wilhelm and D. Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer Verlag, second edition, 1997.
120. Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. MIT, 1993.
121. William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4), 1989.
122. Wolf Zimmermann. On the correctness of transformations in compiler back-ends. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods. First International Symposium, ISoLA 2004, Paphos, Cyprus, October 30 - November 2, 2004*, volume 4313 of *Lecture Notes in Computer Science*, pages 74–95. Springer Berlin/Heidelberg, 2006.
123. Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler back-ends: An asm-approach. *Journal of Universal Computer Science*, 3(5):504–567, may 1997. [http://www.jucs.org/jucs\\_3\\_5/correct\\_compiler](http://www.jucs.org/jucs_3_5/correct_compiler).

124. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. In *COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002)*, volume 65 of *Electronic Notes in Theoretical Computer Science*, pages 1–17, April 2002.
125. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003. [http://www.jucs.org/jucs\\_9\\_3/voc\\_a\\_methodology\\_for](http://www.jucs.org/jucs_9_3/voc_a_methodology_for).
126. L. Zuck, A. Pnueli, and R. Leviathan. Validation of optimizing compilers,. Technical report, Weizmann Institute, Jerusalem, Israel, Israel, 2001.



# Lebenslauf

## Persönliche Daten

Name: GAWKOWSKI  
Vornamen: MAREK, JERZY

Geburtsdatum: 15.05.1965  
Geburtsort: Stettin, Polen  
Familienstand: verheiratet  
Staatsangehörigkeit: deutsch

Anschriften: Davenportplatz 15, 67663 Kaiserslautern,  
(bis zum 30.11.2008)  
ul. Pogodna 25, 05-502 Wólka Kozodawska, Polen,  
(ab dem 01.12.2008)

Telefonnummern: +49 631 303 66 59 (bis zum 30.11.2008)  
+48 22 736 24 28 (ab dem 01.12.2008)

Handynummer: +49 176 222 64 330 (bis zum 20.01.2009)

Email: gawkowsk@informatik.uni-kl.de

## Ausbildung

1972-1980 Besuch der Jan-Kusociński Grundschule Nr. 6 in Stettin  
1980-1984 Besuch des Adam-Asnyk Lyzeums Nr. 5 in Stettin  
05/1984 Abitur am Adam-Asnyk Lyzeum Nr. 5 in Stettin  
1984-1995 Studium der Sportwissenschaft auf Lehramt an der Universität  
Stettin  
12/1995 Abschluss als Sportlehrer mit dem Magistertitel  
1996-2003 Studium der Informatik an der Universität Freiburg  
02/2003 Abschluss als Diplom-Informatiker

## Beruflicher Werdegang

03/2006 - 10/2008 wissenschaftlicher Mitarbeiter, Fachbereich Informatik,  
AG Softwaretechnik (Prof. Dr. A. Poetzsch-Heffter)  
Technische Universität, Kaiserslautern

Kaiserslautern, 25.11.2008

