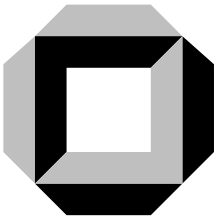

Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
Prof. Dr.-Ing. D. Schmid
Kaiserstr. 12, Geb. 20.20
76128 Karlsruhe

Ein funktionaler Ansatz zur systematischen Formalisierung regulärer Schaltungen

by
Dirk Eisenbiegler, K. Schneider, and R. Kumar

Technical Report SFB 358-C2-15/93
Juni 1993

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Um eine Verifikation durchfhren zu knnen, mu die Hardwarebeschreibung durch eine Formel in einer Logik ausgedrckt werden. Bevor ein Baustein, der in einer HDL (hardware description language) definiert wurde, verifiziert werden kann, ist es deshalb notwendig, die Hardwarebeschreibung in eine entsprechende logische Formel zu konvertieren. Dazu mu bekannt sein, welche Semantik sich hinter den einzelnen Konstrukten der HDL verbirgt und wie sich daraus die Semantik der gesamten Hardwarebeschreibung ableiten lt. Es soll von einer *formal verankerten* HDL gesprochen werden, wenn die HDL eine formal definierte Semantik hat, mit der man jeder Hardwarebeschreibung in eindeutiger Weise eine entsprechende Formel zuordnen kann.

Die formale Verankerung von HDLs ist nicht nur fr die reine Verifikation relevant, sie bildet auch die Grundlage fr formal korrekte HDL–HDL–Umformungen. Die quivalenz von HDL–Schaltungsbeschreibungen wird dabei auf die Ebene der Logik heruntergebrochen: Zwei Schaltungsbeschreibungen, die in der Syntax einer HDL gegeben sind, sind genau dann quivalent, wenn die Formeln, die ihnen durch die Semantik zugeordnet werden, logisch quivalent sind.

Eine HDL formal zu verankern bedeutet, die Lcke zwischen HDL und Logik zu schlieen. Die HDL und die Logik sind recht unterschiedliche Sprachen, was schon daran liegt, da an die beiden Sprachen sehr unterschiedliche Anforderungen gestellt werden. Die HDL ist das Gegenstck zur hheren Programmiersprache im Softwareentwurf. Sie mu ber problemangepate, mchtige Konstrukte zur Darstellung von Signalen, Bausteinschnittstellen, Bausteinstrukturen, Bausteinverhalten etc. verfgen. Die Logik dagegen mu sehr einfach aufgebaut sein. Nur so ist es mglich, die Semantik der Logik, d.h. die Gltigkeit der Formeln der Logik, klar zu definieren. Wie aufwendig die Logik sein mu, hngt davon ab, welche Ausdrucksstrke gefordert wird.

Je grer die Lcke zwischen der HDL und der Logik ist, desto aufwendiger wird es, eine Schaltungsbeschreibung in einer HDL auf eine Formel abzubilden und je aufwendiger diese Abbildung wird, desto schwerer wird es, diese Abbildung ,,korrekt" zu implementieren, mit ihr also die gewnschte Semantik der HDL auszudrcken.

In [BGGH92] wird zwischen *flacher Verankerung* und *tiefer Verankerung* von HDLs unterschieden. Bei der tiefen Verankerung wird die Semantik der HDL durch eine Abbil-

dung definiert, die den Konstrukten der HDL direkt Grundkonstrukte der Logik zuordnet. Dieser Abbildungsvorgang ist i.a. unumkehrbar. Bei der tiefen Verankerung werden dagegen innerhalb der Logik hhere Logikkonstrukte konstruiert, die genau den Konstrukten der HDL entsprechen. Die Semantik der HDL verbirgt sich dann in der Definition der hheren Logikkonstrukte. Im Idealfall kann die Schaltungsbeschreibung einer tief verankerten HDL direkt als Term der Logik gelesen werden. Die Beziehung zwischen Logik und HDL ist dann eins zu eins. Ein bersetzungsvorgang ist nicht mehr notwendig. Im allgemeinen lassen sich syntaktische Unterschiede jedoch nicht vermeiden.

