# ART

Modularisierung von Induktionsbeweisen
über
Gleichungsspezifikationen

## Dokumentation

Robert Eschbach
Universität Kaiserslautern
Erwin Schrödinger Straße
D-67663 Kaiserslautern

# Inhaltsverzeichnis

**Zusammenfassung**

Das System ART (ASF RRL Translation) stellt im wesentlichen eine Umgebung dar, in welcher die Modularisierbarkeit von Beweisen (Induktionsbeweisen über Gleichungsspezifikationen) untersucht werden kann. Es wurde die bereits bestehende Spezifikationsprache ASF (siehe [BeHeKl89]), in welcher modularisierte Spezifikationen möglich sind, so erweitert, daß zusätzlich auch Beweisaufgaben spezifiziert werden können. Im folgenden wird diese erweiterte Spezifikationsprache auch ASF genannt. Als Beweiser für die Beweisaufgaben einer Spezifikation wurde RRL (siehe [KaZh89]) gewählt. RRL kann sowohl Kommandos aus einem File abarbeiten, wie auch Sitzungsprotokolle anfertigen, mit deren Hilfe sich die Beweisverläufe und Benutzereingaben der entsprechenden RRL-Sitzung rekonstruieren lassen. In ART kann nun eine ASF-Spezifikation, die Beweisaufgaben umfassen kann, in ein File übersetzt werden, welches von RRL abgearbeitet werden kann. Dies wird im folgenden kurz mit 'Übersetzung von ASF nach RRL' bezeichnet. Bei der Abarbeitung eines solchen Files wird von RRL ein Sitzungsprotokoll angelegt. ART kann dieses Sitzungsprotokoll dazu heranziehen, neue Ergebnisse, wie etwa den erfolgreichen Beweis einer Beweisaufgabe, zu ermitteln, um diese Ergebnisse der ursprüngliche Spezifikation hinzuzufügen. Dies wird im folgenden kurz mit 'Rückübersetzung von RRL nach ASF' bezeichnet. Im Kern besteht ART also aus einer Komponente zur Übersetzung von ASF nach RRL und aus einer Komponente zur Rückübersetzung von RRL nach ASF.

# Kapitel 1

# Einführung

Es sollte untersucht werden, inwieweit sich Beweise über Gleichungsspezifikationen modularisieren lassen. Dazu mußte ein Weg gefunden werden, Beweisaufgaben in modularisierte Spezifikationen zu integrieren, um diese Spezifikationen dann in einer noch zu präzisierenden Art und Weise einem Beweiser zu übergeben. Erfolgreiche Beweise solcher Beweisaufgaben sollten dann zu einer Erweiterung der Spezifikation führen.
Es stellen sich folgende zentrale Fragen :

*Wie wird eine solche modularisierte Spezifikation repräsentiert ?*
Hierzu wurde die bereits bestehende Spezifikationssprache ASF (Algebraic Specification Formalism, siehe [BeHeKl89]) um ein Konzept erweitert, das es ermöglicht, Beweisaufgaben zu spezifizieren. Dieser so erweiterte Formalismus erlaubt nun (modularisierte) Spezifikationen mit Beweisaufgaben. Eine Spezifikation besteht dann aus Modulen, in denen Beweisaufgaben enthalten sein können. Dieser Formalismus (er wird im folgenden mit ASF bezeichnet) wird im Kapitel ASF beschrieben.

*Welcher konkrete Beweiser soll die Beweisaufgaben einer Spezifikation behandeln ?*
Die Wahl fiel auf den Beweiser RRL (Rewrite Rule Laboratory, siehe [KaZh89]), da dieser neben verschiedenen Beweismethoden auch über Möglichkeiten verfügt, Kommandos aus einem File abzuarbeiten und Informationen über die Abarbeitung von Beweisen in einem File zu protokollieren. Insgesamt erlaubt RRL also eine Kommunikation über Files, d.h. Übergabe der Spezifikation durch ein Kommando-File, Ermittlung des Beweisverlaufs durch das Protokoll. Es wird im Folgenden davon ausgegangen, daß sich der Leser mit RRL auskennt.

*Wie sieht die Übergabe der Spezifikation an RRL aus ?*
Wie oben schon angedeutet, soll über Files mit RRL kommuniziert werden. Das heißt konkret, daß die Spezifikation in ein Kommando-File für RRL umgeformt werden muß. Dazu muß im wesentlichen die Modularisierung der Spezifikation aufgelöst werden. Die so gewonnene 'flache' Spezifikation kann dann zur Erzeugung der Kommando-Files herangezogen werden. Die bei der Normalisierung einer Spezifikation auftretenden Probleme werden im Kapitel Aufgabenstellung näher beleuchtet.

*Welche Beweismethode von RRL soll unterstützt werden ?*
Hier wurden aus den in RRL zur Verfügung stehenden Methoden die Methoden *Induktionslose Induktion mit Test-Mengen* (siehe [La81], [KaNaZh86]) und *Cover-Set Induktion* (siehe [ZhKaKr88]) ausgewählt. Das hat zur Konsequenz, daß für eine ASF-Spezifikation zwei Kommando-Files für RRL erzeugt werden müssen, da beide Beweismethoden unterschiedliches Vorgehen verlangen.

*Wie kann ermittelt werden, ob die Beweisaufgaben erfolgreich oder nicht erfolgreich bewiesen werden konnten ?*
In den Kommando-Files wird verankert, daß RRL bei seiner Abarbeitung ein Sitzungsprotokoll anlegt, d.h. RRL schreibt alle relevanten Informationen, wie etwa den erfolgreichen Beweis einer Beweisaufgabe, in ein Protokoll-File. Dieses Protokoll dient dann zur Bestimmung der gewünschten Informationen, nämlich der Ermittlung der Beweisverläufe. Das Ermitteln dieser Informationen wird im Kapitel Aufgabenstellung genauer beschrieben.

*In welcher Form soll die ursprüngliche Spezifikation erweitert werden, wenn bekannt ist, welche Beweisaufgaben bewiesen und welche nicht bewiesen werden konnten ?*
Da eine Spezifikation aus Modulen besteht, ist die eigentliche Frage, welches Modul oder welche Module erweitert werden sollen. Eine Möglichkeit ist, das oberste Modul der Spezifikation zu erweitern, eine andere, die jeweilige Information in der Hierarchie der Module so tief wie möglich anzusiedeln. Die zweite Möglichkeit scheint die adäquate zu sein, da hier Informationen an die 'richtigen' Stellen gelangen, aber sie birgt in subtiler Form Gefahren in sich, die sich über die Beweis-Strategie des Beweisers RRL einschleichen: RRL versucht eine zu beweisende Gleichung als erstes mit den bestehenden Regeln zu simplifizieren. Wird nun nach der zweiten Möglichkeit verfahren, können auf einmal Simplifikationen bei den Beweisen der in der Spezifikation verankerten Beweisaufgaben möglich werden, die vorher nicht möglich waren. Im schlimmsten Fall kann eine solche Simplifikation einen Beweis, der vorher erfolgreich war, zum Scheitern bringen, etwa weil eine 'gute' Induktionsstelle 'wegsimplifiziert' wurde. Erweitert man jedoch immer das oberste Modul, läuft man nicht in Gefahr zusätzliche 'riskante' Simplifikationen möglich zu machen, aber andererseits werden zusätzliche 'nüzliche' Simplifikationen (eine Beweisaufgabe kann auf einmal durch eine solche Simplifikation erfolgreich bewiesen werden) unmöglich. Trotz diesen offensichtlichen Nachteils wurde auf Nummer Sicher gegangen und die erste Möglichkeit verwirklicht, d.h. es wird jeweils das oberste Modul einer Spezifikation erweitert. Die Erweiterung eines Moduls einer Spezifikation wird von ART durch das Erzeugen eines neuen Files realisiert, dabei wird die Verwaltung der erzeugten Files von ART unterstützt.

Diese Arbeit ist wie folgt gegliedert :

Dieses Kapitel **Einführung** erläutert grob das System ART.

In dem Kapitel **ASF** wird die Spezifikationssprache ASF eingeführt und an einem Beispiel verdeutlicht.

Das Kapitel **Aufgabenstellung** erläutert die dem System zugrundeliegende Aufgabenstellung. Die Aufgabenstellung gliedert sich im wesentlichen in drei Teile: die Übersetzung einer ASF-Spezifikation nach RRL, die Rückübersetzung von RRL nach ASF und die von ART verwaltete Filestruktur, welche die Verzeichnisstruktur und die Namensgebung der in ART entstehenden Files festlegt.

Das Kapitel **ART** beschreibt zum einen die Verzeichnisstruktur der System-Files und die Implementation des Systems ART in LISP, d.h. es werden die Hauptfunktionen beschrieben, und zum anderen die Bedienung, also wie mit dem System konkret gearbeitet wird.

In dem Kapitel **Beispiel : Primfaktorzerlegung** wird die Modularisierung eines von RRL erzeugten Beweises, welcher die Eindeutigkeit der Primfaktorzerlegung zum Hauptresultat hat, beschrieben, d.h. der ursprüngliche Beweis und die zugehörigen, von ART erzeugten ASF-Files der Modularisierung werden angegeben.

Das Kapitel **Erfahrungen** beschäftigt sich mit den beim Arbeiten mit ART gesammelten Erfahrungen über Schwierigkeiten bei der Modularisierung, Schwierigkeiten mit RRL, usw.

Im **Anhang** finden sich sowohl Syntax der ASF-Grammatik, wie auch genauere Informationen zur Implementation (etwa globale Variablen, Datenstrukturen, ...).

# Kapitel 2

# ASF

## 2.1 Einführung

Die hier vorgestellte Spezifikationssprache ist stark angelehnt an die in [BeHeKl89] beschriebene Spezifikationssprache ASF (Algebraic Specification Formalism) , und wird deshalb im folgenden auch mit ASF bezeichnet. Es soll hier nicht der Versuch gemacht werden, ASF ganz zu erklären (hierzu ist die Lektüre von [BeHeKl89] besser geeignet) , sondern es sollen lediglich die wesentlichen Punkte von ASF vorgestellt werden. In ASF lassen sich modularisierte Spezifikationen formulieren. Eine ASF-Spezifikation besteht aus einer Menge von ASF-Modulen, welche über Import- und Export-Anweisungen miteinander verbunden sind. In einer ASF-Spezifikation lassen sich folgende Angaben machen :

**Signatur (exportiert/versteckt)**
Unterschieden wird zwischen der exportierten und der 'versteckten' Signatur. Alle Sorten und Funktionen, welche sich in der exportierten Signatur befinden, können von 'außen' (also von anderen Modulen) gesehen werden. Alle anderen Sorten und Funktionen sind 'versteckt', d.h. sie sind nur in dem Modul sichtbar, in dem sie deklariert sind. Desweiteren kann man in der Signatur angeben, ob ein Operator ein Infix-Operator oder ein monadischer Präfix-Operator ist.

**Import anderer ASF-Module**
In einem Modul können andere Module importiert werden. Möchte man die vordefinierten booleschen Funktionen von RRL verwenden, geschieht dies durch den Import von `boolean` .

**Variablen**
Alle in einem Modul verwendeten Variablen (in Gleichungen, Theoremen, ...) müssen deklariert werden. Da ASF die Möglichkeit bietet, Funktionen zu überladen, sind die Sorten der Variablen notwendig, um eine Spezifikation eine eindeutige Semantik zu geben. Über die Sorten-Information von Konstanten und Variablen kann sozusagen von innen nach außen in einem Term entschieden werden, welche Funktion gemeint ist.

**Gleichungen (bedingt/unbedingt)**
Sowohl unbedingte wie auch bedingte Gleichungen können in ASF dargestellt werden. Die Gleichungen können zusätzlich mit einem `tag` (einen der Gleichung zugeordneten Name) zur besseren Lesbarkeit versehen werden.

**Beweisaufgaben**
Die Beweisaufgaben lassen sich aufteilen in

- Theoreme (Beweisaufgaben, die bewiesen werden konnten)
- echte Beweisaufgaben (Beweisaufgaben, die noch bewiesen werden müssen)
- Hypothesen (Gleichungen, die wie Theoreme behandelt werden, aber nicht bewiesen wurden)

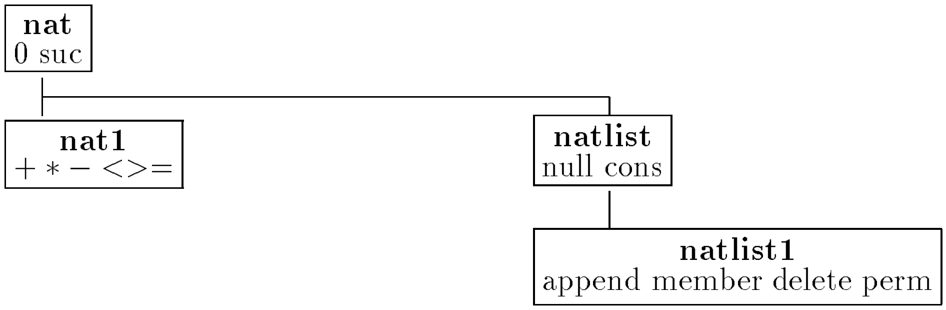
Dabei können Beweisaufgaben entweder exportiert werden, d.h. sie sind von 'außen' sichtbar, oder 'versteckt' sein, d.h. sie sind nur in dem Modul sichtbar, in welchem sie angegeben sind. Die Unterteilung der Beweisaufgaben in Theoreme, echte Beweisaufgaben und Hypothesen und ihr Sichbarkeitsbereich (exportiert, 'versteckt') wird durch Flags erreicht.
Folgende Beweisaufgabe `[T1] x + y == y + x [e]` ist eine echte Beweisaufgabe, die exportiert wird (Flag e), dabei steht 'e' für 'export'. Die Beweisaufgabe `[T2] (x * (y + z)) == ((x * y) + (x * z)) [o,e]` ist ein Theorem (Flag o) , welches exportiert wird (Flag e), dabei steht 'o' für 'ok'.
Die Beweisaufgabe `[T3] 0 < x == not(x = 0) [h]` ist eine Hypothese (Flag h), die 'versteckt' ist (sie hat nicht das Flag e), dabei steht 'h' für 'hypothesis'.

**C-Operatoren, AC-Operatoren, Ordnung, Konstruktoren**
Es kann in der Spezifikation angegeben werden, daß ein Operator kommutativ bzw kommutativ und assoziativ ist. Da in den Beweismethoden von RRL, die ART unterstützt, Gleichungen gemäß einer *lrpo* (lexicographic recursive path ordering) orientiert werden (siehe [De82], [De87]), können die für die *lrpo* notwendigen Angaben, also Status und Präzedenz, gemacht werden. Bei beiden Methoden wird von RRL die vollständige Definiertheit der Nicht-Konstruktoren auf den Konstruktor-Grundtermen überprüft (siehe [KaNaZh87]). Deshalb ist es ebenfalls möglich anzugeben, ob ein Operator ein (freier, nicht freier) Konstruktor ist.

## 2.2 Beispiel

An einem größeren Beispiel sollen nun die Möglichkeiten von ASF demonstriert werden. Das Beispiel umfaßt vier Spezifikationen, von denen sich zwei mit den natürlichen Zahlen und die restlichen zwei mit Listen über natürlichen Zahlen beschäftigen. Die Spezifikation **nat** enthält im wesentlichen die Konstruktoren **0, suc** für die natürlichen Zahlen, die Spezifikation **natlist** die Konstruktoren **null, cons** für die Listen über natürliche

Zahlen. Dabei sind alle Konstruktoren frei. Die übrigen Spezifikationen **nat1** bzw. **natlist1** erweitern die Signatur von **nat** um die Standardfunktionen `+ , *, -, < , >=` bzw. **natlist** um die Standardfunktionen `append, member, delete, perm` mit den üblichen Gleichungen. (Die Funktion `perm` entscheidet für zwei Listen, ob sie bis auf Permutation gleich sind.) Desweiteren enthalten die Spezifikationen **nat1** und **natlist1** Beweisaufgaben, die exportiert werden, d.h. sie sind in einem Modul, welche diese importiert, sichtbar. Die Spezifikation **nat** importiert keine Spezifikation, die Spezifikationen **nat1** und **natlist1** importieren beide **nat**, und die Spezifikation **natlist1** importiert zusätzlich **natlist**. Insgesamt ergibt sich folgende Struktur :




Im folgenden werden die vier Spezifikationen angegeben, und jede mit ein paar erläuternden Bemerkungen versehen.

Die Spezifikation **nat** exportiert die Sorte `NAT` und die Funktionen `0, suc`, welche beide freie Konstruktoren sind. Als Ordnung wird die lrpo angewählt, dabei ist die Präzedenz leer, und es werden keine Angaben bzgl. des Status der Funktionen gemacht.

```
specification nat
begin
  exports
    begin
      sorts NAT
      functions
        0        :           -> NAT
        suc      : NAT       -> NAT
    end

  properties
    begin
      constructors
```

```
        0    : f
        suc : f
      ordering lrpo
    end
end
```

Die Spezifikation **nat1** exportiert die Funktionen `+, *, -, <, >=` und die Sorte `NAT`. Durch den Import von `boolean` stehen die Sorte `bool` und die Standardfunktionen `and or` ... zur Verfügung. Eine Spezifikation **boolean** existiert nicht, dies erlaubt lediglich die Benutzung der vordefinierten Funktionen `=, and, or, ...` von RRL. Durch das in der Import-Anweisung enthaltene `nat^^` wird das Modul **nat** importiert, dabei nimmt das `^^` Bezug auf die File-Struktur, die erst später (siehe 3.3) erklärt wird. Wichtig ist an dieser Stelle eigentlich nur, daß das Modul **nat** importiert wird. Desweiteren werden die Variablen x,y,z,u der Sorte `NAT` deklariert und die üblichen Gleichungen für die eben genannten Funktionen angegeben. Alle Gleichungen sind mit einem sogenannten `tag` versehen, zum Beipiel die erste Gleichung mit dem `tag` N1a. Den Gleichungen schließen sich Ordnungsangaben und Beweisaufgaben an. Zu den Ordnungsangaben gehören die Angabe einer Präzedenz (nämlich $* \prec +, >= \prec <$) und eines Status (der Operator `*` hat den Status `left-to-right`). Die Beweisaufgaben sind mit einem Flag versehen, welches angibt, ob die Beweisaufgabe exportiert wird, hypothetisch oder bewiesen ist. In diesem Beispiel werden alle Beweisaufgaben als echte Beweisaufgaben exportiert.

```
specification nat1
begin
  exports
    begin
      functions
        _ + _  : NAT # NAT -> NAT
        _ * _  : NAT # NAT -> NAT
        _ - _  : NAT # NAT -> NAT
        _ >= _ : NAT # NAT -> bool
        _ < _  : NAT # NAT -> bool
    end

  imports
    boolean,
    nat^^

  variables
    x,y,z,u : -> NAT

  equations
    [N1a] x + 0 == x ;
    [N1b] x + suc(y) == suc(x + y) ;
```

```
    [N2a] 0 - x == 0 ;
    [N2b] x - 0 == x ;
    [N2c] suc(x) - suc(y) == x - y ;

    [N3a] x * 0 == 0 ;
    [N3b] x * suc(y) == x + (x * y) ;

    [N4a] x < 0 == false ;
    [N4b] 0 < suc(x) == true ;
    [N4c] suc(x) < suc(y) == x < y ;

    [N5a] x >= x == true ;
    [N5b] 0 >= suc(y) == false ;
    [N5c] suc(x) >= y == x >= y when (suc(x) = y) == false ;

  properties
  begin
    ordering
      precedence * +, >= <
      status * : r

    theorems
      [T1] x + y == y + x [e] ;
      [T2] (x * (y + z)) == ((x * y) + (x * z)) [e] ;
      [T4] x < suc(0) == x = 0 [e] ;
      [T5] (x + y) = 0 == and((x = 0) ,(y = 0)) [e] ;
      [T6] (x + y) = y == (x = 0) [e] ;
      [T7] (x + z) = (y + z) == (x = y) [e] ;
      [T8] (x * y) = 0 == or((x = 0) ,(y = 0)) [e] ;
      [T9] (x * y) = x == (y = suc(0)) when not(x = 0) == true [e]  ;
      [T10] (x * y) = x == or((y = suc(0)) ,(x = 0)) [e] ;
      [T11] (x * y) = suc(0) == and ((x = suc(0)) ,(y = suc(0))) [e] ;
      [T12] y < suc(y) == true [e] ;
      [T13] 0 >= u == u = 0 [e] ;
      [T14] suc(x) < y  == true
            when and((x < y) ,not(suc(x) = y)) == true [e] ;
      [T15] u >= z == not(u < z) [e] ;
      [T16] (u * y) < suc(y) == false
            when and(not(u = 0) ,and(not(u = suc(0)) ,not(y = 0))) == true [e] ;

end
```

Die Spezifikation **natlist** stimmt in ihrem Aufbau im wesentlichen mit der Spezifikation **nat** überein, besonders ist lediglich der Import von **nat** (wieder wird hier wie oben durch `nat^^` Bezug auf die File-Struktur genommen). Dieser Import ist notwendig, da die Sorte `NAT` aus dem Modul **nat** verwendet werden soll. Ferner ist es notwendig, daß die Sorte `NAT` in der Exportliste des Moduls **nat** vorkommt.

```
specification natlist
begin
  exports
    begin
      sorts NATLIST
      functions
        null    :                           -> NATLIST
        cons    : NAT # NATLIST             -> NATLIST
    end

  imports
    nat^^

  properties
    begin
      constructors
        null : f
        cons : f
      ordering lrpo
    end

end
```

Die Spezifikation **natlist1** stimmt in ihrem Aufbau mit der Spezifikation **nat1** überein. Das Besondere ist hier, daß die Beweisaufgabe T1 als zu beweisendes Theorem angegeben, aber nicht exportiert wird.

```
specification natlist1
begin
  exports
    begin
      functions
        append  : NATLIST # NATLIST        -> NATLIST
        delete  : NAT     # NATLIST        -> NATLIST
        member  : NAT     # NATLIST        -> bool
        perm    : NATLIST # NATLIST        -> bool
    end

  imports
    natlist^^
```

```
  variables
    x,y : -> NAT
    nl1,nl2 : -> NATLIST

  equations
    [NL1a]  append(null,nl1)        == nl1 ;
    [NL1b]  append(cons(x,nl1),nl2) == cons(x,append(nl1,nl2)) ;

    [NL2a] delete(x, null) == null;
    [NL2b] delete(x, cons(x, nl1)) == nl1 ;
    [NL2c] delete(x, cons(y, nl1)) ==  cons(y, delete(x, nl1))
             when (y = x) == false ;

    [NL3a] member(x, null) == false ;
    [NL3b] member(x, cons(x, nl1)) == true ;
    [NL3c] member(x, cons(y, nl1)) == member(x, nl1)
           when (x = y) == false ;

    [NL4a] perm(null, null) == true ;
    [NL4b] perm(null, cons(x, nl1)) == false ;
    [NL4c] perm(cons(x, nl1), null) == false ;
    [NL4d] perm(cons(x, nl1), nl2) ==
           and(member(x, nl2), perm(nl1 , delete(x, nl2))) ;

  properties
  begin
    ordering
      precedence perm member, perm delete
      status perm : l
    theorems
      [T1] delete(x, nl1) == nl1
           when not(member(x, nl1)) == true ;

  end

end
```

# Kapitel 3

# Aufgabenstellung

In diesem Kapitel soll die Aufgabenstellung präzisiert werden. Wie bereits erwähnt, sollten Möglichkeiten geschaffen werden, die Modularisierbarkeit von Beweisen über Gleichungsspezifikationen zu untersuchen. Im wesentlichen wurde dazu die Spezifikationssprache ASF derart erweitert, daß die Integration von Beweisaufgaben in die Spezifikation möglich wurde. Die Beweisaufgaben einer ASF-Spezifikation können durch eine Übersetzung dem Beweiser RRL zugänglich gemacht, die entsprechenden Beweisverläufe in RRL durch eine Rückübersetzung für eine Erweiterung der ursprünglichen ASF-Spezifikation herangezogen werden. Die Wirkungsweise des Systems ART (ASF RRL Translation) läßt sich im folgenden Diagramm veranschaulichen. Eine ASF-

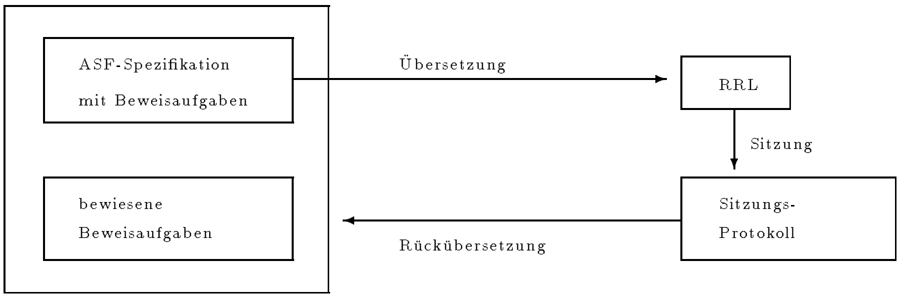



Abbildung 3.1:

Spezifikation, in welcher Beweisaufgaben verankert sind, wird nach RRL übersetzt, d.h. aus der Spezifikation werden Files erzeugt, welche in RRL ausgeführt werden können. ART unterstützt die Beweismethoden 'Cover-Set Induktion' und 'Induktionslose Induktion mit Test-Mengen' von RRL, d.h. es werden bei der Übersetzung zwei Files erzeugt. Wird ein solches File in RRL ausgeführt, werden vor allem die (echten) Beweisaufgaben behandelt (neben der Orientierung von Gleichungen und anderen durch die jeweilige Beweismethode bedingten Aktionen). Die gesamte RRL-Sitzung wird von RRL in einem File protokolliert, welches zur Ermittlung der in RRL enstandenen Ergebnisse herangezogen wird. Im wesentlichen wird hierbei der Verlauf der Beweise ermittelt (neben einer eventuellen Erweiterung der Präzedenz, Änderung des Status, ...).

Bei der Rückübersetzung wird eine neue ASF-Spezifikation erzeugt, welche aus der ursprünglichen Spezifikation und den ermittelten Ergebnissen besteht.
Dies alles soll an einem Beispiel verdeutlicht werden.

**Beispiel 0.0.1:** Im folgenden wird die ASF-Spezifikation `naturals` angegeben. In ihrem Export-Bereich befinden sich die Sorte `NAT` und die Funktionen `0, succ, +, *`. Die Variablen `x, y, z` werden als Variablen der Sorte `NAT` deklariert. Danach folgen die Standardgleichungen `N1, ..., N4` für die Funktionen `+, *`. Im Eigenschaften-Bereich der Spezifikation werden zuerst die Operatoren `0, succ` als freie Konstruktoren, die Ordnung `lrpo` (lexicographic recursive path ordering), und die Beweisaufgaben `K, A1, A2, D` angegeben. Die Beweisaufgaben sind echte Beweisaufgaben (d.h. sie werden bei der Übersetzung als zu beweisende Gleichungen behandelt) und werden aus der Spezifikation `naturals` exportiert (Flag e).

```
specification naturals
begin
  exports
    begin
      sorts NAT
      functions
        0         :            -> NAT
        succ      : NAT        -> NAT
        _ + _     : NAT # NAT -> NAT
        _ * _     : NAT # NAT -> NAT
    end
  variables
    x,y,z : -> NAT
  equations
    [N1]  x + 0 == x ;
    [N2]  x + succ(y) == succ(x + y) ;
    [N3]  x * 0 == 0 ;
    [N4]  x * succ(y) == x + (x * y) ;
  properties
    begin
      constructors
        0    : f
        succ : f
      ordering lrpo
      theorems
        [K]  x + y == y + x [e] ;
        [A1] x + (y + z) == (x + y) + z [e] ;
        [A2] x * (y * z) == (x * y) * z [e] ;
        [D]  x * (y + z) == (x * y) + (x * z) [e] ;
     end
end
```

Die Übersetzung dieser ASF-Spezifikation nach RRL führt zum Erzeugen zweier Files, eines dieser Files unterstützt die Beweismethode 'Cover-Set Induktion', das andere die Methode 'Induktionslose Induktion mit Test-Mengen'. Hier soll mit der 'Cover-Set Induktion' gearbeitet werden, das entsprechende File findet sich in Beispiel 3.1.2 (das zur 'Induktionslosen Induktion mit Test-Mengen' in Beispiel 3.1.1). Wird dieses File in RRL ausgeführt, wird von RRL ein Protokoll angelegt, welches aus zwei Files besteht : `log-file` und `spec-file`. Das `log-file` (es ist durch den RRL-Befehl `log` entstanden) spiegelt im wesentlichen die Eingaben in und Ausgaben von RRL wider. Das `spec-file` (es ist durch den RRL-Befehl `spec-write` entstanden) gibt die Signatur, Konstruktoren, äquivalente Operatoren, Präzedenz, Status, AC-Operatoren, C-Operatoren und das Regelsystem an (dies ist im wesentlichen der Zustand von RRL am Ende der Sitzung). Sowohl das `log-file` wie auch das `spec-file`, die beim Abarbeiten des zur Methode 'cover-set induction' gehörenden Files von RRL angelegt wurden, befinden sich in Beispiel 3.2.1. Bei der Rückübersetzung wird nun das Protokoll von RRL (also `log-file` und `spec-file`) dazu herangezogen, neue, in RRL entstandene Informationen zu ermitteln und die ursprüngliche ASF-Spezifikation um diese zu erweitern. In unserem Beispiel werden dabei folgende Informationen ermittelt:

- Präzedenz : $* > +$
  (wurde vom Benutzer eingegeben)
- Status : $*$ left-to-right
  (wurde vom Benutzer eingegeben)
- AC-Operator : + ([K] und [A1] konnten bewiesen werden)
- Theoreme : [A2] und [D]
- Lemmata :
  - `(0 + x) == x`
  - `(succ(y) + x) == succ((y + x))`
  - `(x + (y + z)) == (z + (x + y))`

Bei der Rückübersetzung wird nun eine ASF-Spezifikation erzeugt, welche über folgende Informationen verfügt :

```
specification naturals
begin
  exports
    begin
      sorts NAT
      functions
        0        :           -> NAT
        succ     : NAT       -> NAT
        _ + _    : NAT # NAT -> NAT
```

```
        _ * _     : NAT # NAT -> NAT
    end
  imports boolean
  variables
    x,y,z : -> NAT
  equations
    [N1]  x + 0 == x ;
    [N2]  x + succ(y) == succ(x + y) ;
    [N3]  x * 0 == 0 ;
    [N4]  x * succ(y) == x + (x * y) ;
  properties
    begin
      constructors
        0    : f
        succ : f
      ac-operators +
      ordering lrpo
        precedence * +
        status * : l
      theorems
        [K]  x + y == y + x [o, e] ;
          [1] (0 + x) == x [ o ] ;
          [2] (succ(y) + x) == succ((y + x)) [ o ] ;
        [A1] x + (y + z) == (x + y) + z [o, e] ;
          [3] (x + (y + z)) == (z + (x + y)) [ o ] ;
        [A2] x * (y * z) == (x * y) * z [o, e] ;
        [D]  x * (y + z) == (x * y) + (x * z) [o, e] ;
     end
end
```

Um Redundanz zu vermeiden, wurde eine File-Struktur vereinbart, in welcher lediglich ein File mit den neuen Informationen angelegt wird. Die Spezifikation ist dann über mehrere Files verteilt und ergibt sich als deren Vereinigung. Die Verwaltung dieser Spezifikation wird von ART unterstützt (siehe Kapitel 3.3). Wesentlich ist dabei, daß ART nur die neu gewonnenen Informationen in einem neuen File ablegt.

```
specification naturals
begin
  properties
  begin
    ac-operators +
    ordering
      precedence * +
      status * : l
    theorems
```

```
      [1] (x + (y + z)) == (z + (x + y)) [ o ] ;
      [2] (succ(y) + x) == succ((y + x)) [ o ] ;
      [3] (0 + x) == x [ o ] ;
      [4] (x * (y + z)) == ((x * y) + (x * z)) [ o, e ] ;
      [5] (x * (y * z)) == ((x * y) * z) [ o, e ] ;
      [6] (x + (y + z)) == ((x + y) + z) [ o, e ] ;
      [7] (x + y) == (y + x) [ o, e ] ;
  end
end
```

Insgesamt erlaubt ART dann den Zugriff auf die gesamte erweiterte ASF-Spezifikation.

## 3.1 Übersetzung von ASF nach RRL

RRL verlangt als Eingabe eine einzige flache Spezifikation. Aus diesem Grund muß bei der Übersetzung von ASF nach RRL zunächst die Modularisierung einer ASF-Spezifikation aufgehoben werden. Dies wird durch Auflösen der Import-Anweisungen erreicht. Bei der Transformation einer ASF-Spezifikation in eine flache Darstellung, können, bedingt durch die Möglichkeit in ASF Funktionen zu überladen, Namenskonflikte auftreten, welche ermittelt und aufgelöst werden müssen. Dies ist nur dann möglich, wenn die den Konflikt verursachenden Funktionen über unterschiedliche Definitionsbereiche verfügen. Das Auflösen der Namenskonflikte erfolgt durch Umbenennung der kollidierenden Funktionen. Nach der Umbenennung dieser Funktionen können dann entsprechend der Beweismethode RRL-Files erzeugt werden.
In ART werden zwei Beweismethoden von RRL unterstützt, welche unterschiedliches Vorgehen erforderlich machen. Die Methode *inductionless induction using test sets* verlangt ein mit dem Gleichungssystem äquivalentes kanonisches Termersetzungssystem und die vollständige Definiertheit aller Nicht-Konstruktoren auf allen Konstuktor-Grundtermen. Um diese Methode anwenden zu können muß daher das Gleichungs-System vervollständigt und die entprechenden Funktionen auf vollständige Definiertheit überprüft werden. Es ergibt sich folgender Aufbau für das zur Methode *inductionless induction using test sets* gehörende File, welches im folgenden (`kb-file`) genannt wird, da bei dieser Methode in RRL mit dem Befehl 'kb' (Knuth Bendix) die Gleichungsmenge vervollständigt wird:

| |
|---|
| Signatur |
| Gleichungen |
| Konstruktoren,Präzedenz,Status, C-Operatoren, AC-Operatoren |
| Vervollständigung |
| Überprüfung der vollständigen Definiertheit |
| Beweismethode auf inductionless induction using test sets setzen |
| Beweisaufgaben |

Um die zu vervollständigende Gleichungsmenge möglichst klein zu halten, werden lediglich die Gleichungen (also nicht die Theoreme und Hypothesen) der ASF-Spezifikation als Gleichungen in das `kb-file` aufgenommen.

Bei der *cover-set induction* hingegen muß anders vorgegegangen werden. In dieser Methode wird zwischen Definitionen und Eigenschaften unterschieden: Aus den Definitionen werden die cover-sets ermittelt, welche Induktive Inferenzregeln festlegen. Die Eigenschaften werden zur Simplifikation herangezogen. Definitionen wie auch Eigenschaften müssen hierbei zunächst zu Regeln orientiert werden. Alle Nicht-Konstruktor Funktionen müssen ebenso wie bei der Methode *inductionless induction using test sets* auf vollständige Definiertheit überprüft werden. Da bei dieser Methode die Definitionen und Eigenschaften in RRL mit dem Befehl 'mr' (makerule) orientiert werden, wird das zugehörige File im folgenden `mr-file` genannt. Es ergibt sich folgender Aufbau für das zur Methode *cover-set induction* gehörende File (`mr-file`):

| |
|---|
| Signatur |
| Definitionen |
| Konstruktoren,Präzedenz,Status, C-Operatoren, AC-Operatoren |
| Orientierung |
| Überprüfung der vollständigen Definiertheit |
| Beweismethode auf cover-set induction setzen |
| Ermittlung der cover-sets |
| Eigenschaften |
| Orientierung |
| Beweisaufgaben |

Dabei werden die Gleichungen der ASF-Spezifikation als Definitionen, die Theoreme und Hypothesen als Eigenschaften interpretiert. In den folgenden beiden Beispielen werden das `kb-file` und das `mr-file` der Spezifikation `naturals` aus Beispiel 3.0.1 angegeben. Beide sind zusätzlich mit Kommentaren (die mit ';;' beginnenden Zeilen) versehen, die in den Original-Files nicht vorhanden sind.

**Beispiel 1.0.1:** `kb-file` (inductionless induction using test sets) der Spezifikation naturals

```
;; Initialisierung
init

;; log-file wird angelegt
log
/usr/users/eschbach/art/specs/docu/naturals/naturals

;; Um Speicherplatz zu sparen wird der Undo-Modus ausgeschaltet.
option
undo
noundo

;; Eingabe der Signatur und der Gleichungen
ADD
[0: NAT]
[succ: NAT -> NAT]
[+: NAT, NAT -> NAT]
[*: NAT, NAT -> NAT]
(x + 0) == x
(x + succ(y)) == succ((x + y))
(x * 0) == 0
(x * succ(y)) == (x + (x * y))
]

;; Konstruktoren, ...
operator
constructor
0

operator
constructor
succ
yes

operator
order
l

;; Vervollstaendigung
kb
```

```
;; Ueberpruefen der vollstaendigen Definiertheit
suffice

;; Beweismethode
option
prove
s

;; Beweisaufgaben
prove
(x + y) == (y + x)

prove
(x + (y + z)) == ((x + y) + z)

prove
(x * (y * z)) == ((x * y) * z)

prove
(x * (y + z)) == ((x * y) + (x * z))

;; spec-file wird angelegt
write
spec-write
/usr/users/eschbach/art/specs/docu/naturals/naturals.spec
```

**Beispiel 1.0.2:** `mr-file` (cover-set induction) der Spezifikation naturals

```
;; Initialisierung
init

;; log-file wird angelegt
log
/usr/users/eschbach/art/specs/docu/naturals/naturals

;; Um Speicherplatz zu sparen wird der Undo-Modus ausgeschaltet.
option
undo
noundo

;; Eingabe der Signatur und der Definitionen
ADD
[0 : -> NAT]
[succ : NAT -> NAT]
```

```
[+ : NAT, NAT -> NAT]
[* : NAT, NAT -> NAT]
(x + 0) := x
(x + succ(y)) := succ((x + y))
(x * 0) := 0
(x * succ(y)) := (x + (x * y))
]

;; Konstruktoren, ...
operator
constructor
0

operator
constructor
succ
yes

operator
order
l

;; Orientierung der Definitionen
Makerule

;; Ueberpruefen der vollstaendigen Definiertheit
suffice

;; Beweismethode
option
prove
e

;; Ermitteln der cover-sets
cover

;; Eigenschaften (hier leer)
add
]

;; Orientierung der Eigenschaften
makerule

;; Beweisaufgaben
prove
```

```
(x + y) == (y + x)

prove
(x + (y + z)) == ((x + y) + z)

prove
(x * (y * z)) == ((x * y) * z)

prove
(x * (y + z)) == ((x * y) + (x * z))

;; spec-file wird angelegt
write
spec-write
/usr/users/eschbach/art/specs/docu/naturals/naturals.spec
```

## 3.2 Rückübersetzung von RRL nach ASF

Bei der Übersetzung von ASF nach RRL erzeugt ART aus einer ASF-Spezifikation zwei Files, die in RRL abgearbeitet werden können. Beim Abarbeiten eines dieser Files fertigt RRL ein Protokoll an, welches vor allem die Beweisverläufe der gestellten Beweisaufgaben beinhaltet. Das Protokoll besteht aus zwei Dateien, zum einen aus dem durch den Befehl

```
log
```

erzeugten `log-file` und zum anderen aus dem durch den Befehl

```
write
spec-write
```

erzeugten `spec-file`. In dem `log-file` werden alle Eingaben protokolliert, das `spec-file` repräsentiert den aktuellen Zustand (zum Zeitpunkt der Ausgabe) von RRL. Die Rückübersetzung von RRL nach ASF hat zur Aufgabe, aus diesem Protokoll neue Ergebnisse wie etwa den erfolgreichen Beweis einer Beweisaufgabe zu extrahieren und mit diesen Ergebnissen die ursprüngliche ASF-Spezifikation zu erweitern. Dazu werden im wesentlichen die Informationen aus dem `spec-file` und dem `log-file` mit denen der ASF-Spezifikation verglichen. Dabei werden zwei Differenzmengen ermittelt, nämlich $ASF \setminus RRL$, d.h. Informationen, die in der ASF-Spezifikation aber nicht im RRL-Protokoll enthalten sind, und $RRL \setminus ASF$, d.h. Informationen, die im RRL-Protokoll, aber nicht in der ASF-Spezifikation enthalten sind. In $ASF \setminus RRL$ sind beispielsweise die Gleichungen enthalten, die in RRL durch Simplifikation gelöscht wurden. Diese Differenz dient eigentlich nur zur Kontrolle für den Benutzer. Die eigentlich interessante Differenz ist $RRL \setminus ASF$, welche gerade alle neuen Informationen enthält; hiermit wird also die ursprüngliche ASF-Spezifikation erweitert. Dazu wird im wesentlichen nach den Regeln der von ART unterstützten File-Struktur ein File erzeugt, welches, um Redundanz zu vermeiden, nur die neuen Informationen enthält. ART erlaubt dann den Zugriff auf die gesamte erweiterte ASF-Spezifikation. Dies wird im nachfolgenden Unterkapitel File-Struktur präzisiert. Die Rückübersetzung von ASF nach RRL soll nun an einem Beispiel verdeutlicht werden. Dazu wird das dieses Kapitel begleitende Beispiel der Spezifikation `naturals` weitergeführt.

**Beispiel 2.0.1:** Die ASF-Spezifikation `naturals` aus Beispiel 3.0.1 wurde nach RRL übersetzt, d.h. es wurde das `kb-file` (siehe Beispiel 3.1.2) und das `mr-file` (siehe Beispiel 3.1.1) erzeugt. Das `kb-file` unterstützt die Beweismethode *inductionless induction using test-sets*, das `mr-file` die Methode *cover-set induction.* Hier soll mit der Cover-Set Induktion gearbeitet werden. Wird das `mr-file` der Spezifikation `naturals` in RRL ausgeführt, legt RRL ein Sitzungsprotokoll an, welches aus dem `log-file` und dem `spec-file` besteht. Das von RRL angelegte `log-file` hat folgendes Aussehen :

```
option
undo
noundo

add
[succ : NAT -> NAT ]
[+ : NAT, NAT -> NAT ]
[* : NAT, NAT -> NAT ]
[0 : -> NAT ]
(x + 0) == x
(x + succ(y)) == succ((x + y))
(x * 0) == 0
(x * succ(y)) == (x + (x * y))
]

operator
constructor
0

operator
constructor
succ
yes

operator
order
l

makerule
1
```

```
suffice

; Note: the system may be not canonical.

; Specification of '+' is completely defined.

; Specification of '*' is completely defined.

option
prove
e

cover

add
]

makerule

prove
(x + y) == (y + x)

; All subgoals are proved so
; (0 + x) == x
;  is an inductive theorem.

; All subgoals are proved so
; (succ(y) + x) == succ((y + x))
;  is an inductive theorem.

; All subgoals are proved so
; (x + y) == (y + x)
;  is an inductive theorem.

; Following equation
;    (x + y) == (y + x)  [user, 5]
;    is an inductive theorem in the current system.

prove
(x + (y + z)) == ((x + y) + z)

; All subgoals are proved so
; (x + (y + z)) == (z + (x + y))
;  is an inductive theorem.
```

```
; Following equation
;     (x + (y + z)) == (z + (x + y))  [user, 6]
;     is an inductive theorem in the current system.

prove
(x * (y * z)) == ((x * y) * z)

; All subgoals are proved so
; (x * (y + z)) == ((x * y) + (x * z))
;  is an inductive theorem.

; All subgoals are proved so
; (x * (y * z)) == ((x * y) * z)
;  is an inductive theorem.

; Following equation
;     (x * (y * z)) == ((x * y) * z)  [user, 7]
;     is an inductive theorem in the current system.

status
*
lr

prove
(x * (y + z)) == ((x * y) + (x * z))
; Yes, it is equational theorem.

write
spec-write
/usr/users/eschbach/art/specs/docu/naturals/naturals.spec
yes
```

Das `log-file` hat sehr viel Ähnlichkeit mit dem `mr-file`, der wesentliche Unterschied besteht in den mit ';' beginnenden Zeilen, die den Beweisverlauf der Beweisaufgaben skizzieren. Zu Beginn wird mit `option undo noundo` der Undo-Modus ausgeschaltet, um Speicherplatz zu sparen. In dem folgenden ADD-Block wird die Signatur und die definierenden Gleichungen angegeben, Danach folgt die Angabe der freien Konstruktoren `0` , `succ` , und die Auswahl der Ordnung 'lexicographic recursive path ordering'. Mit `makerule` werden die definierenden Gleichungen orientiert, die folgende 1 (dies ist eine Benutzereingabe) bedeutet, daß bei der Orientierung der Gleichungen die Präzedenz erweitert wurde. Der Benutzer hat den ersten der von RRL angebotenen Präzedenz-Vorschläge ausgewählt. Mit `suffice` überprüft RRL die vollständige Definiertheit der Nicht-Konstruktoren, also von `+` und `*`. Beide Operatoren sind auf allen Konstruktor-Grundtermen definiert. Mit `option prove e` wird die Beweismethode auf 'cover-set induction' gesetzt, das darauffolgende `cover` bewirkt das Ermitteln der Cover-Sets aus den definierenden Gleichungen. Hiernach folgt die Eingabe der Eigenschaften (hier leer) und deren Orientierung durch `makerule`. Stellvertretend für alle Beweisverläufe wird jetzt der Verlauf der ersten Beweisaufgabe `x + y == y + x` erläutert. Die entspechende Ausgabe ist nachträglich mit Zahlen versehen worden.

```
1  prove
2  (x + y) == (y + x)
3
4  ; All subgoals are proved so
5  ; (0 + x) == x
6  ;  is an inductive theorem.
7
8  ; All subgoals are proved so
9  ; (succ(y) + x) == succ((y + x))
10 ;  is an inductive theorem.
11
12 ; All subgoals are proved so
13 ; (x + y) == (y + x)
14 ;  is an inductive theorem.
15
16 ; Following equation
17 ;    (x + y) == (y + x)  [user, 5]
18 ;    is an inductive theorem in the current system.
```

Mit den ersten beiden Zeilen wird die Gleichung `x + y ==  y + x` RRL als Beweisaufgabe übergeben. RRL entscheidet sich, in dem Term `x + y` über die Variable x die Induktion zu führen. Die Zeilen 4,5,6 besagen, daß RRL den Induktionsanfang, die Zeilen 8,9,10, daß RRL den Induktionsschritt erfolgreich beweisen konnte. Damit sind alle Teilziele der eigentlichen Beweisaufgabe (Zeilen 12, 13, 14) und die eigentliche Beweisaufgabe erfolgreich bewiesen (Zeilen 16,17,18) worden. Am Ende eines Beweisverlaufs werden die beweisenen Theoreme (also eigentliches Theorem und entstandene Lemmata) orientiert. Hier spielt der oben beschriebene Fall eine Sonderrolle, da RRL den Operator `+` als C-Operator einträgt und somit Theorem und Lemmata als Gleichungen nicht mehr explizit auftauchen. Die anderen Beweisverläufe sind ähnlich. Nach dem erfolgreichen Beweis der Assoziativität von $*$, muß die entsprechende Gleichung orientiert werden. Dies erfordert eine Erweiterung des Status. Wie das File dokumentiert, hat sich der Benutzer entschieden, dem Operator $*$ den Status `left-to-right` zu geben. Die letzte Beweisaufgabe, nämlich die Distributivität, ist sogar ein Gleichheitstheorem, da diese als Hilfslemma beim Beweis der Assoziativität von $*$ angefallen ist.