Viele existierende HDLs wie HOL, ELLA und VHDL sind ursprnglich nicht formal verankert. Ihre Semantik wurde umgangssprachlich definiert. Es gibt mehrere Anstrengungen diesen HDLs nachtrglich eine passende formal exakte Semantik zuzuordnen [BGGH92, BGHT91, BoPS92]. Die Umsetzung der umgangssprachlichen Semantik in eine formale Semantik ist nur dann mglich, wenn die umgangssprachliche Definition der Semantik tatschlich eindeutig ist. Wie gro der Aufwand fr eine nachtrgliche formale Verankerung der HDL ist, hngt vor allem davon ab, wie nah die Konstrukte der HDL an der Logik sind. Funktionale Konstrukte, wie man sie beispielsweise in ELLA findet, lassen sich vergleichsweise einfach in HOL umsetzen. Sowohl die rein funktionalen Konstrukte in ELLA als auch die Formeln in HOL basieren auf dem typisierten λ–Kalkl. Es gibt lediglich Unterschiede in der jeweiligen Schreibweise. Dagegen ist die formale Beschreibung imperativer Programmkonstrukte, wie man sie beispielsweise in VHDL findet, bei weitem aufwendiger. Zu jedem Befehl mu definiert werden, wie er den globalen Zustand verndert, d.h. in welcher Relation der Zustand des Programms vor der Ausfhrung zum Zustand des Programms nach der Ausfhrung des Befehls ist. Die Semantik von Unterprogrammen mu aus der Semantik der verwendeten Befehle und aus den verwendeten Kontrollstrukturen abgeleitet werden. Diese Semantik lt sich nicht so unmittelbar in HOL bertragen.

In anderen Forschungsprojekten werden, statt nachtrglich bereits bestehende HDLs formal zu verankern, neue HDLs entwickelt, die direkt auf der Logik aufgebaut werden. Ein Beispiel fr eine solche HDL ist HML [OLLA92]. HML ist eine SML–hnliche Sprache mit einigen hardwarespezifischen Erweiterungen.

Der in dieser Arbeit erarbeitete Ansatz zur Formalisierung von Hardware geht folgenden Weg: Zunchst wird eine einfache funktionale Programmiersprache namens PML (primitive ML) definiert, deren Semantik in HOL verankert ist. PML ist eine Allzweck–Programmiersprache und besitzt keine hardwarespezifischen Sprachkonstrukte. In PML knnen, in hnlicher Weise wie in ML, ganz allgemeine μ–rekursive Funktionen definiert werden. In einem zweiten Schritt wird dann erlutert, wie Schaltungen durch Funktionen reprsentiert werden knnen und wie Beschreibungen von Schaltungen und Schaltungsstrukturen in PML umgesetzt werden knnen.

Mit der Einfhrung der Sprache PML soll nicht der Versuch gemacht werden, mit HDLs wie VHDL zu konkurrieren. PML ist keine HDL. Vielmehr ist PML ein Ansatz, um die Lcke zwischen HDL und Logik von unten, also von der Logik her, zu schlieen. PML ist bereits eine hhere Programmiersprache, und es wird auch gezeigt werden, da es in dieser allgemeinen Programmiersprache auch mglich ist, Schaltungen und Schaltungsstrukturen zu beschreiben.

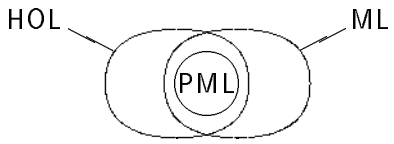

Abbildung 1.1: Die formalen Sprachen HOL, ML und PML.

PML ist eine funktionale Programmiersprache, die auf dem typisierten λ–Kalkl basiert. PML enthlt nur rein funktionale Elemente. Andere funktionale Sprachen wie SML und Lisp enthalten neben den rein funktionalen noch nichtfunktionale, imperative Konstrukte mit Seiteneffekten. Dadurch, da PML nur rein funktionale Konstrukte enthlt, ist diese Sprache sehr nahe an der Logik von HOL. PML–Terme sind, wie auch die HOL–Terme, typisierte λ–Terme. Vereinfacht gesagt, ist PML eine gemeinsame Teilsprache von HOL und SML 1.1. Die Syntax von PML entspricht der von SML. PML–Programm knnen in einer (geringfgig erweiterten) SML–Umgebung ausgefhrt werden. Auf diese Weise knnen Schaltungsbeschreibungen, die in PML implementiert wurden, ausgefhrt, simuliert werden. Durch eine einfache syntaktische Umformung der Terme kommt man zu einer Darstellung als Formeln in HOL. Diese Formeln bilden die Grundlage fr die Verifikation.

Schaltungen werden in PML immer durch Funktionen reprsentiert werden, die Eingangssignale auf Ausgangssignale abbilden. Da lediglich funktionale Zusammenhnge zwischen Ein– und Ausgangsfunktionen zugelassen werden, bedeutet eine Beschrnkung der Ausdrucksmchtigkeit: Fr jede Belegung der Eingangssignale mu die Belegung der Ausgangssignale eindeutig festgelegt werden. Das Verhltnis zwischen Ein– und Ausgangssignalen liee sich auch allgemeiner durch Relationen beschreiben. Es mte dann nicht explizit zwischen Ein– und Ausgangssignalen unterschieden werden. Alle Signale knnen sich bei Relationen gegenseitig beeinflussen. Funktionale bzw. relationale Schaltungsbeschreibungen haben unterschiedliche Vor– und Nachteile, auf die noch genauer eingegangen werden soll. Vorteile der funktionalen Schaltungsbeschreibung sind die Ausfhrbarkeit und die Eindeutigkeit, die insbesondere die Widerspruchsfreiheit impliziert.