Das von RRL erzeugte `spec-file` erklärt sich von selbst und hat folgendes Aussehen :

```
The arities of the operators are:
      [succ : NAT -> NAT ]
      [+ : NAT, NAT -> NAT ]
```

```
      [* : NAT, NAT -> NAT ]
      [0 : -> NAT ]

The system has the following constructors:
     Type 'NAT': { 0, succ }

There are no equivalent operators.

Precedence relation now is:
   * > +

Operators with status are:
   * with left-to-right status.

Associative & commutative operator set = { + }

(x + 0) ---> x
(x + succ(y)) ---> succ((x + y))
(x * 0) ---> 0
(x * succ(y)) ---> (x + (x * y))
(x * (y + z)) ---> ((x * y) + (x * z))
((x * y) * z) ---> (x * (y * z))
```

Das `spec-file` wird ganz am Schluß der RRL-Sitzung angelegt, stellt also im wesentlichen den Endzustand von RRL dar. Der Grund, warum zusätzlich zum `log-file` auch das `spec-file` bei der Rückübersetzung beachtet wird, liegt vor allem in den Präzedenzerweiterungen, die beim Orientieren der Gleichungen durch den RRL-Befehl `makerule` anfallen. Hier liefert RRL eine Liste von Präzedenz-Vorschlägen, die Auswahl eines dieser Vorschläge geschieht durch eine Zahl. Da nur diese Zahl im `log-file` erscheint, muß die korrekte Präzedenz aus dem `spec-file` entnommen werden. Wie oben schon angedeutet, werden von ART die Differenzen $ASF \setminus RRL$ und $RRL \setminus ASF$ ermittelt. Dabei ist in diesem Beispiel die Differenz $ASF \setminus RRL$ leer; es ist also in RRL nichts verloren gegangen. Die Differenz $RRL \setminus ASF$ besitzt folgende Informationen :

| | |
|---|---|
| Präzedenz | `* > +` |
| Status | `* left-to-right` |
| AC-Operator | `+` |
| induktive Theoreme | `(0 + x) == x` |
| | `(succ(y) + x) == succ((y + x))` |
| | `(x + y) == (y + x)` |
| | `(x + (y + z)) == (z + (x + y))` |
| | `(x * (y + z)) == ((x * y) + (x * z))` |
| | `(x * (y * z)) == ((x * y) * z)` |
| Gleichheits-Theorem | `(x * (y + z)) == ((x * y) + (x * z))` |

Diese in RRL entstandenen Informationen werden in ein File geschrieben, dessen Namensgebung durch die File-Struktur festgelegt ist, und folgendes Aussehen hat.

```
specification naturals
begin
  properties
  begin
    ac-operators +
    ordering
      precedence * +
      status * : l
    theorems
      [1] (x + (y + z)) == (z + (x + y)) [ o ] ;
      [2] (succ(y) + x) == succ((y + x)) [ o ] ;
      [3] (0 + x) == x [ o ] ;
      [4] (x * (y + z)) == ((x * y) + (x * z)) [ o, e ] ;
      [5] (x * (y * z)) == ((x * y) * z) [ o, e ] ;
      [6] (x + (y + z)) == ((x + y) + z) [ o, e ] ;
      [7] (x + y) == (y + x) [ o, e ] ;
  end
end
```

Die in RRL angefallenen Lemmata werden dabei mit dem Flag 'o' für 'ok' versehen[1]. Echte Beweisaufgaben behalten ihre Flags und bekommen zusätzlich das Flag 'o' für 'ok', falls ihr Beweis in RRL erfolgreich war [2]. Insgesamt hat man jetzt innerhalb der File-Struktur Zugriff auf die schon zu Beginn beschriebene erweiterte ASF-Spezifikation :

```
specification naturals
begin
  exports
    begin
      sorts NAT
      functions
        0        :             -> NAT
        succ     : NAT         -> NAT
        _ + _    : NAT # NAT -> NAT
        _ * _    : NAT # NAT -> NAT
    end
  imports boolean
  variables
    x,y,z : -> NAT
  equations
    [N1]  x + 0 == x ;
```

[1]zum Beispiel `[1] (x + (y + z)) == (z + (x + y)) [ o ]`

[2]zum Beispiel `[7] (x + y) == (y + x) [ o, e ]`

```
    [N2]  x + succ(y) == succ(x + y) ;
    [N3]  x * 0 == 0 ;
    [N4]  x * succ(y) == x + (x * y) ;
  properties
    begin
      constructors
        0    : f
        succ : f
      ac-operators +
      ordering lrpo
        precedence * +
        status * : l
      theorems
        [K]  x + y == y + x [o, e] ;
          [1] (0 + x) == x [ o ] ;
          [2] (succ(y) + x) == succ((y + x)) [ o ] ;
        [A1] x + (y + z) == (x + y) + z [o, e] ;
          [3] (x + (y + z)) == (z + (x + y)) [ o ] ;
        [A2] x * (y * z) == (x * y) * z [o, e] ;
        [D]  x * (y + z) == (x * y) + (x * z) [o, e] ;
     end
end
```

Hierbei wurde nachträglich, um besser zwischen Theoremen und Lemmata unterscheiden zu können, die Lemmata durchnummeriert und die Theoreme mit ihren ursprünglichen Tags (siehe Beispiel 1) versehen.

## 3.3 File-Struktur

Die dem System ART zugrundeliegende File-Struktur legt die Verzeichnis-Struktur und die Namensgebung der entsprechenden ASF-Files bzw. RRL-Files fest. Das oberste Verzeichnis innerhalb dieser Struktur beinhaltet die Verzeichnisse aller zu betrachtenden ASF-Spezifikationen.

**Beispiel 3.0.1:**
Sei *specs* das oberste Verzeichnis und seien die ASF-Spezifikationen *nat*, *natlist* und *arithmetic* gegeben. Es ergibt sich folgende Verzeichnis-Struktur

Im Verzeichnis *nat* befindet sich das ASF-File *nat.asf*, welches die ASF-Spezifikation *nat* beinhaltet. Analog befinden sich in den anderen Verzeichnissen die ASF-Files *natlist.asf* bzw. *arithmetic.asf*.

Sei im folgenden *specs* das oberste Verzeichnis und $spec_1$, ... ,$spec_n$ die ASF-Spezifikationen. Es ergibt sich die in der folgenden Abbildung dargestellte Verzeichnistruktur.

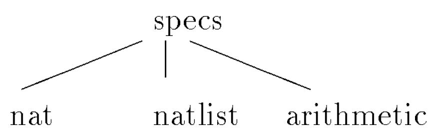



Abbildung 3.2:

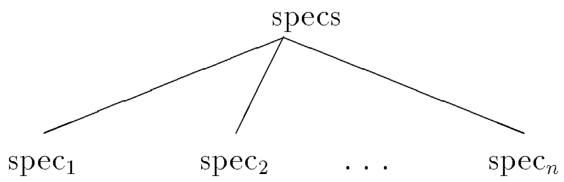



Abbildung 3.3:

Im Verzeichnis $spec_i$ befindet sich das zur ASF-Spezifikation $spec_i$ gehördende ASF-File $spec_i.asf$. Bei der Übersetzung dieses ASF-Files werden zwei RRL-Files (*mr-file*, *kb-file*) erzeugt und die Zwischendarstellung von dem ASF-File in einem File (*intern-file*) abgelegt. Das *mr-file* bekommt den Namen $spec_i.rrlmr$ ,das *kb-file* den Namen $spec_i.rrlkb$ und das File, welches die Zwischendarstellung enthält, den Namen $spec_i.intern$. Beim Abarbeiten eines der beiden RRL-Files wird dann von RRL ein *spec-file* und ein *log-file* erzeugt. Dabei erhält das *spec-file* den Namen $spec_i.spec$ und das *log-file* den Namen $spec_i.cmd$ . Insgesamt können sich also im Verzeichnis $spec_i$ folgende Files befinden :

- $spec_i.asf$
- $spec_i.rrlmr$
- $spec_i.rrlkb$
- $spec_i.spec$
- $spec_i.cmd$
- $spec_i.intern$

Zusätzlich bei der Rückübersetzung gewonnene Informationen gelangen innerhalb der File-Struktur in den sogenannten Implementations-Bereich oder in den sogenannten Eigenschaften-Bereich. Dabei bezeichnet der Implementations-Bereich alle im Verzeichniss $spec_i$ befindlichen Verzeichnisse

$$spec_i.impl\_j \text{ mit } 1 \leq \text{j} \leq \text{m}$$

und der Eigenschaften-Bereich die in $spec_i.impl\_j$ ($1 \leq$ j $\leq$ m) beginnende lineare Kette von Verzeichnissen

$$spec_i.impl\_j.prop\_k \text{ mit } 1 \leq \text{k} \leq \text{p}$$

Insgesamt ergibt sich die in der Abbildung 3.4 dargestellte Struktur.

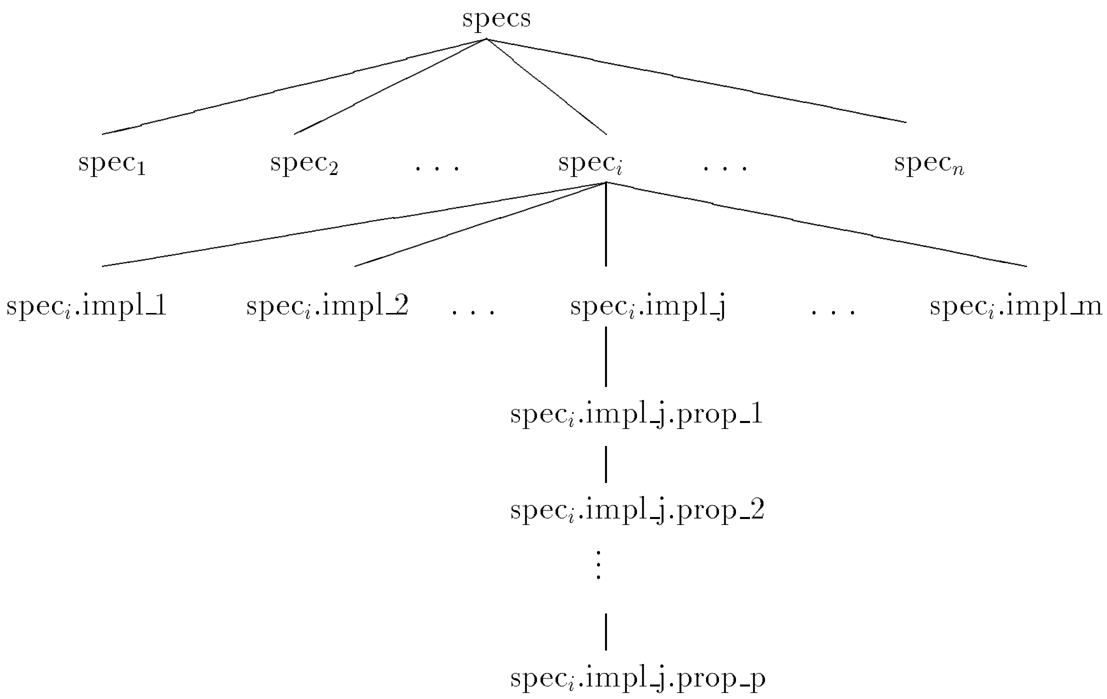



Abbildung 3.4: File-Struktur

In jedem dieser Verzeichnisse können sich, ebenso wie im Verzeichnis $spec_i$, folgende Files befinden

| | |
|---|---|
| <directory-name>.asf | das ASF-File |
| <directory-name>.rrlmr | das bei der Übersetzung erzeugte mr-file |
| <directory-name>.rrlkb | das bei der Übersetzung erzeugte kb-file |
| <directory-name>.intern | das bei der Übersetzung erzeugte intern-file |
| <directory-name>.spec | das von RRL erzeugte spec-file |
| <directory-name>.cmd | das von RRL erzeugte log-file |

Wobei <directory-name> der Name des entsprechenden Verzeichnisses ist.

Innerhalb der File-Struktur wird ein ASF-File im Spezifikations-Verzeichnis $spec_i$ eindeutig festgelegt durch

- Implementations-Nummer

- Eigenschaften-Tiefe

Dieses so festgelegt ASF-File ist für sich kein ASF-Modul; das zu diesem File gehörige ASF-Modul erhält man, in dem alle ASF-Files auf dem Pfad bis zu dem obersten Spezifikationsverzeichnis vereinigt werden, d.h. das zu $spec_i$*.impl_j.prop_k* gehörige ASF-Modul erhält man durch Vereinigen der ASF-Files

- $spec_i$*.asf*
- $spec_i$*.impl_j.asf*
- $spec_i$*.impl_j.prop_1.asf*
- 
- 
- 
- $spec_i$*.impl_j.prop_k.asf*

Eine besondere Rolle spielen mal wieder die Beweisaufgaben. Beweisaufgaben werden bei der Übersetzung von ASF nach RRL lediglich aus dem tiefsten ASF-File entnommen, d.h. nur aus $spec_i$*.impl_j.prop_k.asf*.
Damit wird verhindert, daß ehemalige Beweisaufgaben (die also bewiesen werden konnten) nicht wieder als Beweisaufgaben aufgenommen werden.

**Beispiel 3.0.2:**

Seien in *nat.impl_1* die Beweisaufgaben [T1], [T2] und [T3] enthalten, wobei [T1]

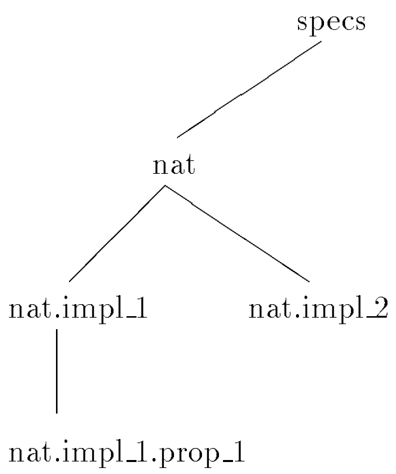



Abbildung 3.5:

und [T2] in RRL bewiesen werden konnten, [T3] jedoch nicht.
Bei der Rückübersetzung wird dann das Verzeichnis *nat.impl_1.prop_1* angelegt und

darin das ASF-File *nat.impl_1.prop_1.asf*, welches [T1], [T2] als bewiesene Theoreme und [T3] wieder als Beweisaufgabe enthält.
Das zu *nat.impl_1.prop_1.asf* gehörende ASF-Modul besteht aus der Vereinigung der ASF-Files

- *nat.asf*
- *nat.impl_1.asf*
- *nat.impl_1.prop_1.asf*

wobei die Beweisaufgaben lediglich aus *nat.impl_1.prop_1.asf* entnommen werden.

Die File-Struktur wird bei der Rückübersetzung aufgebaut. Dabei wird, falls die Rückübersetzung im Verzeichnis $spec_i$ durchgeführt wird (siehe Abbildung 3.4), eine neue Implementation (d.h. $spec_i.impl_{m+1}$) erzeugt. Hier wächst der Baum also in die Breite. Ansonsten (man befindet sich in einer Implementation) kann nur die tiefste Eigenschaft einer Implementation (z.B. $spec_i.impl_j.prop_p$) expandiert werden. Es wird in diesem Fall eine tiefere Eigenschaft erzeugt (z.B. $spec_i.impl_j.prop_{p+1}$). Die Intention, die hinter diesem Aufbau steckt, ist folgende :
Wird die ASF-Spezifikation aus dem Spezifikations-Verzeichnis $spec_i$ nach RRL übersetzt, können in RRL die Gleichungen oft unterschiedlich orientiert werden. Dies ist abhängig davon, wie Präzedenz oder Status vom Benutzer erweitert wird, falls RRL eine Gleichung nicht orientieren kann. Unterschiedliche Orientierung führt in den meisten Fällen auch zu unterschiedlichen Beweisverläufen der Beweisaufgaben, da ja anders reduziert werden kann. In den Implementations-Bereich können dann diese unterschiedlichen Orientierungen berücksichtigt werden, d.h. in den Implementationen können sich also unterschiedliche Präzedenz und Status befinden, die zu unterschiedlicher Orientierung der Gleichungen, und damit vielleicht zu unterschiedlichen Beweisverläufen der Beweisaufgaben führen. Der Eigenschaften-Bereich einer Implementation kann jetzt zusätzliche Beweisaufgaben enthalten, die in dieser Implementation möglicherweise zu einem erfolgreichen Beweisverlauf in RRL führen können, hier kann man also die spezielle Orientierung berücksichtigen.

# Kapitel 4

# ART

## 4.1 System-Files und Verzeichnisstruktur

Als erstes ein paar Bemerkungen zu der Verzeichnis-Struktur von ART, also wo Source Code, Dokumentation und Spezifikationen zu finden sind. Dem System ART liegt folgende Verzeichnis-Struktur zugrunde :

```
art
├── source
│   ├── compile-asf
│   └── save-rrl
├── documentation
└── specs
    ├── prime-factorization
    └── docu
```

Die Funktionalität von ART besteht im wesentlichen aus zwei Hauptfunktionen, nämlich `compile-asf` und `save-rrl`. Der Source Code dieser Funktionen befindet sich in `~/art/source/compile-asf/` bzw. `~/art/source/save-rrl/`.

Das Verzeichnis `~/art/documentation/` enthält diese Dokumentation (LaTex-Files).

Das Verzeichnis `~/art/specs/` enthält unter anderem das Verzeichnis `prime-factorization`, in welchem die gesamte, von ART erzeugte File-Struktur der in einem späteren Kapitel beschriebenen Primfaktorzerlegung vorhanden ist. Desweiteren befindet sich in `~/art/specs/` das Verzeichnis `docu`, in welchem sich das Spezifikationsverzeichnis der in dem Kapitel **Aufgabenstellung** beschriebenen ASF-Spezifikation `naturals` befindet.

In `~/art` befindet sich das File `load-art.lisp`, welches das gesamte System ART lädt. Dieses File muß also in Lisp geladen werden. In `~/art/source/compile-asf/` befindet sich das File `compile-asf.lisp`, welches alle für die Funktion `compile-asf` notwendigen Files lädt. In `~/art/source/save-rrl/` befindet sich das File `save-rrl.lisp`, welches alle für die Funktion `save-rrl` notwendigen Files lädt.

## 4.2 Implementation

### 4.2.1 Übersetzung von ASF nach RRL

Um eine ASF-Spezifikation nach RRL zu übersetzen, werden als erstes die ASF-Files der ASF-Spezifikation geparst und in eine interne Darstellung in LISP gebracht. Dabei wird die Modularisierung der ASF-Spezifikation aufgelöst. Diese interne Darstellung wird auf Namenskonflikte untersucht, und im Falle von Konflikten werden diese durch Umbenennung der kollidierenden Funktionen aufgelöst. Danach wird die interne Darstellung der ASF-Spezifikation semantischen Überprüfungen unterzogen (z.B. wird überprüft, ob jeder Term wohlgeformt ist). Nun kann die interne Darstellung zur Erzeugung der RRL-Files herangezogen werden. Hierbei wird für die Beweismethoden 'cover-set induction' und 'inductionless induction using test-sets' jeweils ein File erzeugt. Die Funktion, die die Übersetzung von ASF nach RRL verwirklicht, heißt `compile-asf`. Als Argument kann man

- einen Pfadnamen
- einen String

angeben. Ist das Argument ein Pfadname, wird lediglich das durch den Pfadnamen festgelegte asf-file übersetzt. Ist das Argument jedoch ein String, wird auf die von ART unterstützte File-Struktur Bezug genommen. Hierbei muß die Variable `*default-baum-dir*` auf das oberste Spezifikationsverzeichnis gesetzt werden. Dann kann über Spezifikations-Name, Implementations-Nummer und Eigenschaften-Tiefe die ASF-Spezifikation festgelegt werden, d.h. durch Angabe von `<name>^[<number>]^[<number>]`.

**Beispiel 2.1.1:** Die in 4.1 angegebene Verzeichnisstruktur enthält das Verzeichnis `art/specs/docu/`. In diesem Verzeichnis ist das Spezifikationsverzeichnis der ASF-Spezifikation `naturals` aus Beispiel 3.1 enthalten. Möchte man diese Spezifikation nach RRL übersetzen, muß `*default-baum-dir*` auf das Verzeichnis `art/specs/docu/` gesetzt werden. Durch
`(compile-asf "naturals")` wird jetzt die ASF-Spezifikation `naturals` nach RRL übersetzt.
Durch
`(compile-asf "naturals^1")` wird die erste Implementation der ASF-Spezifikation `naturals` (d.h. die Vereinigung der asf-files `naturals.asf` und `naturals.impl_1.asf`)

nach RRL übersetzt.
Durch
`(compile-asf "naturals^1^1")` wird die erste Eigenschaft der ersten Implementation der ASF-Spezifikation `naturals` (d.h. die Vereinigung der asf-files `naturals.asf` , `naturals.impl_1.asf` und `naturals.impl_1.prop_1.asf`) nach RRL übersetzt.

Im Anhang findet sich eine genauere Beschreibung von

- Variablen
- interne Darstellung
- Parser
- Auflösen von Namenskonflikten
- semantische Überprüfungen
- Erzeugen von RRL-Files

### 4.2.2 Rückübersetzung von RRL nach ASF

Bei der Rückübersetzung von RRL nach ASF müssen die Informationen des RRL-Protokolls (also `spec-file` und `log-file`) mit denen der ASF-Spezifikation verglichen werden. Hierzu wird als erstes das `spec-file` und das `log-file` der entsprechenden RRL-Sitzung geparst und deren Informationen in einer internen Darstellung abgelegt. Dasselbe geschieht mit den Informationen der ASF-Spezifikation . Danach werden die RRL-Informationen (die Informationen aus `log-file` und `spec-file`) und die ASF-Informationen (Informationen aus der ASF-Spezifikation) auf Konsistenz geprüft (z.B. wird sowohl die ASF-Präzedenz wie auch die RRL-Präzedenz auf Azyklidität überprüft). Nun werden die Differenzen $ASF \setminus RRL$ und $RRL \setminus ASF$ ermittelt. Mit der Differenz $RRL \setminus ASF$ wird die ursprüngliche ASF-Spezifikation erweitert. Zuvor muß jedoch die Umbenennung von Funktionen, die bei der Übersetzung von ASF nach RRL angefallen sind, rückgängig gemacht werden. Bei der Erweiterung der ASF-Spezifikation wird nach den Regeln der File-Struktur ein entsprechendes Verzeichnis angelegt, in welchem dann die Differenz $RRL \setminus ASF$ gemäß ASF-Syntax abgelegt wird. Hier wird entweder eine weitere Implementation oder eine tiefere Eigenschaft erzeugt. Die Funktion `save-rrl` führt die Rückübersetzung von RRL nach ASF durch. Ihr Argument stimmt mit dem der Funktion `compile-asf` überein.

**Beispiel 2.2.2:**
Wie im vorherigen Beispiel (4.2.1.1) sei das oberste Spezifikationsverzeichnis `art/specs/docu/`.
Der Aufruf
`(save-rrl "naturals")`

führt dazu, daß die Informationen aus dem `spec-file`
`art/specs/docu/naturals/naturals.spec`
und aus dem `log-file`
`art/specs/docu/naturals/naturals.cmd`
mit denen der ASF-Spezifikation
`art/specs/docu/naturals/naturals.asf`
verglichen wird. Die Differenz wird dann in einer neuen Implementation abgelegt.
Ist etwa noch keine Implementation vorhanden, wird das Verzeichnis
`art/specs/docu/naturals/naturals.impl_1/`
angelegt, und die Differenz in dem File
`art/specs/docu/naturals/naturals.impl_1/naturals.impl_1.asf`
abgelegt.

Durch den Aufruf
`(save-rrl "naturals^1")`
werden die Informationen aus dem `spec-file`
`art/specs/docu/naturals/naturals.impl_1/naturals.impl_1.spec`
und aus dem `log-file`
`art/specs/docu/naturals/naturals.impl_1/naturals.impl_1.cmd`
mit denen der ersten Implementation der ASF-Spezifikation `naturals` (also die Vereinigung der asf-files `art/specs/docu/naturals/naturals.asf` und
`art/specs/docu/naturals/naturals.impl_1/naturals.impl_1.asf`) verglichen.
Die Erweiterung führt zu einer 'tieferen' Eigenschaft, hier wird das Verzeichnis `art/specs/docu/naturals/naturals.impl_1/naturals.impl_1.prop_1/` angelegt und die Differenz RRL/ASF in dem File
`art/specs/docu/naturals/naturals.impl_1/naturals.impl_1.prop_1/naturals.impl_1.prop_1.asf` abgelegt.

Im Anhang findet man eine genauere Beschreibung von

- Variablen
- Datenstrukturen
- Parser für das `spec-file`
- Parser für das `log-file`
- Parser für das `asf-file`
- Überprüfen der Konsistenz
- Ermitteln der Unterschiede
- Rückumbenennung
- Ausgabe

### 4.2.3 Weitere Funktionen

Folgende Funktionen stehen dem Benutzer in ART zur Verfügung

- unter X
  - xrrl
    startet ein xterm und ruft darin RRL auf
  - xvi
    startet ein xterm und ruft darin den Editor vi auf
- in Lisp
  - rrl
    ruft in Lisp RRL auf
  - vi
    ruft in Lisp den Editor vi auf

Dabei haben alle Funktionen zwei Parameter, wobei der zweite optional ist. Der erste stimmt mit dem Parameter von *compile-asf* überein. Möchte man in RRL das makerule-file ablaufen lassen oder sich im VI die ASF-Spezifikation anschauen , kann auf den zweiten Parameter verzichtet werden. In allen anderen Fällen wird das entsprechende Suffix zum zweiten Parameter. Dies soll lediglich an einem Beispiel deutlich gemacht werden.

**Beispiel 2.3.3:**

| | |
|---|---|
| `(xrrl "nat^1")` | das File `nat.impl_1.rrlmr` wird ausgeführt. |
| `(xrrl "nat^1" "rrlmr")` | das File `nat.impl_1.rrlmr` wird ausgeführt. |
| `(xrrl "nat^1" "rrlkb")` | das File `nat.impl_1.rrlkb` wird ausgeführt. |
| `(xvi "nat^1")` | das File `nat.impl_1.asf` wird angezeigt. |
| `(xvi "nat^1" "asf")` | das File `nat.impl_1.asf` wird angezeigt. |
| `(xvi "nat^1" "rrlmr")` | das File `nat.impl_1.rrlmr` wird angezeigt. |

## 4.3 Bedienung

Möchte man konkret mit ART arbeiten, muß als erstes Lisp (Lucid Common Lisp) gestartet werden. Ist dies geschehen, kann durch den Lisp-Ausdruck

```
(load "~/art/load-art")
```

das System ART geladen werden. Das package 'user (in diesem package befindet man sich nach dem Laden) enthält nun die drei wichtigen Funktionen :

- compile-asf
  Übersetzung von ASF nach RRL

- xrrl
  Aufruf von RRL

- save-rrl
  Rückübersetzung von RRL nach ASF

Desweiteren ist die Variable `*default-baum-dir*` auf ein oberstes Spezifikationsverzeichnis gesetzt, und zwar per Default auf das Verzeichnis `~/art/specs/docu/`. Der Benutzer muß als erstes gegebenenfalls die Variable `*default-baum-dir*` auf sein oberstes Spezifikationsverzeichnis setzen. Hiernach kann mit den oben genannten Funktionen gearbeitet werden. All dies wird jetzt an einem ausführlichen Beispiel verdeutlicht.

**Beispiel 3.0.1:** Als erstes wird auf Unix-Ebene ein 'oberstes' Spezifikations-Verzeichnis mit einem Spezifikations-Verzeichnis `naturals` und zugehörigem asf-file `naturals.asf` erzeugt. Zu diesem Zweck wird in dem Verzeichnis `~/art/specs/` ein Verzeichnis angelegt, etwa das Verzeichnis `test`, falls dieses noch nicht vorhanden ist.

```
> mkdir ~/art/specs/test
```

In diesem Verzeichnis wird nun das Spezifkations-Verzeichnis `naturals` erzeugt und in dieses das ASF-File `~/art/specs/docu/naturals/naturals.asf` kopiert.

```
> mkdir ~/art/specs/test/naturals
> cp ~/art/specs/docu/naturals/naturals.asf
     ~/art/specs/test/naturals/naturals.asf
```

Jetzt kann mit ART gearbeitet werden. Dazu muß als erstes Lisp gestartet und dann mit

```
(load "~/art/load-art")
```

ART geladen werden.

Als erstes soll die **ASF**-Spezfikation `naturals` aus dem Verzeichnis `~/art/specs/test/naturals/` **nach RRL übersetzt** werden. Dazu muß die Variable `*default-baum-dir*` auf das Verzeichnis `~/art/specs/test/` gesetzt werden.

```
(setq *default-baum-dir* '("~" "art" "specs" "test"))
```

Danach kann die Spezfikation `naturals` von ASF nach RRL übersetzt werden. Dies geschieht durch

```
(compile-asf "naturals")
```

Möchte man das zur Methode 'cover-set induction gehörende File (mr-file) **in RRL ausführen**, geschieht dies durch

```
(xrrl "naturals")
```

Beim Ausführen dieses Files verlangt RRL einige Benutzer-Eingaben (etwa eine Präzedenz-Erweiterung). Ist von RRL das Protokoll angelegt, kann die **Rückübersetzung von RRL nach ASF** vorgenommen werden. Dies geschieht durch

```
(save-rrl "naturals")
```

Nachdem sich der Benutzer Unterschiede und Theoreme ansehen konnte [1], wird hier die erste Implementation der Spezifikation `naturals` erzeugt, d.h. es wird das Verzeichnis `~/art/specs/test/naturals/naturals.impl_1/` angelegt, und die ermittelten neuen Informationen in dem File
`~/art/specs/test/naturals/naturals.impl_1/naturals.impl_1.asf` abgelegt.

[1] ART gibt dem Benutzer die Möglichkeit, sich diese Informationen anzuschauen.

# Kapitel 5

# Beispiel : Primfaktorzerlegung

## 5.1 Einführung

In diesem Beispiel soll ein mit RRL geführter Beweis, welcher die Eindeutigkeit der Primfaktorzerlegung zur Hauptaussage hat, modularisiert werden. Zu diesem Zweck wird als erstes der Originalbeweis (in Form eines Kommando-Files, welches in RRL abgearbeitet werden kann) vorgestellt. Hier wird nicht nur der Aufbau des Kommando-Files, sondern auch die eigentliche Idee des Beweises beschrieben. Danach wird eine mögliche Modularisierung in ART angegeben und auf Probleme hingewiesen, die bei dieser Modularisierung entstehen. Gefahren sind hierbei als Schwierigkeiten mit RRL, genauer bei den Beweisen mit RRL, zu verstehen, die diese vorgeschlagene Modularisierung mit sich bringen. Eine typische Gefahr ist etwa eine Änderung (in Bezug auf den Originalbeweis) der Reihenfolge, in welcher die Beweisaufgaben in RRL bewiesen werden. Eine solche Änderung, die sich über die Modularisierung und die Übersetzung von ASF nach RRL einschleichen kann, kann das Scheitern von Beweisen nach sich ziehen. Im Anschluß daran wird konkret die Modularisierung in ART, also die entsprechenden ASF-Files der Spezifikation, angegeben und die zugrundeliegende Vorgehensweise in ART beschrieben.

## 5.2 Originalbeweis

Im folgenden wird ein Kommando-File vorgestellt, welches in RRL mit dem Befehl 'auto' abgearbeitet werden kann. Die Grundidee dieses Beweises ist, die Primfaktoren einer natürlichen Zahl ($> 1$) auszudividieren und diese in einer Liste abzulegen. Die Eindeutigkeit ist gezeigt, wenn je zwei Listen, die nur Primzahlen enthalten (`primelist`) und die gleiche Zahl repräsentieren (`timelist`), bis auf Permutation gleich sind (`perm`).

```
perm(x, y) == true if (primelist(x) and (primelist(y) and
                       (timelist(x) = timelist(y))))
```

| |
|---|
| Initialisierung |
| Beweismethode wird auf 'cover-set induction' gesetzt. |
| Angabe, welcher Bereich der Gleichung (linke Seite, rechte Seite, Bedingungen) reduziert wird. Hier wird nur die linke Seite einer Gleichung reduziert. |
| Protokoll (log-file) wird angelegt. |
| Angabe von Signatur, Definitionen und Eigenschaften |
| Angabe der Ordnung |
| Undo-Modus wird ausgeschaltet, um Speicherplatz zu sparen. |
| Angabe von Konstruktoren |
| Angabe des Status |
| Angabe der Präzedenz |
| Orientierung der Definitionen und Eigenschaften |
| Beweisaufgaben |

Tabelle 5.1: Aufbau des Kommando-Files

In dem nachfolgenden Kommando-File wird mithilfe der freien Konstruktoren `0, suc` die Menge der natürlichen Zahlen definiert. Danach werden die Funktionen `+, *, <, -, div, rem, divides` angegeben. Mit den freien Konstruktoren `nl, cons` können Listen erzeugt werden, mit `append, delete` stehen übliche Listen-Funktionen zur Verfügung. Genauere Betrachtung verdienen die Funktionen `prime, primelist, primefac`. Die Funktion `prime` entscheidet, ob eine Zahl eine Primzahl ist oder nicht, die Funktion `primelist` prüft, ob eine Liste nur Primzahlen enthält, und die Funktion `primefac` führt die eigentliche Primfaktorzerlegung durch, dabei werden alle Primfakoren in einer Liste abgelegt.

Das im folgenden angegebene Kommando-File ist zusätzlich mit Kommentaren (die mit ';;' beginnenden Zeilen) versehen worden. In diesen Kommentaren spiegelt sich der in Tabelle 5.1 angegebene Aufbau des Kommando-Files wider.

```
;; Initialisierung
init

;; Beweismethode
option prove e

;; Nur die linke Seite einer Gleichung wird reduziert.
option reduce 0

;; Protokoll
log
~/specs/primes/primes-mr-log

;; Signatur, Definitionen, Eigenschaften
add
[ 0 : num]
```

```
[ suc : num -> num]
[ + : num, num -> num]
[ * : num, num -> num]
1 == suc(0)
2 == suc(suc(0))
x + 0 := x
x + suc(y) := suc(x + y)
x * 0 := 0
x * suc(y) := x + x * y
[< : num, num -> bool]
0 < suc(x) := true
x < 0 := false
suc(x) < suc(y) := x < y
[sub1 : num -> num]
sub1(0) := 0
sub1(suc(x)) := x
[- : num, num -> num]
0 - x := 0
x - 0 := x
suc(x) - suc(y) := x - y
[div : num, num -> num]
div(x, 0) := 0
div(x, y) := 0 if x < y
div(y + x, y) := suc(div(x, y)) if not(0 = y)
[rem : num, num -> num]
rem(x, 0) := x
rem(x, y) := x if x < y
rem(y + x, y) := rem(x, y)
[divides : num, num -> bool]
divides(x, y) := rem(y,x) = 0
[nl : list]
[cons : univ, list -> list]
[append : list, list -> list]
append(nl, y) := y
append(cons(x, y), z) := cons(x, append(y, z))
[delete : univ, list -> list]
delete(x, nl) := nl
delete(x, cons(x, y)) := y
delete(x, cons(y, z)) := cons(y, delete(x, z)) if not(x = y)
[prime1 : num, num -> bool]
prime1(x, 0) := false
prime1(x, suc(0)) := true
prime1(x, suc(y)) := not(divides(suc(y), x)) and prime1(x,y) if not(y = 0)
[prime : num -> bool]
prime(0) := false
```

```
prime(suc(x)) := prime1(suc(x), x)
[primelist : list -> bool]
primelist(nl) := true
primelist(cons(x, y)) := prime(x) and primelist(y)
[timelist : list -> num]
timelist(nl) := suc(0)
timelist(cons(x, y)) := x * timelist(y)
[primefac : num -> list]
primefac(0) := nl
primefac(suc(0)) := nl
primefac(x * y) := append(primefac(x), primefac(y)) if not(x = 0) and not(y = 0)
[member : univ, list -> bool]
member(x, nl) := false
member(x, cons(x, z)) := true
member(x, cons(y, z)) := member(x, z) if not(x = y)
[perm : list, list -> bool]
perm(nl, nl) := true
perm(nl, cons(x, y)) := false
perm(cons(x, y), nl) := false
perm(cons(x, y), z) := member(x, z) and perm(y, delete(x, z))
[>= : num, num -> bool]
x >= x := true
0 >= suc(y) := false
suc(x) >= y := x >= y if not(suc(x) = y)
[gcd : num, num -> num]
gcd(x, 0) := x
gcd(0, y) := y
gcd(x + y, y) := gcd(x, y)
gcd(x, x + y) := gcd(x, y)
rem(y, y) == 0
rem(suc((y + y1)), y) ==  rem(suc(y1), y)
div(y,y) == suc(0) if not(0 = y)
]

;; Ordnung
option ordering l

;; Undo-Modus aus
option undo noundo

;; Konstruktoren
operator
constructor
0 suc nl cons
y
```

```
y

;; Status
operator
status
perm
lr
operator
status
*
lr
operator
status
<
lr

;; Praezedenz
operator
precedence
* +
operator
precedence
divides rem
operator
precedence
prime1 rem
operator
precedence
primelist prime
operator
precedence
timelist *
operator
precedence
primefac append
operator
precedence
perm member
operator
precedence
perm delete
operator
precedence
prime prime1
operator
```

```
precedence
div +
operator
precedence
div *
operator
precedence
timelist div
operator
precedence
>= <
operator
precedence
gcd *

;; Orientierung
makerule

;; Beweisaufgaben
prove
div(0, y) == 0

prove
rem(0, y) == 0

prove
x+y == y+x

prove
y * div(y+x,y) == y * suc(div(x, y))

prove
(rem(x, y) + (y * div(x, y))) == x

prove
(y * div(x, y)) == x if divides(y, x)

prove
(x * (y + z)) == ((x * y) + (x * z))

prove
(x * y) * z == x * (y * z)

prove
x * y == y * x
```

```
prove
0 < x == not(x = 0)

prove
x < suc(0) == x = 0

prove
(x + y) = 0 == (x = 0) and (y = 0)

prove
(x + y) = y == (x = 0)

prove
(x + z) = (y + z) == (x = y)

prove
(x * y) = 0 == (x = 0) or (y = 0)

prove
(x * y) = x == (y = suc(0)) if not(x = 0)

prove
(x * y) = x == (y = suc(0)) or (x = 0)

prove
(x * y) = suc(0) == (x = suc(0)) and (y = suc(0))

prove
div((x * y), x) == y if not((x = 0))

prove
rem((y * x), x) == 0

prove
rem((y * z), x) == 0 if (rem(z, x) = 0) and not(x = 0)

prove
rem(x+y,z) = rem(x, z) if rem(y, z) = 0

prove
(div(x, y) < x) if (not(x = 0)) and (not(y = 0)) and (not(y = suc(0)))

prove
divides(x, timelist(y)) == true if member(x, y)
```

```
prove
delete(x, y) == y if not member(x, y)

prove
primelist(delete(x, y)) if primelist(y)

prove
y < suc(y)

prove
(x = suc(0)) == false if prime(x)

prove
(x = 0) == false if prime(x)

prove
div(z + y, x) == div(z, x) + div(y, x) if
  divides(x, z) and not(x = 0)

prove
div(z * y, x) == y * div(z, x)
  if divides(x, z) and not(x = 0)

prove
timelist(delete(x, y)) == div(timelist(y), x) if
  (not((x = 0)) and  member(x, y))

prove
timelist(x) = 0 == false if primelist(x)

prove
timelist(append(z, z1)) == (timelist(z) * timelist(z1)) if
  (not(timelist(z) = 0) and not(timelist(z1) = 0))

prove
timelist(primefac(x)) == x if not(x = 0)

prove
primelist(primefac(x)) if not(x = 0)

prove
0 >= u == u = 0

prove
```

```
rem(suc(0), x) == suc(0) if not(x = suc(0))

prove
prime1(w*z, u) == false if not(z = suc(0)) and not(z = 0) and
  (u >= z) and not(u = suc(0))

prove
suc(x) < y if (x < y) and not(suc(x) = y)

prove
u >= z == not(u < z)

prove
(u * y) < suc(y) == false if not(u = 0) and not(u = suc(0)) and not(y = 0)

prove
sub1(x) < y == x < suc(y) if not(x = 0)

prove
prime1(x, sub1(x)) == false if not(z = suc(0)) and
    not(z = x) and not(x = 0) and
    not(x = suc(0)) and divides(z, x)

prove
rem(x, y) = 0 == false if prime(x) and not(y = suc(0)) and not(x = y)

prove
prime(y) == false if (rem(x, y) = 0) and prime(x) and not(x = y)

prove
gcd(x, y) == gcd(y, x)

prove
gcd((x * z), (y * z)) == (z * gcd(x, y))

prove
gcd(x*y, z) = y == false if (rem(z, x) = 0) and not(rem(y, x) = 0)

prove
rem(y * z, x) = 0 == false if (gcd(x*y, y*z) = y) and
  not(y = 0) and not(rem(y, x) = 0)

prove
gcd(x, suc(0)) == suc(0) if not((x = 0))
```

```
prove
gcd(x, y) == suc(0) if
 (rem(x, gcd(x, y)) = 0) and
 not(x = 0) and not(x = suc(0)) and
 prime1(x, sub1(x)) and
 not(gcd(x, y) = x)

prove
rem(x, gcd(x, y)) == 0

prove
gcd(x, y) = x == false if not(rem(y, x) = 0)

prove
prime1(x, sub1(x)) == false if not (rem(y, x) = 0) and
  not(gcd(x,y) = suc(0))

prove
rem(y * z, x) = 0 == false if prime(x) and
  not(divides(x, y)) and not(divides(x, z))

prove
member(x, y) if prime(x) and primelist(y) and divides(x, timelist(y))

prove
perm(x, y) == true if (primelist(x) and (primelist(y) and
  (timelist(x) = timelist(y))))
```

## 5.3 Modularisierung

In diesem Kapitel wird eine Modularisierung des zuvor beschriebenen Originalbeweises vorgestellt. In diesem Beweis werden die natürlichen Zahlen mit den freien Konstruktoren `0, suc`, die Listen über natürliche Zahlen mit den freien Konstruktoren `null, cons` spezifiziert. Auf den natürlichen Zahlen werden die Funktionen `+, *, -, <, >=` definiert und durch die Funktionen `div, rem, divides, gcd, sub1` eine Arithmetik festgelegt. Auf den Listen über natürliche Zahlen werden die üblichen Funktionen `append, delete, member, perm` (entscheidet, ob zwei Listen bis auf Permutation gleich sind) definiert. Für den Bereich Primzahlen sind die Funktionen `prime` (entscheidet, ob eine Zahl eine Primzahl ist), `primelist` (entscheidet, ob eine Liste nur aus Primzahlen besteht), `timelist` (multipliziert alle Zahlen aus einer Liste) und `primefac` (legt die Primfaktoren einer Zahl in einer Liste ab), zuständig.

Es ergibt sich folgende naheliegende Modularisierung :

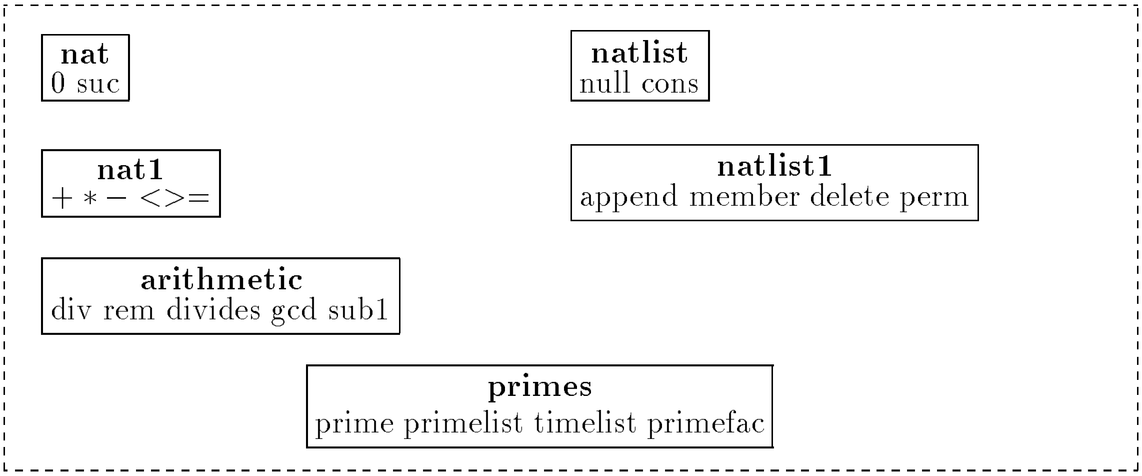



Abbildung 5.1: Modularisierung des Originalbeweises

In dieser Modularisierung enthält also das Modul **nat** die Funktionen `0` und `suc`, das Modul **natlist** die Funktionen `null` und `cons`, usw..

Desweiteren ergibt sich in natürlicher Weise folgende Import-Abhängigkeit :

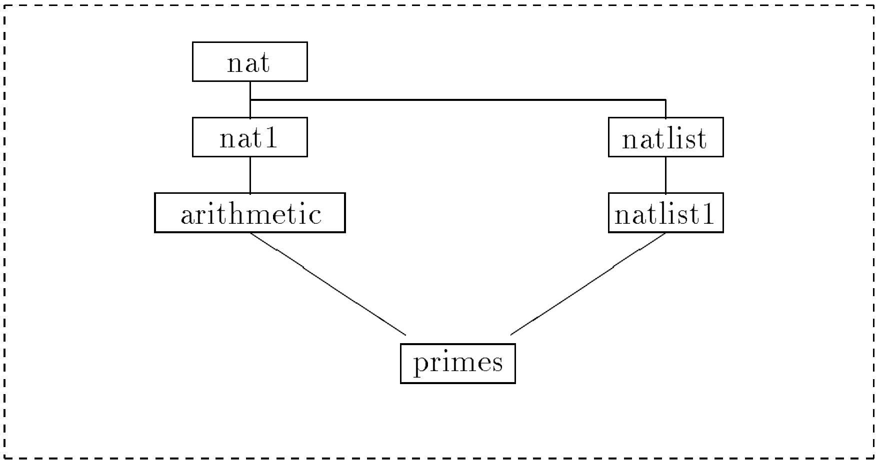



Abbildung 5.2: Import-Abhängigkeit der Module

D.h. das Modul **nat** wird von den Modulen **natlist** und **nat1** importiert, das Modul **nat1** von dem Modul **arithmetic**, usw.

Insgesamt stehen 58 Beweisaufgaben an, welche auf die Module verteilt werden müssen. Dabei ist folgende Strategie verfolgt worden: Die Beweisaufgaben wandern in das Modul, in welchem die Signatur gerade groß genug ist. Dies ist bis auf einige Ausnahmen unproblematisch, d.h. einige Beweisaufgaben mußten in 'größere' Module geschoben werden, um den erfolgreichen Beweis der Hauptaussage, d.h. der Eindeutigkeit der

Primfaktorzerlegung, zu garantieren.

**Beachte** : Durch diese Aufteilung der Beweisaufgaben auf die Module, wird die Reihenfolge der Beweisaufgaben in Bezug auf den Originalbeweis geändert. Dies ist insofern problematisch, als daß RRL beim Beweis einer Beweisaufgabe zuerst versucht zu simplifizieren. Dabei werden sowohl Definitionen und Eigenschaften, als auch die zuvor bewiesenen Beweisaufgaben zur Simplifikation herangezogen. Nun kann durch diese Modularisierung, insbesondere durch die damit verbundenene Änderung der Reihenfolge, in der die Beweisaufgaben behandelt werden, bei einem Beweis eine Simplifikation möglich werden, die im Originalbeweis nicht möglich war oder umgekehrt. Im schlimmsten Fall kann der Beweis wegen einer solchen Simplifikation scheitern, etwa weil eine 'gute' Induktionsstelle 'wegsimplifiziert' wurde.
Ein anderes Problem, welches im Zusammenhang mit dieser Modularisierung auftritt, findet sich in der Heuristik, die RRL verwendet um Induktionsstellen zu ermitteln. In dieser Heuristik spielt der Sachverhalt, ob ein Operator ein AC-Operator ist oder nicht, eine wichtige Rolle. Läßt man einen Operator 'zu früh' zu einem AC-Operator werden, können bei den nachfolgenden Beweisaufgaben andere Induktionsstellen als im Originalbeweis bevorzugt werden. Dies kann zum Scheitern von Beweisen führen. Weitere Probleme, die man sich durch Modularisierungen in RRL einhandeln kann, sind im Kapitel Erfahrungen (6) beschrieben.

Konkret mußte bei dieser Modularisierung folgendes beachtet werden.

- `(u >= z) == not(u < z)` darf nicht im Modul **arithmetic** bewiesen werden, da sonst dieses Theorem bei einer Beweisaufgabe aus **primes** zur Simplifikation herangezogen wird und aus diesem Grund die nachfolgende Induktion scheitert.

- Der Operator `*` darf erst in dem Modul **arithmetic** AC werden (also nicht im Modul **nat**), da sonst in der internen Repräsentation des cover-sets von $*$ ein Flag gesetzt wird, welches die Bevorzugung anderer Terme und Variablen bei den nachfolgenden Induktionen [1] zur Konsequenz hat. Wird nun `*` 'zu früh' AC, so scheitern einige der nachfolgenden Beweise vor allem durch eine andere Wahl der Induktionsstelle im Vergleich zum Original.

Abbildung 5.3 zeigt die zahlenmäßige Aufteilung der Beweisaufgaben auf die Module. Im Modul **nat** befinden sich also keine Beweisaufgaben, in dem Modul **nat1** 15 Beweisaufgaben, usw.

Eine weitere Besonderheit dieser Modularisierung findet sich in den drei exportierten Hypothesen aus dem Modul **arithmetic**.

- $\mathrm{rem}(y, y) == 0$ [ h , e ]

- $\mathrm{rem}(\mathrm{suc}((y + x)), y) == \mathrm{rem}(\mathrm{suc}(x), y)$ [ h , e ]

[1] d.h. vor allem bei der Suche nach Induktionsstellen

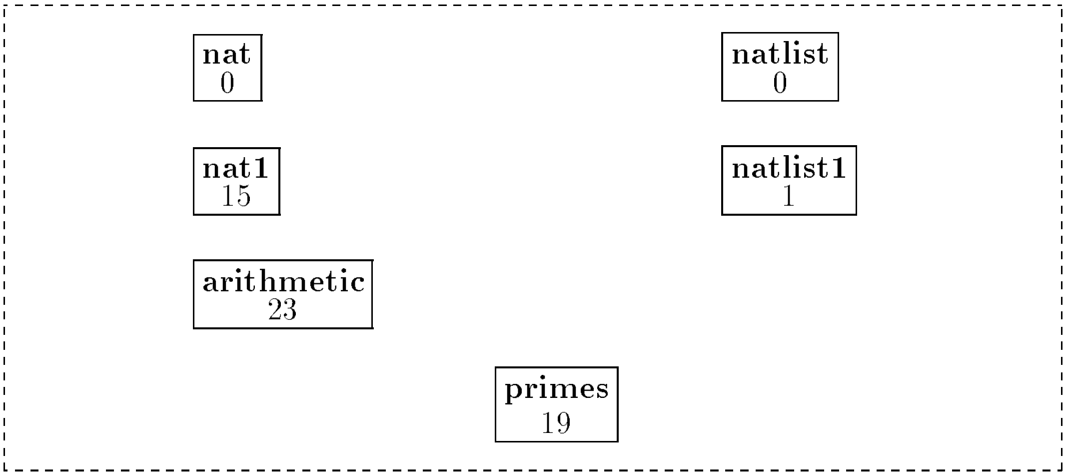



Abbildung 5.3: Zahlenmäßige Aufteilung der Beweisaufgaben auf die Module

- div(y, y) == suc(0) when not(0 = y) == true [ h , e ]

Diese werden als Hypothesen angegeben, da RRL sie zumindest nicht unmittelbar mit der Methode 'cover-set induction' als induktiv gültig nachweisen kann, diese aber für den Beweis der Beweisaufgaben aus dem Modul **primes** notwendig sind. Die Hypothesen entstehen durch Bilden Kritischer Paare zwischen den Definitionen aus dem Modul **arithmetic**.