Oft werden Schaltungsbeschreibungen durch Relationen formalisiert. Relationale Schaltungsbeschreibungen sind allgemeiner als funktionale. Eine Umwandlung von einer funktionalen in eine relationale Schaltungsbeschreibung ist immer mglich, wohingegen die umgekehrte Richtung i.a. nicht mglich ist. In [Cami88] findet man ein Verfahren, mit dem relationale Schaltungsbeschreibungen, die in einer bestimmten Form gegeben sein mssen, in funktionale Schaltungsbeschreibungen konvertiert werden knnen, um sie dann per Evaluierung simulieren zu knnen.

Ein Schwerpunkt dieser Arbeit liegt auf der Formalisierung regulrer Schaltungsstrukturen. Regularitt soll explizit ausgedrckt werden. Damit ist folgendes gemeint: Statt eine regulre Struktur, in der ein Baustein 100–fach in Serie geschaltet wird, durch eine Netzliste zu beschreiben, in der der Baustein 100–fach vorkommt, soll statt dessen zunchst das Schema der n–fachen Serienschaltung durch eine Rekursionsschema allgemein beschrieben werden, und von dieser n–fachen Serienschaltung soll dann die 100–fache Serienschaltung

abgeleitet werden. Der Vorteil dieser Vorgehensweise besteht darin, da auf diese Weise die Regularitt tatschlich genutzt werden kann: Statt die quivalenz zweier solcher 100–fach Serienschaltungen durch ,,100–fache Fallunterscheidung" zu beweisen, soll per Induktion die quivalenz der n–fach Serienschaltungen bewiesen werden, woraus dann direkt auch die quivalenz der 100–fach Serienschaltung folgt. An die Stelle komplexer Fallunterscheidungen treten schnelle Induktionsbeweise.

Bei der Formalisierung von Hardwareschaltungen in PML werden zwei Formen von Funktionen unterschieden werden: Zum einen Funktionen, die als *konkrete Schaltungen* bezeichnet werden, und die genau eine reale Schaltung reprsentieren, und zum anderen Funktionen, die fr eine Menge von Schaltungen stehen. Die letztgenannten Funktionen werden als *abstrakte Schaltungen* bezeichnet. Die n–fache Serienschaltung ist ein Beispiel fr eine abstrakte Schaltung. Als eine Ausprgung einer abstrakten Schaltung soll eine konkrete Schaltung bezeichnet werden, die zu der von der abstrakten Schaltung beschriebenen Schaltungsmenge gehrt. Die 100–fache Serienschaltung ist ein Beispiel fr eine Ausprgung der n–fachen Serienschaltung.

Die Arbeit beginnt mit der Einfhrung der Sprache PML in Kapitel 2. Hier werden die Sprachkonstrukte zunchst nur syntaktisch eingefhrt, und ihre Bedeutung wird umgangssprachlich erlutert. In Kapitel 3 folgt dann eine formal exakte Definition der Semantik von PML in HOL. In Kapitel 4 und 5 wird errtert, welche Mglichkeiten zur funktionalen Formalisierung von Schaltungen bestehen, und es wird insbesondere auch ein Schema zur Formalisierung regulrer Schaltnetze und Schaltwerke vorgestellt. Kapitel 4 widmet sich konkreten Schaltungen und Kapitel 5 abstrakten Schaltungen. Das Schema der funktionalen Schaltungsbeschreibung wird dann in Kapitel 6 an zwei greren Fallbeispielen illustriert. In Kapitel 7 wird die Problemstellung der Verifikation erlutert und es wird aufgezeigt, welche Techniken zur Beweisfindung angewandt werden knnen.

Kapitel 2

PML

In diesem Kapitel wird eine funktionale Programmiersprache mit dem Namen primitive ML, kurz PML, eingeführt. Das besondere an der Sprache PML ist, daß die Semantik der Sprache in einer Logik höherer Ordnung formal verankert ist. Die Sprache baut auf der Logik auf — die Konstrukte der Sprache PML sind nichts anderes als Schreibweisen für Formeln, die die Semantik der Konstrukte beschreiben.