## 5.4 ASF-Files

Bevor die einzelnen ASF-Files angegeben werden, einige Bemerkungen zum prinzipiellen Vorgehen bei der Modularisierung:
In den Modulen **nat** bzw. **natlist** wurden lediglich die Konstruktoren `0`, `suc` bzw. `null`, `cons` deklariert. Bei den anderen Modulen **nat1**, **natlist1**, **arithmetic** und **primes** wurde wie folgt (siehe auch Abbildung 5.4) verfahren:
Das gesamte Modul wurde innerhalb der Filestruktur (siehe Kapitel 3.3) über drei Files verteilt, nämlich dem *Basis-File*, der *1. Eigenschaft* und der *1. Implementation der 1. Eigenschaft*. Das Basis-File enthält neben der vollständigen Signatur des Moduls auch die für die in der Signatur enthaltenen Funktionen entsprechende Definitionen. Dieses ASF-File wurde nach RRL übersetzt und bzgl. der Beweismethode 'cover-set induction' in RRL ausgeführt. In der entsprechenden RRL-Sitzung wurden im wesentlichen nur die Definitionen orientiert, d.h. an neuen Informationen sind in RRL lediglich Ordnungsinformationen, also Präzedenz- und Statusinformationen, entstanden. Die Rückübersetzung von RRL nach ASF erzeugte nun die *1. Implementation*, welche, wie oben schon beschrieben, nur Ordnungsinformationen zum Inhalt hatte. Die erste Implementation wurde dann nachträglich um die echten Beweisaufgaben und Hypothesen erweitert. Nach der Übersetzung der *1. Implementation* nach RRL und deren Ausführung in RRL bzgl. der Beweismethode 'cover-set induction' wurde durch die Rückübersetzung von RRL nach ASF die *1. Eigenschaft der 1. Implementation* ange-

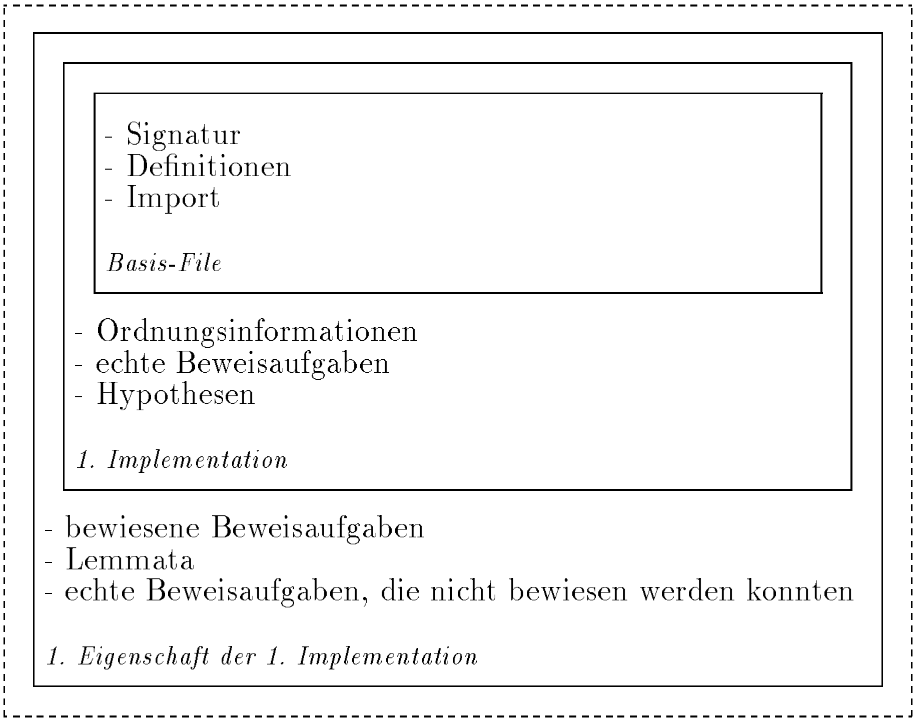



Abbildung 5.4: Repräsentation eines Moduls innerhalb der Filestruktur

legt. Diese enthält neben den echten Beweisaufgaben, die bewiesen werden konnten, und den dabei angefallenen Lemmata auch die Beweisaufgaben, deren Beweis in RRL gescheitert war. Die gescheiterten Beweisaufgaben werden dabei wiederum als echte Beweisaufgaben angegeben. Importiert wurde bis auf wenige Ausnahmen immer die *1. Eigenschaft der 1. Implementation*, und zwar immer aus dem *Basis-File* heraus, zum Beispiel importiert das Basis-File des Moduls **arithmetic** die 1. Eigenschaft der 1. Implementation des Moduls **nat1**.
Da sich die Module in ihrem prinzipiellen Aufbau sehr ähneln (siehe Abbildung 5.4), werden lediglich die Module **nat** und **nat1** stellvertretend für alle anderen Module ausführlicher beschrieben.

### 5.4.1 Modul nat

#### Basis-File

Das Basis-File `nat.asf` exportiert die Sorte `NAT` und die Funktionen `0` und `suc`. Durch den Import von `boolean` stehen die Sorte `bool` und die üblichen booleschen Funktionen zur Verfügung. Auf RRL-Seite stehen einem somit die booleschen built-in Funktionen von RRL zur Verfügung. Die Operatoren `0` und `suc` werden als freie Konstruktoren angegeben.

```
specification nat
begin
  exports
```

```
    begin
      sorts NAT
      functions
        0        :           -> NAT
        suc      : NAT       -> NAT
    end

  imports boolean

  properties
    begin
      constructors
        0 : f
        suc : f
     end
end
```

### 5.4.2 Modul nat1

#### Basis-File

Das Basis-File `nat1.asf` exportiert die Funktionen `+ , * , - , < , >=` und importiert das zuvor beschriebene Modul `nat`. Die Variablen `x , y, z, u` werden als Variablen der Sorte `NAT` deklariert. Nachfolgend werden die üblichen Gleichungen für die exportierten Funktionen angegeben.

```
specification nat1
begin
  exports
    begin
      functions
        _ + _  : NAT # NAT -> NAT
        _ * _  : NAT # NAT -> NAT
        _ - _  : NAT # NAT -> NAT
        _ < _  : NAT # NAT -> bool
        _ >= _ : NAT # NAT -> bool

    end

  imports
    nat^^

  variables
    x,y,z,u : -> NAT
```

```
  equations
    [N1a] x + 0 == x ;
    [N1b] x + suc(y) == suc(x + y) ;

    [N2a] 0 - x == 0 ;
    [N2b] x - 0 == x ;
    [N2c] suc(x) - suc(y) == x - y ;

    [N3a] x * 0 == 0 ;
    [N3b] x * suc(y) == x + (x * y) ;

    [N4a] x < 0 == false ;
    [N4b] 0 < suc(x) == true ;
    [N4c] suc(x) < suc(y) == x < y ;

    [N5a] x >= x == true ;
    [N5b] 0 >= suc(y) == false ;
    [N5c] suc(x) >= y == x >= y when (suc(x) = y) == false ;

end
```

**1. Implementation**

In RRL sind die angegebenen Ordnungsinformationen entstanden, nachträglich wurde die von ART gelieferte 1. Implementation vom Benutzer um die echten Beweisaufgaben erweitert.

```
specification nat1
begin

  properties
  begin
    ordering
      precedence * +, >= <
      status * : r

    theorems
      [T1] x + y == y + x [e] ;
      [T2] (x * (y + z)) == ((x * y) + (x * z)) [e] ;
      [T4] x < suc(0) == x = 0 [e] ;
      [T5] (x + y) = 0 == and((x = 0) ,(y = 0)) [e] ;
      [T6] (x + y) = y == (x = 0) [e] ;
      [T7] (x + z) = (y + z) == (x = y) [e] ;
      [T8] (x * y) = 0 == or((x = 0) ,(y = 0)) [e] ;
      [T9] (x * y) = x == (y = suc(0)) when not(x = 0) == true [e]  ;
```

```
      [T10] (x * y) = x == or((y = suc(0)) ,(x = 0)) [e] ;
      [T11] (x * y) = suc(0) == and ((x = suc(0)) ,(y = suc(0))) [e] ;
      [T12] y < suc(y) == true [e] ;
      [T13] 0 >= u == u = 0 [e] ;
      [T14] suc(x) < y  == true
            when and((x < y) ,not(suc(x) = y)) == true [e] ;
      [T15] u >= z == not(u < z) [e] ;
      [T16] (u * y) < suc(y) == false
            when and(not(u = 0) ,and(not(u = suc(0)) ,
                                 not(y = 0))) == true [e] ;
  end
end
```

**1. Eigenschaft der 1. Implementation**

Alle Beweisaufgaben der 1. Eigenschaft konnten in RRL bewiesen werden. Nachträglich wurden die Lemmata, die in RRL entstanden waren, zum Export freigegeben. ART ordnet bei der Rückübersetzung diesen Lemmata lediglich das Flag 'o' für 'ok' zu, möchte man diese Lemmata anderen Modulen zugänglich machen, muß das Flag 'e' für 'export' hinzugefügt werden.

```
specification nat1
begin

  variables
    z1 : -> NAT

  properties
  begin
    ac-operators +
    theorems
      ((u * y) < suc(y)) == false when equ((0 = u), false) == true ,
                               equ((suc(0) = u), false) == true ,
                               equ((0 = y), false) == true [ o, e ] ;
      ((u + z) < suc(suc(y))) == false when equ((0 = u), false) == true ,
                               equ((suc(0) = u), false) == true ,
                               equ((z < suc(y)), false) == true [ o, e ] ;
      ((u + x) < suc(y)) == false when
                               equ(((u + x) < suc(suc(y))), false) == true ,
                               equ((x < y), false) == true ,
                               equ((suc(0) = u), false) == true ,
                               equ((0 = u), false) == true [ o, e ] ;
      ((u + x) < suc(y)) == false when (x < suc(y)) == true ,
                               equ((x < y), false) == true ,
                               equ((suc(0) = u), false) == true ,
```

```
                          equ((0 = u), false) == true [ o, e ] ;
    (u < suc(suc(0))) == false when equ((suc(0) = u), false) == true ,
                          equ((0 = u), false) == true [ o, e ] ;
    (suc(x) < z) == (x < z) when equ((suc(x) = z), false) == true [ o, e ] ;
    xor(true, (u < u)) == true [ o, e ] ;
    (0 < x) == true when equ((0 = x), false) == true [ o, e ] ;
    (0 >= u) == (0 = u) [ o, e ] ;
    ((x * y) = suc(0)) == false when
                          equ((suc(0) = x), false) == true [ o, e ] ;
    ((x + z) = suc(0)) == false when
                          equ((suc(0) = x), false) == true ,
                          equ((suc(0) = z), false) == true [ o, e ] ;
    ((x * y) = x) == xor(xor((0 = x), (suc(0) = y)),
                          and((0 = x), (suc(0) = y))) [ o, e ] ;
    ((x * y) = x) == (suc(0) = y) when
                          equ((0 = x), false) == true [ o, e ] ;
    ((x * y) = 0) == xor(xor((0 = x), (0 = y)),
                           and((0 = x), (0 = y))) [ o, e ] ;
    ((y + z) = (x + z)) == (y = x) [ o, e ] ;
    ((x + y) = y) == (0 = x) [ o, e ] ;
    ((x + y) = 0) == false when equ((0 = x), false) == true [ o, e ] ;
    (x < suc(0)) == (0 = x) [ o, e ] ;
    (x + (z + z1)) == (z + (x + z1)) [ o, e ] ;
    (suc(y) + x) == suc((y + x)) [ o, e ] ;
    (0 + x) == x [ o, e ] ;
    ((u * y) < suc(y)) == false when and(not((u = 0)),
                    and(not((u = suc(0))), not((y = 0)))) == true [ o, e ] ;
    (suc(x) < y) == true when and((x < y),
                           not((suc(x) = y))) == true [ o, e ] ;
    (suc(x) < y) == true when equ((suc(x) = y), false) == true ,
                           (x < y) == true [ o, e ] ;
    (0 >= u) == (u = 0) [ o, e ] ;
    (y < suc(y)) == true [ o, e ] ;
    ((x * y) = suc(0)) == and((x = suc(0)), (y = suc(0))) [ o, e ] ;
    ((x * y) = x) == or((y = suc(0)), (x = 0)) [ o, e ] ;
    ((x * y) = x) == (y = suc(0)) when not((x = 0)) == true [ o, e ] ;
    ((x * y) = 0) == or((x = 0), (y = 0)) [ o, e ] ;
    ((x + z) = (y + z)) == (x = y) [ o, e ] ;
    ((x + y) = y) == (x = 0) [ o, e ] ;
    ((x + y) = 0) == and((x = 0), (y = 0)) [ o, e ] ;
    (x < suc(0)) == (x = 0) [ o, e ] ;
    (x * (y + z)) == ((x * y) + (x * z)) [ o, e ] ;
    (x + y) == (y + x) [ o, e ] ;

end
```

```
end
```

### 5.4.3 Modul natlist

#### Basis-File

Das Basis-File `natlist.asf` exportiert die Sorte `NATLIST` und die Funktionen `null` und `cons`. Durch den Import des Moduls `nat` steht die Sorte `NAT` zur Verfügung. Die Operatoren `null` und `cons` werden als freie Konstruktoren angegeben.

```
specification natlist
begin
  exports
    begin
      sorts NATLIST
      functions
        null    :                          -> NATLIST
        cons    : NAT # NATLIST            -> NATLIST
    end

  imports
    nat^^

  properties
    begin
      constructors
        null : f
        cons : f
    end

end
```

### 5.4.4 Modul natlist1

#### Basis-File

Das Basis-File `natlist1.asf` exportiert die Funktionen `append` , `delete` , `member` , `perm` und importiert das zuvor beschriebene Modul `natlist`. Die Variablen `x` , `y` werden als Variablen der Sorte `NAT`, die Variablen `nl1` , `nl2` als Variablen der Sorte `NATLIST` deklariert. Es folgen die üblichen Gleichungen für die exportierten Funktionen.

```
specification natlist1
begin
```

```
  exports
    begin
      functions
        append  : NATLIST # NATLIST        -> NATLIST
        delete  : NAT     # NATLIST        -> NATLIST
        member  : NAT     # NATLIST        -> bool
        perm    : NATLIST # NATLIST        -> bool
    end

  imports
    natlist^^

  variables
    x,y : -> NAT
    nl1,nl2 : -> NATLIST

  equations
    [NL1a]  append(null,nl1)        == nl1 ;
    [NL1b]  append(cons(x,nl1),nl2) == cons(x,append(nl1,nl2)) ;

    [NL2a] delete(x, null) == null;
    [NL2b] delete(x, cons(x, nl1)) == nl1 ;
    [NL2c] delete(x, cons(y, nl1)) ==  cons(y, delete(x, nl1))
             when (y = x) == false ;

    [NL3a] member(x, null) == false ;
    [NL3b] member(x, cons(x, nl1)) == true ;
    [NL3c] member(x, cons(y, nl1)) == member(x, nl1)
           when (x = y) == false ;

    [NL4a] perm(null, null) == true ;
    [NL4b] perm(null, cons(x, nl1)) == false ;
    [NL4c] perm(cons(x, nl1), null) == false ;
    [NL4d] perm(cons(x, nl1), nl2) ==
           and(member(x, nl2), perm(nl1 , delete(x, nl2))) ;
end
```

**1. Implementation**

```
specification natlist1
begin

  properties
  begin
    ordering
```

```
      precedence perm member, perm delete
      status perm : l
    theorems
      [T1] delete(x, nl1) == nl1
           when not(member(x, nl1)) == true [e] ;

  end
end
```

**1. Eigenschaft der 1. Implementation**

```
specification natlist1
begin

  properties
  begin
    theorems
      delete(x, nl1) == nl1 when not(member(x, nl1)) == true [ o, e ] ;

  end
end
```

### 5.4.5 Modul arithmetic

**Basis-File**

Das Basis-File `arithmetic.asf` exportiert die Funktionen `div` , `rem` , `divides` , `gcd` , `sub1` und importiert das Modul `nat1`, d.h. die erste Eigenschaft der ersten Implementation des Moduls `nat1`. Die Variablen `x` , `y` , `z` sind Variablen der Sorte `NAT`.

```
specification arithmetic
begin
  exports
    begin
      functions
        div     : NAT # NAT -> NAT
        rem     : NAT # NAT -> NAT
        divides : NAT # NAT -> bool
        gcd     : NAT # NAT -> NAT
        sub1   : NAT -> NAT
    end
  imports
    nat1^1^1
```

```
  variables
    x,y,z : -> NAT

  equations
    [A1a]  div(x, 0) == 0 ;
    [A1b]  div(x, y) == 0
            when (x < y) == true ;
    [A1c]  div((y + x),y) == suc(div(x,y))
            when (0 = y) == false ;

    [A2a]  rem(x, 0) == x ;
    [A2b]  rem(x, y) == x
            when (x < y) == true ;
    [A2c]  rem(y + x, y) == rem(x, y) ;

    [A3a]  divides(x, y) == rem(y,x) = 0 ;

    [A4a]  gcd(x, 0) == x ;
    [A4b]  gcd(0, y) == y ;
    [A4c]  gcd(x + y, y) == gcd(x, y) ;
    [A4d]  gcd(x, x + y) == gcd(x, y) ;

    [A5a]  sub1(0) == 0 ;
    [A5b]  sub1(suc(x)) == x ;

end
```

**1. Implementation**

```
specification arithmetic
begin
  properties
    begin

      ordering
        precedence divides rem

      theorems
        [T1]  div(0, y) == 0 [ e ] ;
        [T2]  rem(0, y) == 0 [ e ] ;
        [T3]  y * div(y + x, y) == y * suc(div(x, y)) [ e ] ;
        [T4]  (rem(x, y) + (y * div(x, y))) == x [ e ] ;
        [T5]  (y * div(x, y)) == x when divides(y, x) == true [ e ] ;
        [T6]  x * y == y * x [ e ] ;
        [T7]  x * (y * z) == (x * y) * z [ e ] ;
```

```
      [T8]  div((x * y), x) == y when not((x = 0)) == true [ e ] ;
      [T9]  rem((y * x), x) == 0 [ e] ;
      [T10]  rem((y * z), x) == 0 when and((rem(z, x) = 0),
                                   not(x = 0)) == true [ e ] ;
      [T11]  rem(x + y, z) == rem(x, z) when (rem(y, z) = 0) == true [ e] ;
      [T12] div(x, y) < x == true
              when and(not(x = 0), and(not(y = 0),
              (not (y = suc(0))))) == true [ e ] ;
      [T13] div(z + y, x) == div(z, x) + div(y, x)
              when and(divides(x, z), not(x = 0)) == true [ e ] ;
      [T14] div(z * y, x) == y * div(z, x)
              when and(divides(x, z), not(x = 0)) == true [ e ] ;
      [T15] rem(suc(0), x) == suc(0)
              when not(x = suc(0)) == true [ e] ;
      [T16] sub1(x) < y == x < suc(y) when not(x = 0) == true [ e ] ;
      [T17] gcd(x, y) == gcd(y, x) [ e ] ;
      [T18] gcd((x * z), (y * z)) == (z * gcd(x, y)) [ e ] ;
      [T19] gcd(x*y, z) = y == false
              when and((rem(z, x) = 0),
                        not(rem(y, x) = 0)) == true [ e ] ;
      [T20] rem(y * z, x) = 0 == false
              when and((gcd(x*y, y*z) = y), and(not(y = 0),
              not(rem(y, x) = 0))) == true [ e ] ;
      [T21] gcd(x, suc(0)) == suc(0) when not((x = 0)) == true [ e ] ;
      [T22] rem(x, gcd(x, y)) == 0 [ e ] ;
      [T23] gcd(x, y) = x == false
              when not(rem(y, x) = 0) == true [ e ] ;
      [CP1] rem(y, y) == 0 [ h , e ] ;
      [CP2] rem(suc((y + x)), y) ==  rem(suc(x), y) [ h , e ] ;
      [CP3] div(y, y) == suc(0) when not(0 = y) == true [ h , e ] ;
  end

end
```

**1. Eigenschaft der 1. Implementation**

```
specification arithmetic
begin

  variables
    z1 : -> NAT

  properties
  begin
    ac-operators *
```

```
c-operators gcd
ordering
  precedence div *, gcd *
  status < : l
theorems
  gcd(x, suc(0)) == suc(0)
    when equ((0 = x), false) == true [ o, e ] ;
  (sub1(x) < y) == (x < suc(y))
    when equ((0 = x), false) == true [ o, e ] ;
  rem(suc(0), x) == suc(0)
    when equ((suc(0) = x), false) == true [ o, e ] ;
  div((y + (x * z)), x) == (z + div(y, x))
    when equ((0 = x), false) == true [ o, e ] ;
  (div(x, y) < x) == true
    when equ((0 = x), false) == true ,
         equ((0 = y), false) == true ,
         equ((suc(0) = y), false) == true [ o, e ] ;
  (suc(z) < (x + y)) == true
    when equ((0 = y), false) == true ,
         equ((suc(0) = y), false) == true ,
         (z < x) == true [ o, e ] ;
  rem((x + (z * z1)), z) == rem(x, z) [ o, e ] ;
  rem((x * y), x) == 0 [ o, e ] ;
  div((x * y), x) == y when equ((0 = x), false) == true [ o, e ] ;
  (x * (y * z)) == (z * (x * y)) [ o, e ] ;
  (suc(y) * x) == (x + (y * x)) [ o, e ] ;
  (0 * x) == 0 [ o, e ] ;
  ((y * div(x, y)) + rem(x, y)) == x [ o, e ] ;
  (y * div((x + y), y)) == (y + (y * div(x, y))) [ o, e ] ;
  (suc(suc((x + y))) < y) == false when ((x + y) < y) == true ,
                                        (x < y) == true [ o, e ] ;
  (suc(x) < x) == false [ o, e ] ;
  (gcd(x, y) = x) == false
    when not((rem(y, x) = 0)) == true [ o, e ] ;
  rem(x, gcd(x, y)) == 0 [ o, e ] ;
  (rem((y * z), x) = 0) == false
    when and((gcd((x * y), (y * z)) = y),
         and(not((y = 0)),
         not((rem(y, x) = 0)))) == true [ o, e ] ;
  (gcd((x * y), z) = y) == false
    when and((rem(z, x) = 0),
         not((rem(y, x) = 0))) == true [ o, e ] ;
  gcd((x * z), (y * z)) == (z * gcd(x, y)) [ o, e ] ;
  gcd(x, y) == gcd(y, x) [ o, e ] ;
  (sub1(x) < y) == (x < suc(y)) when not((x = 0)) == true [ o, e ] ;
```

```
      rem(suc(0), x) == suc(0) when not((x = suc(0))) == true [ o, e ] ;
      div((z * y), x) == (y * div(z, x))
      when and(divides(x, z),
           not((x = 0))) == true [ o, e ] ;
      div((z + y), x) == (div(z, x) + div(y, x))
        when and(divides(x, z),
             not((x = 0))) == true [ o, e ] ;
      (div(x, y) < x) == true
        when and(not((x = 0)), and(not((y = 0)),
             not((y = suc(0))))) == true [ o, e ] ;
      rem((x + y), z) == rem(x, z)
        when (rem(y, z) = 0) == true [ o, e ] ;
      rem((y * z), x) == 0
        when and((rem(z, x) = 0), not((x = 0))) == true [ o, e ] ;
      rem((y * x), x) == 0 [ o, e ] ;
      div((x * y), x) == y when not((x = 0)) == true [ o, e ] ;
      (x * (y * z)) == ((x * y) * z) [ o, e ] ;
      (x * y) == (y * x) [ o, e ] ;
      (y * div(x, y)) == x when divides(y, x) == true [ o, e ] ;
      (rem(x, y) + (y * div(x, y))) == x [ o, e ] ;
      (y * div((y + x), y)) == (y * suc(div(x, y))) [ o, e ] ;
      rem(0, y) == 0 [ o, e ] ;
      div(0, y) == 0 [ o, e ] ;

  end
end
```

### 5.4.6 Modul primes

#### Basis-File

Das Basis-File `primes.asf` exportiert die Funktionen `prime` , `primelist` , `primefac` und importiert die Module `arithmetic.impl_1.prop_1` und `natlist1.impl_1.prop_1`, also jeweils die erste Eigenschaft der ersten Implementation. Dieses File verfügt desweiteren über eine 'versteckte' Signatur, denn die Funktionen `prime1` und `timelist` sind nur innerhalb dieses Files (und all seinen Erweiterungen) sichtbar. Nach der Deklaration der Variablen folgen die Gleichungen der exportierten bzw. 'versteckten' Funktionen.

```
specification primes
begin
  exports
    begin
      functions
        prime     : NAT     -> bool
```

```
        primelist : NATLIST -> bool
        primefac  : NAT      -> NATLIST
    end

  imports
    arithmetic^1^1,
    natlist1^1^1

  functions
    prime1   : NAT # NAT -> bool
    timelist : NATLIST   -> NAT

  variables
    x,y  : -> NAT
    nl : -> NATLIST

  equations
    [P1a] prime(0) := false ;
    [P1b] prime(suc(x)) := prime1(suc(x), x)  ;

    [P2a] primelist(null) := true ;
    [P2b] primelist(cons(x,nl)) := and(prime(x),primelist(nl)) ;

    [P3a] timelist(null) := suc(0) ;
    [P3b] timelist(cons(x,nl)) := x * timelist(nl) ;

    [P4a] primefac(0) := null ;
    [P4b] primefac(suc(0)) := null ;
    [P4c] primefac(x * y) := append(primefac(x), primefac(y))
            when (x = 0) == false, (y = 0) == false ;

    [P5a] prime1(x, 0) := false ;
    [P5b] prime1(x, suc(0)) := true ;
    [P5c] prime1(x, suc(y)) := and(not(divides(suc(y), x)), prime1(x, y))
            when not(y = 0) == true ;
end
```

**1. Implementation**

Theorem T15 enthält die eigentliche Kernaussage, nämlich die Eindeutigkeit der Primfaktorzerlegung.

```
specification primes
begin
  variables
```

```
  z,u  : -> NAT
  nl1  : -> NATLIST
properties
begin
  ordering
    precedence primelist prime, timelist *, primefac append,
               prime prime1, prime1 rem
  theorems
    [T]  divides(x, timelist(nl)) == true
           when member(x, nl) == true [ e ] ;
    [T]  (x = suc(0)) == false
           when prime(x) == true [ e ] ;
    [T]  (x = 0) == false
           when prime(x) == true [ e ] ;
    [T1] primelist(delete(x, nl)) == true
           when primelist(nl) == true [ e ] ;
    [T2] timelist(delete(x, nl)) == div(timelist(nl), x)
           when and(not(x = 0), member(x, nl)) == true [ e ] ;
    [T3] timelist(nl) = 0 == false
           when  primelist(nl) == true [ e ] ;
    [T4] timelist(append(nl, nl1)) == (timelist(nl) * timelist(nl1))
           when and(not(timelist(nl) = 0),
                not(timelist(nl1) = 0)) == true [ e ] ;
    [T5] timelist(primefac(x)) == x
           when not(x = 0) == true [ e ] ;
    [T6] primelist(primefac(x)) == true
           when not(x = 0) == true [ e ] ;
    [T7] prime1(x*z, u) == false
           when and(not(z = suc(0)), and(not(z = 0),
                and((u >= z), not(u = suc(0))))) == true [ e ] ;
    [T]  u >= z == not(u < z) [ e ] ;
    [T8] prime1(x, sub1(x)) == false
           when and(not(z = suc(0)), and(not(z = x),
                and(not(x = 0), and(not(x = suc(0)),
                divides(z, x))))) == true [ e ] ;
    [T9] rem(x, y) = 0 == false
           when and(prime(x), and(not(y = suc(0)),
                    not(x = y))) == true [ e ] ;
   [T10] prime(y) == false
         when and((rem(x, y) = 0), and(prime(x),
                 not(x = y))) == true [ e ] ;
   [T11] gcd(x, y) == suc(0)
           when and((rem(x, gcd(x, y)) = 0),
                and(not(x = 0), and(not(x = suc(0)),
                and(prime1(x, sub1(x)),
```

```
                    not(gcd(x, y) = x))))) == true [ e ] ;
     [T12] prime1(x, sub1(x)) == false
             when and(not(rem(y, x) = 0),
                  not(gcd(x,y) = suc(0))) == true [ e ] ;
     [T13] rem(y * z, x) = 0 == false
             when and( prime(x), and(not(divides(x, y)),
                  not(divides(x, z)))) == true [ e ] ;
     [T14] member(x, nl) == true
             when and(prime(x), and(primelist(nl),
                      divides(x, timelist(nl)))) == true [ e ] ;
     [T15] perm(nl, nl1) == true
             when and(primelist(nl), and(primelist(nl1),
            (timelist(nl) = timelist(nl1)))) == true [ e ] ;
  end
end
```

### 1. Eigenschaft der 1. Implementation

Die erste der angegebenen Theoreme zeigt, daß RRL die Kernaussage (d.h. T15 der 1. Eigenschaft) erfolgreich beweisen konnte.

```
specification primes
begin

  variables
    z1 : -> NAT
    zn : -> NATLIST
    z1n : -> NATLIST

  properties
  begin
    ordering
      precedence timelist div
    theorems
      perm(nl, nl1) == true when and(primelist(nl),
        and(primelist(nl1),
           (timelist(nl) = timelist(nl1)))) == true [ o, e ] ;
      perm(nl, nl1) == true when primelist(nl1) == true ,
        primelist(nl) == true ,
        (timelist(nl) = timelist(nl1)) == true [ o, e ] ;
      member(x, nl) == true when and(prime(x), and(primelist(nl),
        divides(x, timelist(nl)))) == true [ o, e ] ;
      member(x, nl) == true
        when prime(x) == true , primelist(nl) == true ,
             (rem(timelist(nl), x) = 0) == true [ o, e ] ;
```

```
prime(y) == false when and((rem(x, y) = 0), and(prime(x),
  not((x = y)))) == true [ o, e ] ;
(rem((y * z), x) = 0) == false
  when equ((rem(y, x) = 0), false) == true ,
       equ((rem(z, x) = 0), false) == true ,
       prime(x) == true [ o, e ] ;
(rem(x, y) = 0) == false when and(prime(x), and(not((y = suc(0))),
  not((x = y)))) == true [ o, e ] ;
(rem(x, y) = 0) == false when equ((y = x), false) == true ,
  equ((suc(0) = y), false) == true , prime(x) == true [ o, e ] ;
prime1(x, sub1(x)) == false when and(not((z = suc(0))),
  and(not((z = x)),
  and(not((x = 0)), and(not((x = suc(0))),
  divides(z, x))))) == true [ o, e ] ;
prime1((z * z1), sub1((z * z1))) == false
  when equ((suc(0) = z1), false) == true ,
       equ((0 = z1), false) == true ,
       equ((0 = z), false) == true ,
       equ((suc(0) = z), false) == true [ o, e ] ;
(u >= z) == not((u < z)) [ o, e ] ;
(u >= z) == xor(true, (u < z)) [ o, e ] ;
prime1((x * z), u) == false
  when and(not((z = suc(0))), and(not((z = 0)),
       and((u >= z), not((u = suc(0)))))) == true [ o, e ] ;
prime1((x * z), u) == false when equ((suc(0) = u), false) == true ,
  equ((0 = z), false) == true ,
  equ((suc(0) = z), false) == true ,
  (u >= z) == true [ o, e ] ;
prime1((u * x), u) == false when equ((0 = u), false) == true ,
  equ((suc(0) = u), false) == true [ o, e ] ;
rem((x * z), suc(suc(0))) == 0 when (suc(suc(0)) >= z) == true ,
  equ((suc(0) = z), false) == true ,
  equ((0 = z), false) == true [ o, e ] ;
primelist(primefac(x)) == true when not((x = 0)) == true [ o, e ] ;
primelist(primefac(x)) == true
  when equ((0 = x), false) == true [ o, e ] ;
primelist(append(zn, z1n)) == true when primelist(z1n) == true ,
  primelist(zn) == true [ o, e ] ;
timelist(primefac(x)) == x when not((x = 0)) == true [ o, e ] ;
timelist(primefac(x)) == x
  when equ((0 = x), false) == true [ o, e ] ;
timelist(append(nl, nl1)) == (timelist(nl) * timelist(nl1))
  when and(not((timelist(nl) = 0)),
           not((timelist(nl1) = 0))) == true [ o, e ] ;
timelist(append(nl, nl1)) == (timelist(nl1) * timelist(nl))
```

```
        when equ((timelist(nl1) = 0), false) == true ,
        equ((timelist(nl) = 0), false) == true [ o, e ] ;
      (timelist(nl) = 0) == false when primelist(nl) == true [ o, e ] ;
      timelist(delete(x, nl)) == div(timelist(nl), x)
        when and(not((x = 0)), member(x, nl)) == true [ o, e ] ;
      timelist(delete(x, nl)) == div(timelist(nl), x)
        when equ((0 = x), false) == true , member(x, nl) == true [ o, e ] ;
      (x = suc(0)) == false when prime(x) == true [ o, e ] ;
      primelist(delete(x, nl)) == true
        when primelist(nl) == true [ o, e ] ;
      divides(x, timelist(nl)) == true
        when member(x, nl) == true [ o, e ] ;
      rem(timelist(nl), x) == 0 when member(x, nl) == true [ o, e ] ;

  end
end
```

# Kapitel 6

# Erfahrungen

Dieses Kapitel stellt einige Erfahrungen mit ART vor, die im wesentlichen bei der Primfaktorzerlegung (siehe Kapitel 5), d.h. vielmehr bei der entsprechenden Modularisierung angefallen sind. Die hier vorgestellten Erfahrungen beziehen sich in den meisten Fällen auf Probleme und Schwierigkeiten mit dem Beweiser RRL. Es hat sich zum Beispiel herausgestellt, daß die von RRL verwendeten Strategien bzw. Heuristiken von RRL Auswirkungen bis in die Modularisierung haben können. Alle Erfahrungen werden lediglich informell und vereinfachend dargestellt, insbesondere wurde auf theoretische Formalismen verzichtet. Falls nicht ausdrücklich anders vermerkt, wird im folgenden unter Induktion die Beweismethode 'cover-set induction' verstanden.

## 6.1 Modularisierung von Beweisen

Die Untersuchung der Modularisierbarkeit von Beweisen legt zwei Vorgehensweisen nahe. Zum einen können in einer Spezifikation Beweisaufgaben verankert und die um die in RRL bewiesenen Theoreme erweiterte Spezifikation in einer größeren Spezifikation importiert werden (d.h. man beweist sozusagen 'von unten nach oben'), zum anderen kann ein in RRL bestehender Beweis modularisiert werden (d.h. hier beweist man gewissermaßen 'von oben nach unten'). Vor allem wegen der zweiten Vorgehensweise wurde in ASF die Möglichkeit geschaffen, die in RRL eingebauten boole'schen Funktionen and, or, xor,... und das Gleichheitsprädikat = zu benutzen. RRL hat für seine eingebauten boole'schen Funktionen eigene Vorgehensweisen [1], die mit selbstdefinierten boole'schen Funktionen nur schwer simuliert werden können.

## 6.2 Induktion

Beim Arbeiten mit dem System ART, also bei der Modularisierung von Beweisen, hat sich herausgestellt, daß RRL den Benutzer zu wenig Einfluß auf die Induktion nehmen

[1] etwa bei der Simplifikation und beim 'contextual rewriting'

läßt. Es wäre wünschenswert, daß der Benutzer folgende Angaben machen könnte :

- Angabe des Terms und der Variablen, über welche die Induktion geführt werden soll.
- Generalisierung (ja/nein)
- Abstraktions-Methode (ja/nein)
- Simplifikation (ja/nein)

Da hier kein Einfluß genommen werden kann, diese Punkte aber Auswirkungen bis hin in die Modularisierung haben, muß schon bei der Modularisierung auf RRL-Strategien [2] Rücksicht genommen werden, was in den meisten Fällen aber nicht unmittelbar vom Benutzer [3] einzusehen ist.

## 6.3 Simplifikation

Theoreme sind in einem großen Beweis oft untereinander verflochten: einerseits muß zum Beweiszeitpunkt über bestimmtes Wissen verfügt werden, andererseits kann zusätzliches Wissen jedoch dazu führen, daß ein Beweis nicht erfolgreich geführt werden kann. Modularisiert man einen großen Beweis, kann ein 'zu früh' bewiesenes Theorem zur Simplifikation von 'späteren' Beweisaufgaben herangezogen werden, und somit die Induktion zum Scheitern bringen, etwa weil eine gute Induktionsstelle 'wegsimplifiziert' wurde. Die Reihenfolge der Beweise spielt also eine nicht zu unterschätzende Rolle. Oft ist man gezwungen eine Beweisaufgabe innerhalb der Modularisierung aus einem kleinen Modul in ein größeres zu schieben, um den Erfolg der Beweise zu gewährleisten. Die Strategie, in einem Modul möglichst viel zu beweisen, kann daher in größeren Modulen zu Schwierigkeiten führen. Dies sollte vor allen Dingen beachtet werden, wenn man sich nicht an einem bestehenden Beweis orientiert, sondern 'von unten nach oben' beweist Hier hat man nämlich nicht die Möglichkeit, sich am Original zu orientieren, wenn ein Beweis scheitert. Das oben beschriebene Problem ist bei der Primfaktorzerlegung (siehe Kapitel 5) aufgetreten: Die Beweisaufgabe `(u >= z) == not(u < z)` durfte nicht im Modul **arithmetic** bewiesen werden, da sonst dieses Theorem bei einer Beweisaufgabe aus **primes** zur Simplifikation herangezogen wird und aus diesem Grund die nachfolgende Induktion scheitert.
Ein weiterer Punkt in diesem Zusammenhang ist die Strategie, mit der RRL geeignete Regeln zur Simplifikation auswählt. Die Strategie ist denkbar einfach, sie lautet vereinfacht ausgedrückt : Nehme 'outermost' die 'erste' Regel, die sich anwenden läßt. RRL verwaltet die Regelmenge in einer Liste. Die Reihenfolge der Regeln in dieser Liste entspricht der Reihenfolge, in welcher die Regeln in RRL aufgenommen wurden. Da hier die 'erste' Regel genommen wird, kommt auch aus diesem Grund der Reihenfolge

[2]beispielsweise auf die Strategie zur Simplifikation oder zur Suche nach geeigneten Induktionsstellen. Diese Auswirkungen werden in den folgenden Punkten genauer beschrieben.

[3]bzw. Spezifizierer

der Regeln [4] eine wichtige Rolle zu.

## 6.4 Induktionsstellen

Bei der Suche nach Induktionsstellen in RRL spielt der Sachverhalt, ob ein Operator ein AC-Operator ist oder nicht, eine wichtige Rolle. In der in Kapitel 5 vorgestellten Modularisierung führte genau dies zu einem Problem. Läßt man in dieser Modularisierung den Operator $*$ schon in dem Modul **nat** zu einem AC-Operator werden, wird bei einer Beweisaufgabe aus dem Modul **arithmetic** eine andere Induktionsstelle als im Originalbeweis bevorzugt. Dies führt zum Scheitern der nachfolgenden Induktion. Orientert man sich bei der Modularisierung an einem Original-Beweis, muß dies berücksichtigt werden, insbesondere wenn sich in der Modularisierung die Reihenfolge der Beweisaufgaben in Bezug auf den Original-Beweis ändert.

## 6.5 Reihenfolge der Definitionen

Wird die Methode *cover-set induction* verwendet, so ist auch die Reihenfolge der Definitionen zu beachten. Baut eine Funktion in ihren Definitionen auf eine andere auf, sollten diese nicht vor den Definitionen der anderen Funktion angegeben werden [5]. RRL trägt beim Einlesen der Gleichungen [6] alle auftauchenden Funktionssymbole in dieser Reihenfolge in eine Liste ein. Die Reihenfolge der Symbole in dieser Liste spielt bei der Suche nach einer Induktionsstelle eine wichtige Rolle.

## 6.6 Abstraktions-Methode

Es sei hier noch erwähnt, daß es bei der Methode *cover-set induction* wichtig ist, vor den Eigenschaften die cover-sets zu ermitteln und die Beweismethode zu setzen. Versäumt man die cover-sets zu ermitteln, wird keine der Eigenschaften für die Abstraktions-Methode vorgesehen [7], ist die Beweismethode nicht gesetzt, erhält die Variable $induc nicht den richtigen Wert. In ART wird dieser Sachverhalt natürlich beachtet [8]; dieser Hinweis richtet sich vielmehr an den, der direkt mit RRL arbeitet.

[4] in dieser Liste

[5] also zum Beispiel nicht die Definitionen der Division vor denen der Addition

[6] Definitionen und Eigenschaften

[7] also nicht in die Lisp-Variable $build-rules eingetragen

[8] Anfänglich ist dies in ART nicht beachtet worden, worauf eine langwierige Suche nach der Ursache notwendig war, nachdem die Abstraktions-Methode zwar im Originalbeweis der Primfaktorzerlegung (siehe Kapitel 5), aber nicht im entsprechenden ART-Beweis angewendet worden war.

## 6.7 Linearität

**Nicht** lineare Beweisaufgaben können in RRL zu fehlerhaften Induktionen führen. Es kann vorkommen, daß nicht jedes Vorkommen einer Variablen im Induktionsschritt richtig substituiert wird. Eine Möglichkeit, wie man sich oft behelfen kann, wird in dem anschließenden Beispiel verdeutlicht.

**Beispiel 7.0.1:**

```
is_subset(u, rev(u)) == true
```

Diese Gleichung kann in RRL problematisch sein.
Behelfen kann man sich etwa durch :

```
is_subset(u, rev(v)) == true if u = v
```

## 6.8 Fallunterscheidungen, Größe der Cover-Sets

Ist die Anzahl der definierenden Gleichungen einer Funktion groß, führt dies in den meisten Fällen auch zu einem großen cover-set, und daher auch zu einem Induktions-Schema mit vielen Beweispunkten. Ist eine Funktion über eine große Fallunterscheidung definiert, hat dies den Vorteil, daß in jedem Beweispunkt viel Wissen über den speziellen Fall vorhanden ist, aber den Nachteil, daß viele Beweispunkte anfallen. Ebenso kann es für eine Funktion eine bedingte und eine unbedingte Definition geben, hier ergeben sich dann entsprechende Probleme. Insgesamt muß die Definition einer Funktion sorgsam überlegt und im Hinblick auf das zu Beweisende ausgewählt werden.

**Beispiel 8.0.1:** Unterschiedliche Definitionen von >=
Seien `< : NAT # NAT -> bool` und `>= : NAT # NAT -> bool` auf den freien Konstruktoren `0 : -> NAT`, `succ : NAT -> NAT` definiert durch

```
x < 0 := false
0 < succ(x) := true
succ(x) < succ(y) := x < y

x >= x := true
0 >= succ(y) := false
succ(x) >= y := (x >= y) if not(succ(x) = y)
```

Die in diesem Zusammenhang oft auftauchenden Theoreme
`x >= y if y < x` bzw. `succ(x) >= succ(y) if x >= y`
werfen bei ihrem Nachweis Schwierigkeiten auf. Eine Induktion über `>=` verläuft nach folgendem Schema :

```
The induction will be done on x, y in (x >= y), and will follow the scheme:
    [1] P(x, x)
    [2] P(0, succ(y))
    [3] P(succ(x), y) if
        { (succ(x) = y) <=> false,
          P(x, y) }
```

Eine alternative, unbedingte Version von `>= : NAT # NAT -> bool`

```
x >= x := true
0 >= succ(y) := false
succ(x) >= succ(y)  := (x >= y)
```

führt zu einem anderen cover-set, und damit auch zu einem anderen Induktions-Schema:

```
The induction will be done on x, y in (x >= y), and will follow the scheme:
    [1] P(x, x)
    [2] P(0, succ(y))
    [3] P(succ(x), succ(y)) if  { P(x, y) }
```

Mit diesem Schema lassen sich beide Theoreme leicht beweisen [9]. Innerhalb der Primfaktorzerlegung findet sich erstaunlicherweise die auf den ersten Blick komplizierter aussehende erste Definition der Funktion `>=`.

**Beispiel 8.0.2:**
Im folgenden werden zwei Definitionen des Prädikats `sorted` angegeben. Dieses Prädikat entscheidet, ob eine auf den freien Konstruktoren `nil : -> LIST` und `cons : NAT # LIST -> LIST` erzeugte Liste sortiert ist. Die erste Definition unterscheidet sich von der zweiten im wesentlichen dadurch, daß sie ohne Bedingungen auskommt und vor allem, daß die Fälle `(x = y)` und `(x < y)` in einer Gleichung zusammenfallen.

```
;; 1. Definition
sorted(nil) := true
sorted(cons(x, nil)) := true
sorted(cons(x, cons(y, u))) := ((x = y) or (x < y)) and sorted(cons(y, u))

;; 2. Definition
sorted(nil) := true
sorted(cons(x, nil)) := true
sorted(cons(x, cons(y, u))) := sorted(cons(y, u)) if (x = y)
```

[9] Das letzte ist sogar ein Gleichheits-Theorem.

```
sorted(cons(x, cons(y, u))) := sorted(cons(y, u)) if (x < y)
sorted(cons(x, cons(y, u))) := false  if (y < x)
```

Induktions-Schema für die erste Definition :

```
The induction will be done on u in sorted(u), and will follow the scheme:
    [1] P(null)
    [2] P(cons(x, null))
    [3] P(cons(x, cons(y, u))) if  { P(cons(y, u)) }
```

Induktions-Schema für die zweite Definition :

```
The induction will be done on u in sorted(u), and will follow the scheme:
    [1] P(null)
    [2] P(cons(x, null))
    [3] P(cons(y, cons(y, u))) if  { P(cons(y, u)) }
    [4] P(cons(x, cons(y, u))) if
        { (x < y),
          P(cons(y, u)) }
    [5] P(cons(x, cons(y, u))) if  { (y < x) }
```

Das zweite Induktions-Schema brücksichtigt die Fälle `x = y, x < y, y < x`. Dies führt aber zu fünf Beweispunkten. Die erste Definition liefert nur drei Beweispunkte, dafür ist Beweispunkt `[3]` schwer nachzuweisen, da in diesem die einzelnen Fälle `x = y, x < y` zusammengefaßt und daher einzeln nicht zugänglich sind.

# Anhang A

# Syntax von ASF

Die hier angegebene Grammatik ist in der von ScaPa (Scanner- und Parser-Generator) verlangten Grammatik-Beschreibungssprache dargestellt. Diese wird in der Dokumentation von ScaPa ausführlich beschrieben. Trotzdem an dieser Stelle dazu einige Bemerkungen :

| | |
|---|---|
| `[ .. ]` | optional |
| `+` | Menge, mindestens ein Element |
| `*` | Menge, eventuell leer |
| `{... ","}` | nicht-leere Menge von Elementen durch "," getrennt |
| `\|` | Trennung von Alternativen |

```
specification-def   ::=  specification
                         ident
                         begin
                         [ exports-def ]
                         [ imports-def ]
                         [ sorts-def ]
                         [ functions-def ]
                         [ variables-def ]
                         [ equations-def ]
                         [ properties-def ]
                         end ;

exports-def         ::=  exports
                         begin
                         [ sorts-def ]
                         [ functions-def ]
                         end ;

imports-def         ::=  imports { spec-expression "," } ;

spec-expression     ::=  spec-ident ;
```

```
sorts-def            ::=  sorts sort-list ;

sort-list            ::=  { sort ”,” } ;

sort                 ::=  ident ;

functions-def        ::=  functions function+ ;

function             ::=  fun-ident-list ”:” input-type ”− >” output-type |
                          operator ”_” ”:” sort ”− >” output-type |
                          ”_” operator ”_” ”:” sort ”#” sort ”− >” output-type ;

fun-ident-list       ::=  { fun-ident ”,” } ;

fun-ident            ::=  ident ;

fun-op-list          ::=  fun-ident [ fun-op-list ] | operator [ fun-op-list ] ;

input-type           ::=  [ product ] ;

output-type          ::=  product ;

product              ::=  { sort ”#” } ;

variables-def        ::=  variables variable-list ;

variable-list        ::=  variable-list-elem+ ;

variable-list-elem   ::=  var-ident-list ”:” ”− >” sort ;

var-ident-list       ::=  { var-ident ”,” } ;

var-ident            ::=  ident ;

equations-def        ::=  equations cond-equation1+ ;

cond-equation1       ::=  [ tagg ] equation1 [ when-condition ] ”;” ;

cond-equation2       ::=  [ tagg ] equation2 [ when-condition ] ;

tagg                 ::=  ”[” ident ”]” ;

when-condition       ::=  when condition ;
```

```
equation1           ::=  term := term |
                         term ”==” term |
                         term ”−− >” term ;

equation2           ::=  term ”==” term
                         term ”−− >” term ;

condition           ::=  { condition-equation ”,” } ;

condition-equation  ::=  term ”==” term ;
term                ::=  [ term_and_operator ] primary ;

term_and_operator   ::=  term operator ;

primary             ::=  ident ”(” [ term-list ] ”)” |
                         tuple |
                         ”(” term ”)” |
                         operator primary |
                         var-ident ;

term-list           ::=  { term ”,” } ;

tuple               ::=  ”<” term ”,” term-list ”>” ;

properties-def      ::=  properties
                         begin
                         [ constructors-def ]
                         [ ac-operators-def ]
                         [ c-operators-def ]
                         [ ordering-def ]
                         [ theorems-def ]
                         end ;

constructors-def    ::=  constructors const-list ;

const-list          ::=  const-list-elem∗ ;

const-list-elem     ::=  fun-ident [ dp-f ] | operator [dp-f ] ;

ac-operators-def    ::=  ac-operators fun-op-list ;

c-operators-def     ::=  c-operators fun-op-list ;
```

```
ordering-def        ::=  ordering [ ord-ident-def ]
                         [ precedence-def ]
                         [ status-def ] ;

ord-ident-def       ::=  lrpo ;

precedence-def      ::=  precedence [ { fun-op-list ”,” } ] ;

status-def          ::=  status single-status-def* ;

single-status-def   ::=  fun-op-list ”:” stat-ident ;

stat-ident          ::=  ”l” | ”r” | ”m” ;

theorems-def        ::=  theorems theorem* ;

theorem             ::=  cond-equation2 [ ekl-flag-list ] ”;” ;

ekl-flag-list       ::=  ”[” flag-list ”]” ;

flag-list           ::=  { flag ”,” } ;

flag                ::=  ”e” | ”o” ;

ident               ::=  char | digit (char | digit | ”-” | ”_”)*

specident           ::=  pfadname | modid

digit               ::=  ”0” | ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”

char                ::=  ”a” | ”b” | ... ”z” | ”A” | ”B” | ... ”Z” |

pfadname            ::=  ([”/”] ident2)+ [”/”]

modid               ::=  ident2[”^”[digit+] [”^”[digit+]]]

ident2              ::=  (char2 | digit)+

char2               ::=  char | ”!” | ”@” | ”&” | ”-” |”_” | ”.” | ”=”

char3               ::=  char2 | ”^” | ”+” | ”*” | ”?” | ” ” | ”\” | ”/” | ”|” | ”‘” | ”<” | ”>”

operator            ::=  char3+
```

# Anhang B

# Implementation

## B.1 Übersetzung von ASF nach RRL

### B.1.1 Datenstrukturen

Die zentrale Datenstruktur bei der Übersetzung von ASF nach RRL ist die Zwischendarstellung, also die vom Parser zurückgelieferte interne Darstellung in Lisp-Notation. Die Zwischendarstellung ist im wesentlichen eine Liste von sechs kleineren Listen, in denen alle Informationen der entsprechenden ASF-Spezifikation enthalten sind. Die Elemente der Listen sind alle von der Form:
(element-identifier herkunft element-typ )

Hier nun die vollständige Beschreibung:

| | | |
|---|---|---|
| Liste | ::= | (sortlist funlist varlist eqnlist proplist deptree) |
| | $\equiv$ | (list sortlist funlist varlist eqnlist deptree) |
| sortlist | ::= | (sortident sortident ...) |
| | $\equiv$ | (list sortident sortident ...) |
| sortident | ::= | (sortname herkunft $\left\{\begin{array}{c}\text{"hidden"}\\ \text{"export"}\end{array}\right\}$ "sort") |
| | $\equiv$ | (list sortname herkunft $\left\{\begin{array}{c}\text{"hidden"}\\ \text{"export"}\end{array}\right\}$ "sort") |
| herkunft | ::= | 'Pfad-tupel' (siehe Index) |
| sortname | ::= | 'string' |

| | | |
|---|---|---|
| funlist | ::= | (funlist-elem funlist-elem ...) |
| | ≡ | (list funlist-elem funlist-elem ...) |
| funlist-elem | ::= | (funspec herkunft $\left\{\begin{array}{c}\text{”hidden”}\\ \text{”export”}\end{array}\right\}$ ”function”) |
| | ≡ | (list funspec herkunft $\left\{\begin{array}{c}\text{”hidden”}\\ \text{”export”}\end{array}\right\}$ ”function”) |
| funspec | ::= | (funid inputlist outputlist) |
| | ≡ | (list funid inputlist outputlist) |
| inputlist | ::= | (sortname sortname .....) |
| | ≡ | (list sortname sortname .....) |
| outputlist | ::= | (sortname sortname .....) |
| | ≡ | (list sortname sortname .....) |
| funident | ::= | (funname . $\left\{\begin{array}{c}\text{”functor”}\\ \text{”infix”}\\ \text{”prefix”}\end{array}\right\}$) |
| | ≡ | (cons funname $\left\{\begin{array}{c}\text{”functor”}\\ \text{”infix”}\\ \text{”prefix”}\end{array}\right\}$) |
| varlist | ::= | (varlist-elem varlist-elem ...) |
| | ≡ | (list varlist-elem varlist-elem ...) |
| varlist-elem | ::= | (varid herkunft $\left\{\begin{array}{c}\text{”hidden”}\\ \text{”export”}\end{array}\right\}$ ”variable”) |
| | ≡ | (list varid herkunft $\left\{\begin{array}{c}\text{”hidden”}\\ \text{”export”}\end{array}\right\}$ ”variable”) |
| varid | ::= | ((varname) () outputlist) |
| | ≡ | (list (list varname) nil outputlist) |

| | | |
|---|---|---|
| eqnlist | ::= | (eqnlist-elem eqnlist-elem ...) |
| | $\equiv$ | (list eqnlist-elem eqnlist-elem ...) |
| eqnlist-elem | ::= | (eqnspec herkunft) |
| | $\equiv$ | (list eqnspec herkunft) |
| eqnspec | ::= | (tag eqnid eqntype) |
| | $\equiv$ | (list tag eqnid eqntype) |
| tag | ::= | 'string' |
| eqntype | ::= | $\left\{ \begin{array}{c} \text{"gl-gl"} \\ \text{"def-gl"} \\ \text{"rgl"} \end{array} \right\}$) |
| eqnid | ::= | (term term term term ......) |
| | $\equiv$ | (list term term term term ......) |

Dabei sind die Terme der Reihe nach aufgelistet wie sie in der Gleichung vorkommen. Das heißt : der erste Term ist die linke Seite von der Folgerung der Gleichung der zweite Term dann die rechte Seite , der dritte Term ist die linke Seite der ersten Bedingung u.s.w..

| | | |
|---|---|---|
| term | ::= | $\left\{ \begin{array}{c} \text{varname} \\ \text{funident} \\ \text{bigterm} \end{array} \right\}$ |
| bigterm | ::= | ($\left\{ \begin{array}{c} \text{"\#\textbackslash <"} \\ \text{funident} \end{array} \right\}$ term term term ...) |
| | $\equiv$ | (list $\left\{ \begin{array}{c} \text{"\#\textbackslash <"} \\ \text{funident} \end{array} \right\}$ term term term ...) |
| proplist | ::= | (construclist aclist clist orderlist theoremlist) |
| | $\equiv$ | (list aclist clist orderlist theoremlist) |

| | | |
|---|---|---|
| construclist | ::= | ((constid-list herkunft) (constid-list herkunft) ...) |
| | ≡ | (list (list constid-list herkunft) (list constid-list herkunft) ...) |
| constid-list | ::= | (constid constid ...) |
| | ≡ | (list constid constid ...) |
| constid | ::= | (constname $\left\{ \begin{array}{c} \text{nil} \\ \text{"frei"} \end{array} \right\}$) |
| | ≡ | (list constname $\left\{ \begin{array}{c} \text{nil} \\ \text{"frei"} \end{array} \right\}$) |
| constname | ::= | ident \| operator |
| aclist | ::= | ((operlist herkunft) (operlist herkunft) ...) |
| | ≡ | (list (list operlist herkunft) (list operlist herkunft) ...) |
| clist | ::= | ((operlist herkunft) (operlist herkunft) ...) |
| | ≡ | (list (list operlist herkunft) (list operlist herkunft) ...) |
| operlist | ::= | (opername opername ...) |
| | ≡ | (list opername opername ...) |
| orderlist | ::= | (($\left\{ \begin{array}{c} \text{nil} \\ \text{"lrpo"} \end{array} \right\}$ precdef statdef ) herkunft) |
| | ≡ | (list (list ($\left\{ \begin{array}{c} \text{nil} \\ \text{"lrpo"} \end{array} \right\}$ precdef statdef ) herkunft) |
| precdef | ::= | (operlist operlist ...) |
| | ≡ | (list operlist operlist ...) |
| statdef | ::= | ((operlist $\left\{ \begin{array}{c} \text{"l"} \\ \text{"m"} \\ \text{"r"} \end{array} \right\}$ ) (operlist $\left\{ \begin{array}{c} \text{"l"} \\ \text{"m"} \\ \text{"r"} \end{array} \right\}$ ) ... ) |
| | ≡ | (list (list operlist $\left\{ \begin{array}{c} \text{"l"} \\ \text{"m"} \\ \text{"r"} \end{array} \right\}$ ) (list operlist $\left\{ \begin{array}{c} \text{"l"} \\ \text{"m"} \\ \text{"r"} \end{array} \right\}$ ) ... ) |

| | | |
|---|---|---|
| theoremlist | ::= | (theorem-listelem theorem-listelem ...) |
| | ≡ | (list theorem-listelem theorem-listelem ...) |
| theorem-listelem | ::= | (theorem-def herkunft) |
| | ≡ | (list theorem-def herkunft) |
| theorem-def | ::= | ( theorem flaglist) |
| | ≡ | (list tag theorem flaglist) |
| theorem | ::= | (tag theoremid theoremtype) |
| | ≡ | (list tag eqnid theoremtype) |
| tag | ::= | 'string' |
| theoremtype | ::= | $\left\{ \begin{array}{c} \text{"gl-gl"} \\ \text{"rgl"} \end{array} \right\}$ |
| flaglist | ::= | $(\left\{ \begin{array}{c} \text{"e"} \\ \text{"o"} \\ \text{"h"} \end{array} \right\} \left\{ \begin{array}{c} \text{"e"} \\ \text{"o"} \\ \text{"h"} \end{array} \right\} \ldots )$ |
| | ≡ | $(\text{list} \left\{ \begin{array}{c} \text{"e"} \\ \text{"o"} \\ \text{"h"} \end{array} \right\} \left\{ \begin{array}{c} \text{"e"} \\ \text{"o"} \\ \text{"h"} \end{array} \right\} \ldots )$ |

### Parser für die ASF-Spezifikation

Der Parser für ASF wird von dem Parser-Generator ScaPa erzeugt (jeweils zu Beginn von ART). ScaPa nimmt dabei eine Beschreibung der ASF-Grammatik als Eingabe und erzeugt daraus einen Parser. Ebenso verhält es sich mit dem Scanner, welcher aber lediglich einmal generiert wird. Den Kern des Parsers stellen die semantischen Aktionen dar, in welchen verankert ist, wie die geparsten Informationen in Lisp weiterverarbeitet werden. Die Funktion, welche den Parser anstößt, heißt *parse* ; als Argument kann man wahlweise

- einen String
- einen Pfadnamen
- ein Pfad-Tupel

angeben. Hierbei ist zu beachten, daß, falls das Argument ein Pfadname ist, wirklich nur das durch den Pfadnamen festgelegte File geparst wird. Ist das Argument ein String oder ein Pfad-Tupel, wird auf die File-Struktur Bezug genommen, d.h. hier wird eine gesamte Spezifikation geparst. Voraussetzung ist, daß das oberste Verzeichnis der File-Struktur bekannt ist, d.h. die Variable *default-baum-dir* muß entsprechend gesetzt sein. Der String muß sowohl den Namen der Spezifikation, als auch die Implementations-Nummer und Eigenschaften-Tiefe beinhalten, d.h.

```
<name>^[<number>]^[<number>]
```

**Beispiel 1.1.1:**
`nat^1^1` bezeichnet die ASF-Spezifikation `nat.impl_1.prop_1.asf` ,und `nat^^` bezeichnet die ASF-Spezifikation `nat.asf` .

Ebenso verhält es sich mit dem Pfad-Tupel, welcher aus vier Teilen besteht :

- Name der Spezifikation
- Implementation
- Eigenschaft
- Pfad

**Beispiel 1.1.2:**
`("nat" 1 1 "#P"<*default-baum-dir*>/nat/nat.impl_1/nat.impl_1.prop_1.asf")`
ist dasselbe wie `nat^1^1`.

Ist das Argument ein String oder Pfad-Tupel, werden zu Beginn alle abhängigen Files ermittelt und geparst, dabei werden Mehrfach-Importe (ein Modul wird über verschiedene Wege importiert) ermittelt und mehrfaches Parsen vermieden. Für jedes ASF-File wird beim Parsen ein Intern-File angelegt, falls dieses nicht existiert. Ist hingegen ein Intern-File schon vorhanden, wird dessen Aktualität geprüft, und dieses je nach dem geladen oder neu angelegt. Auf diesem Weg wird das Parsen erheblich optimiert, da das Laden des Intern-Files wesentlich schneller ist als das Parsen des entsprechenden ASF-Files. Fie Funktion *parse* liefert die Zwischendarstellung der ASF-Spezifikation zurück, dies ist die flache Darstellung der Spezifikation, in welcher noch Namenskonflikte ermittelt und aufgelöst werden müssen.

### B.1.2 Auflösen von Namenskonflikten

**Namenskonflikte in ASF**

Die Funktion *renaming* untersucht als erstes die Zwischendarstellung der ASF-Spezifikation auf Namenskonflikte. Ein Namenskonflikt besteht dann, wenn mehrere Funktionen unter dem gleichen Namen in verschiedenen Modulen deklariert wurden. Unterschieden wird zwischen auflösbaren und nicht auflösbaren Namenskonflikten. Auflösbar ist ein Konflikt, wenn die entsprechenden Funktionen über unterschiedliche Definitionsbereiche verfügen.

Auch das Überladen von null-stelligen Funktionen führt zu einem nicht-auflösbaren Namenskonflikt. Befinden sich in der Zwischendarstellung nur auflösbare Konflikte, können die Terme in den Gleichungen und Theoremen einer Umbenennung unterzogen werden. Wichtig ist, daß in allen Termen entschieden werden kann, welche Funktionen gemeint sind.

**Beispiel 1.2.3:**

Gegeben seien

```
c : -> S1
f : S1 -> S2
f : S2 -> S3
```

Der Term `f(c)` enthält das Funktionssymbol `f : S1 -> S2`. Sei zusätzlich noch `c : -> S2` (also ein nicht-auflösbarer Konflikt) gegeben, dann ist die Sorte des Terms `f(c)` nicht mehr eindeutig.

Indem die Sorten-Informationen von innnen (also von Konstanten und Variablen) nach außen getragen werden, kann immer entschieden werden, welche Funktion gemeint ist, falls alle Konflikte auflösbar sind.

**Namenskonflikte mit RRL**

Die Funktion *compile-asf* erzeugt zwei RRL-Files. Aus diesem Grund muß auch die Eingabe-Syntax von RRL beachtet werden. Bestimmte Funktionsnamen werden von RRL nicht akzeptiert, diese lauten

```
and, or, xor, equ, not, false, true,
nil, all, exist , eq, cond, if, =
```

Diese Funktionsnamen, welche alle in der Variablen **builtinexport** enthalten sind, müssen also umbenannt werden.
Desweiteren hat der Benutzer die Möglichkeit in der ASF-Spezifikation die built-ins von RRL zu benutzen, also gerade folgende Funktionen

```
  and, or, xor, equ, not, false, true
```

Diese Funktionen, welche alle in der Variablen **builtinhidden** enthalten sind, werden in der ASF-Spezifikation nicht deklariert. Will der Benutzer hingegen zum Beispiel sein eigenes `and` spezifizieren, wird im Laufe der Übersetzung von ASF nach RRL dieses `and` umbenannt [1].

In RRL ist eine Variable ein String aus alphanumerischen Zeichen, welcher mit einem Buchstaben aus { u, v, w, x, y, z } beginnen muß. Variablen aus der ASF-Spezifikation, die nicht mit einem Buchstaben aus { u, v, w, x, y, z } beginnen, werden durch Voranstellen des Buchstabens $x$ und eventuell einer Zahl umbenannt. Sämtliche Umbenennungen von Funktionen und Variablen werden in einer Assoziationsliste festgehalten, welche mit der umbenannten Zwischendarstellung zurückgeliefert wird.

**Beispiel 1.2.4:**

```
specification M1
begin
  exports
    begin
      sorts S1
      functions
        f : S1 -> S1
        g : S1 -> S1
    end
end

specification M2
begin
  exports
    begin
      sorts S2
      functions
        f : S1 -> S2
        g : S1 # S1 -> S1
        exist : -> S2
    end
  imports M1
end
```

[1]Um einen Konflikt mit dem in RRL vorhandenen `and` zu vermeiden.

Das Modul **M2** importiert das Modul **M1**. Das Modul **M1** enthält in seinem export-Teil die Funktionsnamen $f$ und $g$. Da das Modul **M2** das Modul **M1** importiert, entstehen nun folgenden Namenskonflikte

- $f$ : S1 -> S2 kollidiert mit $f$ : S1 -> S1
- $g$ : S1 # S1 -> S1 kollidiert mit $g$ : S1 -> S1
- *exist* kollidiert mit der RRL-Funktion *exist*

Der erste Namenskonflikt ist nicht auflösbar[2], da die Funktionen mit Namen $f$ über dem gleichen Definitionsbereich deklariert sind.
Der zweite Namenskonflikt wird aufgelöst indem z.B. das $g$ aus dem Module **M1** umbenannt wird in *g.S*. Falls auch dieser Name nicht eindeutig ist so wird *g.S1* oder *g.S1S* oder *g.S1S1* ... gewählt.
Der dritte Namenskonflikt wird aufgelöst, indem *exist* entsprechend umbenannt wird.

### B.1.3 Semantische Überprüfungen

Nach der Umbenennung werden semantische Überprüfungen vorgenommen. Für Gleichungen und Theoreme wird folgendes untersucht (dabei sei f eine Funktion, welche in einer Gleichung oder einem Theorem der Spezifikation SPEC vorkommt)

- Ist f vom Benutzer definiert oder ist es eine von RRL definierte Funktion ?
  Trifft beides zu, kommt es zur Umbenennung von f.
- Ist f in SPEC sichtbar, d.h. ist sie im obersten ASF-Modul definiert oder wird sie von einem der importierten ASF-Module exportiert ?
- Für jeden Term wird überprüft, ob er wohlgeformt ist.

Die anderen Properties (Präzedenz, Status, ...) werden analog zu den zwei oberen Punkten überprüft.

### B.1.4 Ausgabe

Nachdem die Namenskonflikte durch Umbenennung aufgelöst worden sind, kann die Zwischendarstellung zur Erzeugung der von RRL ausführbaren Files herangezogen werden. Dies wird von der Funktion *create-rrl-files* ausgeführt. Da zwei Beweismethoden von RRL in ART unterstützt werden, werden auch zwei Files erzeugt (jede Methode verlangt ihr eigenes Vorgehen). Das zur *inductionless induction using test sets* gehörende File (kb-file) hat folgende Struktur :

[2] Der Benutzer wird hiervon durch eine Meldung in Kenntnis gesetzt.

| |
|---|
| Signatur |
| Gleichungen |
| Konstruktoren,Präzedenz,Status,<br>C-Operatoren, AC-Operatoren |
| Vervollständigung |
| Beweismethode auf inductionless induction<br>using test sets setzen |
| Beweisaufgaben |

Es wird also von *create-rrl-files* ein File erzeugt, welches folgendes Aussehen hat :

```
;; Initialisierung von RRL
init

;; Undo-Modus wird ausgeschaltet
option
undo
noundo

;; log-file wird angelegt
log
<filename>

;; Eingabe der Signatur und der Gleichungen
ADD
<signature>
<equations>
]

;; Angabe von Konstruktoren
operator
constructor
<function>

;; Angabe von AC-Operatoren
operator
ac-operator
<functions>

;; Angabe von C-Operatoren
operator
commutative
<functions>

;; Angabe der Ordnung
```

```
operator
order
l

;; Angabe der Praezedenz f1 > f2 ... > fn
operator
precedence
f1 f2 ... fn

;; Angabe von Status
operator
status
<function>
<status>

;; Vervollstaendigung
kb

;; Ueberpruefen der vollstaendigen Definiertheit
suffice

;; Beweismethode
option
prove
s

;; Beweisaufgaben
prove
<equation>
.
.
.

;; Ausgabe des aktuellen Zustandes von RRL
write
spec-write
<filename>
```

Das kb-file wird innerhalb der File-Struktur in dem Verzeichnis der ASF-Spezifikation abgelegt und erhält dabei das Suffix ".rrlkb".

**Beispiel 1.4.5:**
Sei `specs` das oberste Verzeichnis der File-Struktur. Das kb-file der Spezifikation `nat^1^1` wird in dem Verzeichnis `specs/nat/nat.impl_1/nat.impl_1.prop_1` unter dem Namen `nat.impl_1.prop_1.rrlkb` abgelegt.

Das zur *cover-set induction* gehörende File (mr-file) hat folgenden Aufbau :

| |
|---|
| Signatur |
| Definitionen |
| Konstruktoren,Präzedenz,Status,<br>C-operatoren, AC-Operatoren |
| Orientierung |
| Beweismethode auf cover-set induction<br>setzen |
| Ermittlung der cover-sets |
| Eigenschaften |
| Orientierung |
| Beweisaufgaben |

Es wird also von *create-rrl-files* ein File erzeugt, welches folgendes Aussehen hat :

```
;; Initialisierung von RRL
init

;; log-file wird angelegt
log
<filename>

;; Undo-Modus wird ausgeschaltet
option
undo
noundo

;; Eingabe der Signatur und der Definitionen
ADD
<signature>
<equations>
]

;; Angabe von Konstruktoren
operator
constructor
<function>

;; Angabe von AC-Operatoren
operator
ac-operator
<functions>
```

```
;; Angabe von C-Operatoren
operator
commutative
<functions>

;; Angabe der Ordnung
operator
order
l

;; Angabe der Praezedenz f1 > f2 ... > fn
operator
precedence
f1 f2 ... fn

;; Angabe von Status
operator
status
<function>
<status>

;; Orientierung der Definitionen
makerule

;; Beweismethode
option
prove
e

;; Ermitteln der cover-sets
cover

;; Eigenschaften
add
<equations>
]

;; Orientierung der Eigenschaften
makerule

;; Beweisaufgaben
prove
<equation>
.
```

.
.

```
;; Ausgabe des aktuellen Zustandes von RRL
write
spec-write
<filename>
```

Das mr-file erhält den Suffix "rrlmr" und wird an der gleichen Stelle wie das kb-file innerhalb der File-Struktur abgelegt.

### B.1.5 Hauptfunktion

Die Funktion *compile-asf* ruft die schon beschriebenen Funktionen *parse, renaming, create-rrl-files* in dieser Reihenfolge auf, d.h. als erstes wird die ASF-Spezifikation geparst, dann werden die Namenskonflikte aufgelöst, die Zwischendarstellung semantischen Überprüfungen unterzogen und schließlich die beiden RRL-Files erzeugt. Das Argument von *compile-asf* stimmt mit dem von *parse* überein.

## B.2 Rückübersetzung von RRL nach ASF

### B.2.1 Variablen

Im *package minsky* sind folgende Variablen deklariert :

- *rrl-log-parse-table*
- *rrl-spec-parse-table*
- *asf-spec*
- *log-spec*
- *diff-spec*
- *diff-asf-spec*

Die Variable **rrl-log-parse-table** wird von dem Parser für das Parsen des *log-files* benötigt. In ihr sind zu erkennende Befehls-Sequenzen und die damit assoziierten Verarbeitungsfunktionen enthalten, d.h. wird eine Befehl-Sequenz vom Parser erkannt, wird die damit assoziierte Funktion angestoßen. Sie ist dabei ein Repräsentant der Datenstruktur LOG-PARSE-TABLE (siehe B.2.2) und wird zu Beginn wie folgt aufgebaut :

| **Befehls-Sequenz** | **Verarbeitungsfunktion** |
|---|---|
| add | logcmd-add |
| akb | logcmd-akb |
| delete | logcmd-delete |
| init | logcmd-init |
| load | logcmd-load |
| log | logcmd-log |
| equivalence | logcmd-equiv |
| status | logcmd-oper-status |
| operator constructor | logcmd-oper-constr |
| operator commutative | logcmd-oper-c |
| operator acoperator | logcmd-oper-ac |
| operator precedence | logcmd-oper-p |
| operator status | logcmd-oper-status |
| option | logcmd-option |
| prove | logcmd-prove |
| read | logcmd-read |
| save | logcmd-save |
| suffice | logcmd-suffice |
| write | logcmd-write |

Die Variable **rrl-spec-parse-table** wird von dem Parser für das Parsen des *spec-files* benötigt. In ihr sind alle möglichen Meldungen und die damit assoziierten Verarbeitungsfunktionen enthalten, d.h. erkennt der Parser eine Meldung, wird die damit assoziierte Funktion angestoßen.
Zu Beginn wird sie wie folgt aufgebaut :

| **Meldung** | **Verarbeitungsfunktion** |
|---|---|
| The arities of the operators are | read-arities |
| The system has the following constructors: | read-constructors |
| Equivalence relation among operators | read-equivalence |
| Precedence relation | read-precedence |
| Operators with status | read-status |
| Associative & commutative operator set | read-ac-set |
| Commutative operator set | read-c-set |
| Transitive operator set | nil |
| There are no equivalent operators. | nil |
| There is no ordering yet in the precedence relation. | nil |
| There are no operators with status. | nil |

Die restlichen Variablen sind Repräsentanten der Datenstruktur SPEC. In der Variablen **asf-spec** werden alle Informationen aus der ASF-Spezifikation eingetragen, in der Variablen **log-spec** werden die Informationen aus dem log-file bzw. spec-file eingetragen. In die restlichen Variablen werden die Unterschiede zwischen ASF und RRL eingetragen, d.h. **diff-spec** erhält alle Informationen, welche in RRL aber nicht in

ASF und **diff-asf-spec** alle Informationen, welche in ASF aber nicht in RRL gefunden wurden.

### B.2.2 Datenstrukturen

Folgende Datenstrukturen finden bei der Rückübersetzung ihre Anwendung

- LOG-PARSE-TABLE
- SPEC

**LOG-PARSE-TABLE**
Die Datenstruktur LOG-PARSE-TABLE stellt einen TRIE dar, in welchem Befehl-Sequenzen und die damit assoziierten Verarbeitungsfunktionen eingetragen werden können. Da lediglich die Variable *rrl-log-parse-table* dieser Datenstruktur angehört, beziehen sich folgende Funktionen auf diese Variable :

- make-log-table
- insert-log-table

Erzeugt wird die *rrl-log-parse-table* durch die Funktion
*make-log-table.*
Einfügen einer Befehl-Sequenz mit zugehöriger Funktion geschieht durch die Funktion
*insert-log-table <command> ... <command> <function>*
wobei *command* ein String und *function* ein funktionales Objekt ist.

**Beispiel 2.2.1:**
```
(make-log-table)
(insert-log-table "operator" "constructor" #'logcmd-oper-constr)
(insert-log-table "operator" "commutative" #'logcmd-oper-c)
(insert-log-table "option"                 #'logcmd-option)
```

Die erste Anweisung bewirkt das Erzeugen der leeren *rrl-log-parse-table*, die weiteren Anweisungen erzeugen den in Abbildung B.1 dargestellten TRIE

**SPEC**
Die Datenstruktur SPEC hat die Aufgabe, die in den entsprechenden files (asf-file, log-file, spec- file) anfallenden Informationen verwalten zu können. Dabei sind folgende Informationen von Interesse :

- sorts
- functions

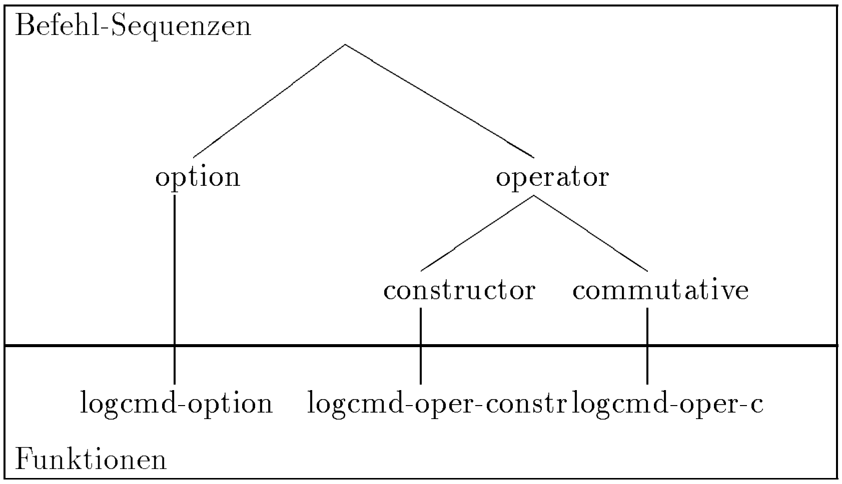



Abbildung B.1: Trie

- variables
- equations
- constructors
- ac-operators
- c-operators
- status
- precedence
- theorems
- sufficient complete

Da der Zugriff auf Variablen vom Typ SPEC ausschließlich über Funktionen erfolgen soll, sind Funktionen zum Erzeugen der Datenstruktur, zum Einfügen, Löschen und Lesen von Daten aus der Datenstruktur notwendig. Zu diesem Zweck wurden folgende Funktionen vorgesehen :

| **function** | **effect** |
|---|---|
| make-spec | erzeugt eine leere SPEC |
| insert-spec | fügt Informationen in einen Repräsentanten des Datentyps SPEC ein |
| get-spec | liefert alle Einträge eines Slots wieder |
| get-entry | liefert den Eintrag für ein Symbol zurück |
| delete-spec | löscht alle Einträge in einem Slot |
| delete-entry | löscht den Eintrag für ein Symbol |
| print-spec | Ausgabe der SPEC auf dem Bildschirm |

Dabei sind folgende Slots vorgesehen : sort, function, variable, equation, contructor, ac-operator, c-operator, status, precedence, prove, suffice.

**Beispiel 2.2.2:** Erzeugen, Einfügen und Löschen

```
> (setq *log-spec* (make-spec))
((NIL) (NIL) (NIL) NIL NIL (NIL) (NIL) (NIL)
 (NIL) (NIL) (NIL) NIL (NIL))

> (insert-spec *log-spec* 'function
'(("cons" ("NAT" "NAT_LIST") ("NAT_LIST") "functor")))

((NIL) ((("cons" ("NAT" "NAT_LIST") ("NAT_LIST") "functor")))
(NIL) NIL NIL (NIL) (NIL) (NIL) (NIL) (NIL) (NIL) NIL (NIL))

> (insert-spec *log-spec* 'function
'(("succ" ("NAT") ("NAT") "functor")))

((NIL) ((("succ" ("NAT") ("NAT") "functor")
("cons" ("NAT" "NAT_LIST") ("NAT_LIST") "functor")))
(NIL) NIL NIL (NIL) (NIL) (NIL) (NIL) (NIL) (NIL) NIL (NIL))

> (print-spec *log-spec*)

functions :
   succ : NAT -> NAT
   cons : NAT NAT_LIST -> NAT_LIST

NIL

> (get-spec *log-spec* 'function)

(("succ" ("NAT") ("NAT") "functor")
("cons" ("NAT" "NAT_LIST") ("NAT_LIST") "functor"))

> (get-entry *log-spec* 'function "succ")

("succ" ("NAT") ("NAT") "functor")

> (delete-spec *log-spec* 'function)

(NIL)

>  (get-spec *log-spec* 'function)
```

```
NIL
```

### B.2.3 Parser für das *spec-file*

Die Funktion *spec-parse* hat die Aufgabe, die Informationen aus dem *spec-file* zu parsen, und sie in die globale Varibale **log-spec** einzutragen.
In der Variable **rrl-spec-parse-table** (siehe B.2.1) befinden sich die möglichen Meldungen aus dem *spec-file*, die also anzeigen, was der darunterliegende Block für Informationen enthält, und damit assoziiert ein Funktionsname, nämlich gerade den der Funktion, welche das Einlesen dieser Informationen zur Aufgabe hat.
Die Funktion *spec-parser* versucht nun diese Meldungen im *spec-file* zu finden. Ist eine solche Meldung gefunden, wird die damit assoziierte Funktion aufgerufen, welche dann die entsprechenden Informationen einliest und sie in die Variable **log-spec** einträgt.
Dabei werden folgende Verarbeitungsfunktionen benutzt :

- read-arities
- read-constructors
- read-equivalence
- read-precedence
- read-status
- read-ac-set
- read-c-set

**read-arities** : Diese Funktion liest die nach der entsprechenden Meldung im *spec-file* erscheinenden Funktionen und die in den Funktionsdeklarationen benutzten Sorten ein und legt sie in **log-spec** unter *'function* bzw. unter *'sort* ab.

**read-constructors** : Diese Funktion liest die im *spec-file* nachfolgenden Konstruktoren ein und trägt diese in die **log-spec** unter *'constructor* ein.
Dabei ist zu beachten, daß im *spec-file* nicht angegeben wird, ob es sich um freie oder nicht-freie Konstruktoren handelt. Dies wird von der Funktion *log-parser* ermittelt (durch die Funktion *logcmd-oper-constr*), welche das Parsen des *log-files* zur Aufgabe hat.

**read-equivalence** : Diese Funktion liest die im *spec-file* als äquivalent deklarierten Operatoren ein und fügt diese in die **log-spec** unter *'equivalence* ein.

**read-precedence** : Diese Funktion liest die im *spec-file* nachfolgenden Operatoren ein und fügt diese in die *log-spec* unter *'precedence* ein.

**read-status** : Diese Funktion liest die im *spec-file* nachfolgenden Operatoren mit ihrem Status ein und fügt diese Information in die **log-spec** unter *'status* ein.

**read-ac-set** : Diese Funktion liest die im *spec-file* nachfolgenden Operatoren ein und fügt diese in die **log-spec** unter *'ac-operator* ein.

**read-c-set** : Diese Funktion liest die im *spec-file* nachfolgenden Operatoren ein und fügt diese in die **log-spec** unter *'c-operator* ein.

### B.2.4 Parser für das *log-file*

Das Parsen des *log-files* wird von der Funktion *log-parser* durchgeführt. Die Funktion *log-parser* bekommt als Eingabe das *log-file* in Listen-Form, d.h. als eine Liste von Strings. Genauer gesagt bekommt sie als Eingabe den letzten init-Block, d.h. den Inhalt des *log-files* ab dem letzten init-Befehl .
Die Aufgabe dieser Funktion ist das Erkennen von Befehlssequenzen in dem *log-file* und das entsprechende Eintragen der eingelesenen Informationen in die Variable **log-spec**.
Die Funktion *log-parser* versucht eine Befehlssequenz zu erkennen, indem sie die eingelesenen Daten mit den Einträgen aus der Variablen *rrl-log-parse-table* vergleicht, und dann, falls dies möglich ist, die Funktion anwendet, welche in der Variablen **rrl-log-parse-table** mit der Befehlssequenz assoziiert ist.

**Beispiel 2.4.3:**

```
operator
constructor
succ
y
```

Die Funktion *log-parser* erkennt die Befehlssequenz *operator constructor* und wendet die Funktion *logcmd-oper-constr* an, welche nun *succ* als freien Konstruktor in die **log-spec** einträgt ("y" steht für 'yes' und bedeutet freier Konstruktor).

Die funktionalen Objekte, welche gerade die Blätter des **rrl-log-parse-table** sind, haben also die Aufgabe, die durch die entsprechende Befehlssequenz festgelegten Informationen aus dem *log-file* herauszulesen und sie in die **log-spec** einzutragen.
Dabei sind folgende Funktionen in der Variablen **rrl-log-parse-table** enthalten :

**logcmd-add** : Diese Funktion liest solange ein, bis sie eine Zeile "]" oder das Textende erreicht. Die in diesem Textblock befindlichen Funktionsdeklarationen und Gleichungen werden herausgelesen und in die **log-spec** unter *'function* bzw. unter *'add-equation* eingetragen.

**logcmd-delete** : Diese Funktion uberliest alle Zeilen *help*.

**logcmd-load** : Diese Funktion überliest den entsprechenden Textblock.

**logcmd-log** : Diese Funktion überliest eine Zeile.

**logcmd-equiv** : Diese Funktion überliest eine Zeile.

**logcmd-oper-constr** : Diese Funktion liest als erstes die Liste der Konstruktoren ein und überprüft für jeden Konstruktor, ob dieser als freier oder nicht-freier Konstruktor angegeben wurde. Entsprechend wird dann jeder Konstruktor unter *'constructor* in die **log-spec** eingetragen. Dabei ist zu beachten, daß Konstanten (null-stellige Funktionen) von RRL stets als freie Konstruktoren betrachtet werden, hier wird von RRL also nicht nachgefragt.
In RRL können mit der Befehlssequenz *operator constructor* neue Funktionen deklariert werden. Dies führt zu einer Fehlermeldung, da vorausgesetzt wird, daß neue Funktionen mit dem Befehl *add* deklariert werden.

**logcmd-oper-c** : Diese Funktion liest die Liste der Operatoren ein und fügt sie unter *'c-operator* in die **log-spec** ein.

**logcmd-oper-ac** : Diese Funktion liest die Liste der Operatoren ein und fügt sie unter *'ac-operator* in die **log-spec** ein.

**logcmd-oper-p** : Diese Funktion liest die Liste der Operatoren $< f_1 >$ ,$< f_2 >$ ... $< f_n >$ ein ein und fügt jedes Tupel $< f_i >$, $< f_{i+1} >$ i=1,...,n-1 unter *'precedence* in die **log-spec** ein.

**logcmd-option** : Diese Funktion überliest alle Zeilen *help*. Wird eine Zeile *support* gelesen, dann wird solange weitergelesen, bis eine Zeile l erscheint. Wird *prove* gelesen, dann wird eine weitere Zeile eingelesen.

**logcmd-prove** : Diese Funktion ermittelt als erstes die Zeile mit dem Theorem. Es gibt dafür folgende Möglichkeiten: Im *log-file* steht

- ```
  prove
  y
  ```
  Dies bedeutet, daß schon vorher versucht wurde (erfolglos) ein Theorem zu beweisen. RRL fragt : *Is it ok to continue ?*. Diese Frage wurde mit 'y' beantwortet.
  ```

In diesem Fall wird der ganze prove-Bock überlesen, d.h. es wird solange weitergelesen, bis eine Zeile mit ";" und *theorem* oder eine Zeile *quit* erreicht wird.
Im ersten Fall wurde etwas nachgewiesen, im zweiten wurde die kb-Bearbeitung durch RRL (bedingt durch die Beweis-Methode) mit *quit* verlassen. Dieser Fall wird also von der Funktion nicht bearbeitet.

- ```
  prove
  n
  <theorem>
  ```
  Obige Frage wird mit n(o) beantwortet. Das Theorem befindet sich eine Zeile weiter, und wird eingelesen.

- ```
  prove
  ]
  ```
  Die prove-Bearbeitung wurde direkt verlassen.

- ```
  prove
  <theorem>
  ```
  Das Theorem wird eingelesen.

Anschliessende ermittelt die Funktion *log-cmd-prove* den Ausgang des Beweises, d.h. es muß entschieden werden, ob es sich um ein *equational theorem*, *inductive theorem* oder keines von beiden handelt.
Dabei ist folgendes zu beachten : Ist ein Theorem als *equational theorem*, als *inductive theorem* oder als *not inductive theorem* nachgewiesen worden, ist der Beweis beendet. Wurde nun aber für das Theorem *not equational* nachgewiesen, kann es aber noch ein *inductive theorem* sein.
In diesem Fall fragt RRL, ob dies überprüft werden soll.
Auch kann man bedingt durch die Beweis-Methode in eine kb-Bearbeitung gelangen, welche mit *quit* verlassen werden kann. Ist dies nun geschehen, ist auch die prove-Bearbeitung beendet und beim nachsten *prove* gerät man in die erste der oben angegebenen Möglichkeiten. Zu beachten ist, daß bei der Methode *cover-set induction* Lemmata entstehen können. Diese müssen ebenfalls erkannt und in die *log-spec* eingetragen werden.

### B.2.5 Parser für die ASF-Spezifikation

Um die ursprüngliche ASF-Spezifikation mit dem RRL-Protokoll zu vergleichen, muß diese geparst werden. dabei werden die Funktionen *parse* und *renaming* verwendet. Liegt die ASF-Spezifikation in ihrer Zwischendarstellung vor, werden die Informationen der Zwischendarstellung in die Variable *\*asf-spec\** eingetragen.
Der Grund, warum hier die ASF-Spezifikation geparst wird, anstatt eins der von *compile-asf* erzeugten RRL-Files einzulesen, liegt in der Umbenennung von Variablen und Funktionen begründet. Diese Informationen, nämlich welche Variablen oder welche Funktionen umbenannt wurden und wie deren ursprüngliche Namen waren, sind
  ```

in den RRL-Files nicht verfügbar. Bei der Rückübersetzung von RRL nach ASF muß diese Umbenennung rückgängig gemacht werden. Zu beachten ist auch, daß in RRL neue Variablen entstehen können (z.B. durch Generalisierung), welche konfliktfrei in der neuen ASF-Spezifikation angelegt werden müssen.

### B.2.6 Überprüfen der Konsistenz

Die Funktion *make-consistent* hat die Aufgabe die Inhalte der Variablen **asf-spec** und **log-spec** auf Konsistenz zu überprüfen. In der Variablen **asf-spec** befinden sich die Informationen aus der ASF-Spezifikation, und in der Variablen **log-spec** die Informationen aus dem *log-file* bzw. *spec-file*. Diese Informationen können sich aus folgenden Gründen in einem inkonsistenten Zustand befinden

- Funktionen, welche in der **log-spec** infix bzw. präfix eingetragen sind, können in der **asf-spec** präfix bzw. infix eingetragen sein. Benutzt man zum Beispiel in der ASF-Spezifikation die built-ins *boolean* von RRL, werden die built-ins (etwa *and, or, xor, ...*) in ASF präfix, aber in RRL infix geschrieben. Wird ein solcher Präfix-Infix-Unterschied ermittelt, wird die **log-spec** an die **asf-spec** angeglichen.

- Hat man in der ASF-Spezifikation die built-ins *boolean* von RRL verwendet, muß die Sorte *bool* in die **asf-spec** nachgetragen werden.

- Sowohl die ASF-Präzedenz wie auch die RRL-Präzedenz muß auf Konsistenz überprüft werden. Jede Präzedenz läßt sich als gerichteter Graph repräsentieren, und sie befindet sich in einem konsistenten Zustand, wenn dieser Graph azyklisch ist.

- In der **log-spec** können nicht-deklarierte Funktionen als constructor, c-operator, ac-operator, mit Status und in der Präzedenz angegeben sein. Diese müssen nun aus der **log-spec** gelöscht werden.

### B.2.7 Ermitteln der Unterschiede

Die Funktion *compute-difference* bekommt als Eingabe die Variablen **log-spec** und **asf-spec** und berechnet nun die Differenz zwischen diesen beiden. Einträge, welche in der **log-spec** vorhanden sind, aber nicht in der **asf-spec**, werden in die Variable **diff-spec**, Einträge, welche in der **asf-spec** vorhanden sind, aber nicht in der **log-spec**, in die Variable **diff-asf-spec** eingetragen.
Unterschieden wird zwischen den Gleichungen, welche durch den Befehl *add* hinzugekommen sind (add-equations) und den Gleichungen, welche durch die Befehlssequenz *write spec-write* ausgegeben wurden (spec-equations). Dabei ist zu beachten, daß durch die Knuth-Bendix Vervollständigung neue Gleichungen hinzukommen , aber auch Gleichungen, welche der Benutzer mit *add* eingeführt hat, gelöscht werden können. Hier muß der Benutzer entscheiden, welche Gleichungen er in der Differenz behalten möchte.

Wird eine zu exportierende Beweisaufgabe bewiesen, wird das entsprechende Theorem mit den Flags o,e in die Differenz aufgenommen, ansonsten nur mit dem Flag o. Beweisaufgaben, welche in RRL nicht bewiesen werden konnten, werden wieder als Beweisaufgaben in die Differenz aufgenommen. Damit wird erreicht, daß Beweisaufgaben innerhalb der File-Struktur 'mit nach unten wandern'. Alle restlichen Gleichungen (also Theoreme oder nicht bewiesene Gleichungen) , welche in der Variablen **log-spec** im slot 'prove gefunden werden, sind also keine Beweisaufgaben der Spezifikation , sondern vom Benutzer eingegebene Beweisaufgaben oder Lemmata, welche bei einem grösseren Beweis angefallen sind. Diese Gleichungen können entweder gesamt oder selektiv übernommen werden.

### B.2.8 Rückumbenennung

Da die ermittelten Informationen in der File-Struktur abgelegt werden, muß die Umbenennung rückgängig gemacht werden. Hierfür wird die von der Funktion *renaming* mitgelieferte Umbenennungsliste gebraucht, aus welcher hervorgeht, welche Funktionen umbenannt wurden und wie diese ursprünglich hießen. Die Funktion *benenne-um* macht die Umbenennung rückgängig. Zu beachten ist, daß in RRL neue Variablen entstehen können (z.B. Generalisierung), welche konfliktfrei in der neuen ASF-Spezifikation abgelegt werden müssen. Vor der Rückumbenennung müssen also die neuen Variablen ermittelt und gegebenenfalls so umbenannt werden, daß sie in der neuen ASF-Spezifikation keinen Konflikt verursachen. Die Funktion *find-variables* bringt durch eventuelle Umbenennung alle Variablen in einen konsistenten Zustand.

### B.2.9 Ausgabe

Die Funktion *print-asf-file* hat die Aufgabe, die ermittelte Differenz, welche sich in *diff-spec* befindet, gemäß ASF-Syntax in ein File auszugegeben. Hier werden die Regeln der File-Struktur beachtet, d.h. es wird entweder eine neue Implementation oder eine tiefere Eigenschaft erzeugt.

### B.2.10 Hauptfunktion

Die Hauptfunktion *save-rrl* wendet die zuvor beschriebenen Funktionen an. Zuerst wird das spec-file, dann das log-file und schließlich die ASF-Spezifikation geparst. Die eingelesenen Informationen werden in einen konsistenten Zustand gebracht, dann werden die Unterschiede ermittelt. Ist dies geschehen, kann die Umbenennung rückgängig gemacht und das neue ASF-File in der File-Struktur abgelegt werden. Die Funktion *save-rrl* hat denselben Aufruf-Parameter wie die Funktion *compile-asf*

# Literaturverzeichnis


[Ba87] Bachmair, L. (1987): *Proof methods for equational theories.* Ph.D. Thesis, Dept. of Computer Science, University of Illinois, Urbana.

[BeHeKl89] Bergstra, J.A., Heering, J., Klint, P. (1989): *Algebraic Specification.* Addison Wesley, Workingham (England).

[De82] Dershowitz, N. (1982): *Orderings for term rewriting systems.* Theoretical Computer Science 17, 279-301.

[De87] Dershowitz, N. (1987). *Termination of rewriting.* J. of Symbolic Computation 3, 69-116.

[EhGoLi89] Ehrich, H.-D., Gogolla, M. und Lipeck, U.W. (1989): *Algebraische Spezifikation abstrakter Datentypen. Eine Einführung in die Theorie.* Stuttgart: Teubner, 1989 (Leitfäden und Monographien der Informatik) ISBN 3-519-02266-4

[GoThWa78] Goguen, J.A., Thatcher, J.W. and Wagner, E.W. (1978): *Initial algebra approach to the specification, correctness and implementation of abstract data types.* in: R. Yeh (ed.), Data structuring, Current Trends in Programming Methodology 4, 80-149

[HuOp80] Huet, G. and Oppen, D. (1980): *Equations and rewrite rules: a survey.* in: R.Book (ed.), Formal Languages : Perspectives and Open Problems, (Academic Press, New York, 1980)

[JoKo86] Jouannaud, J., and Kounalis, E. (1986): *Automatic proofs by induction in equational theories without constructors.* Proc. of Symposium on Logic in Computer Science, 358-366

[KaLe80] Kamin, S., and Levy, J-J. (1980): *Attempts for generalizing the recursive path ordering.* Unpublished Manuscript, INRIA

[KaNaZh86] Kapur, D., Narendran, P., and Zhang, H. (1986): *Proof by Induction using test sets.* 8th Intl. Conf. on Automated Deduction, Lecture Notes in Computer Science, 230, Springer Verlag, New York.

[KaNaZh87] Kapur, D., Narendran, P., and Zhang, H. (1987): *On sufficient completness and related properties of term rewriting systems.* Acta Informatica, Vol. 24, Fasc. 4, 395-416

[KaZh89] Kapur, D., Zhang, H. (1989): *RRL : Rewrite Rule Laboratory User's Manual*

[KnBe70] Knuth, D.E. and Bendix, P.B. (1970): *Simple word problems in universal algebras.* in: Computational Problems in Abstract Algebras. (ed. J. Leech), Pergamon Press, 237-297.

[La81] Lankford, D.S. (1981): *A simple explanation of an inductionless induction.* MTP-14, Louisiana Tech University. Ruston, LA, 1981

[ZhKaKr88] Zhang, H., Kapur, D., and Krishnamoorthy, M.S. (1988): *A mechanizable induction principle for equational specifications.* Proc. of Ninth International Conference on Automated Deduction (CADE-9), Argonne, IL , May 1988. Springer-Verlag LICS 310