Der Name primitive ML deutet auf die Verwandtschaft mit der Programmiersprache ML hin. PML ist eine Teilsprache von ML. Die verwendete Syntax ist identisch. In PML sind jedoch nur sehr wenige Grundkonstrukte definiert und es sind im Vergleich zu ML mehrere Einschränkungen zu beachten. Einige Elemente der Sprache ML wie Ausnahmebehandlungen und Funktionen mit Seiteneffekten kommen in PML nicht vor.

Mit PML können ganz allgemein μ–rekursive Funktionen beschrieben werden. Da die Semantik von PML in HOL formal verankert ist, kann über PML–Programmen auf logischer Ebene argumentiert werden. Ein PML–Programm entspricht einer HOL–Formel, die die in dem Programm beschriebenen Datentypen und Funktionen charakterisiert. Durch eine weitere HOL Formel kann eine Eigenschaft für dieses Programm formuliert werden und diese Eigenschaft kann auf der logischen Ebene verifiziert werden. Die Äquivalenz von PML–Programmen kann exakt definiert werden: Zwei PML–Programme sind genau dann äquivalent, wenn die zugehörigen Formeln, die die Semantik der beiden Programme beschreiben, logisch äquivalent sind.

In dieser Arbeit bildet PML die Grundlage für ein funktionales Formalisierungsschema zur Beschreibung von Schaltnetzen und synchronen Schaltwerken. Es soll jedoch der Eindruck vermieden werden, daß PML eine Hardware–Beschreibungssprache ist. PML besitzt keine hardwarespezifischen Konstrukte — es können lediglich μ–rekursive Funktionen definiert werden, und diese Funktionen können als Hardwarebeschreibungen interpretiert werden.

In diesem Kapitel wird zunächst die Syntax von PML vorgestellt, und es wird grob erläutert, welche Funktionalität die Konstrukte haben sollen. Im Kapitel 3 wird dann die Semantik der Sprachelemente in HOL formal definiert.

2.1 Datentypen

Datentypen, die in PML verwendet werden sollen, müssen mit der Anweisung

`primitive_datatype` *string*;

deklariert werden. Die Zeichenkette, die der Funktion als Parameter übergeben wird, enthält die Beschreibung des neuen Datentyps: seinen Namen, die Namen seiner Konstruktoren und die Typen der Argumente der Konstruktoren. Die Zeichenkette muß den folgenden Aufbau haben:

newdatatype `=`
constructor `of` *type* `#` *type* `#` ... `#` *type* |
constructor `of` *type* `#` *type* `#` ... `#` *type* |
⋮
constructor `of` *type* `#` *type* `#` ... `#` *type*

Beispielsweise werden durch die folgende Datentypdeklaration die natürlichen Zahlen eingeführt:

```
primitive_datatype  " num = Zero | Suc of num " ;
```

Mit dieser Anweisung wird der Datentyp *num* mit den beiden Konstruktoren Zero_{num} und $\mathsf{Suc}_{num \to num}$ deklariert. Zum Vergleich die entsprechende Deklaration in ML–Syntax:

```
datatype num = Zero | Suc of num;
```

Nicht jeder ML–Datentyp kann jedoch auch in PML deklariert werden. Es müssen die folgenden beiden Regeln eingehalten werden:

1. In ML sind die Argumente der Konstruktoren in einem Tupel zusammengefaßt. Dagegen müssen die PML–Konstruktoren in Curry–Form sein. So hat beispielsweise der Listenkonstruktor Cons in ML den Typ $(\alpha * (\alpha)list) \to (\alpha)list$ und in PML den Typ $\alpha \to (\alpha)list \to (\alpha)list$

2. Für die Typen der Argumente der Konstruktoren gilt die Einschränkung, daß es sich entweder um atomare Typen, um Typvariablen oder um den neu zu definierenden Typ selbst handeln muß.

In PML sind bereits die Datentypen *bool*, *prod*, *list*, *num* und *partial* vordefiniert. Sie sind folgendermaßen definiert:

```
primitive_datatype  " bool    = T | F " ;
primitive_datatype  " prod    = Comma of 'a # 'b " ;
primitive_datatype  " list    = Nil | Cons of 'a # list " ;
primitive_datatype  " num     = Zero | Suc of num " ;
primitive_datatype  " partial = Defined 'a | Undefined " ;
```

Für Terme über den Datentypen *prod*, *list* und *num* gibt es benutzerfreundlichere Schreibweisen:

- Statt (Comma a b) darf (a, b) geschrieben werden.
- Die „Zahlen" Zero, (Suc Zero), (Suc(Suc Zero)), ... dürfen mit den Numeralen $0, 1, 2, \ldots$ bezeichnet werden.
- Listenausdrücke der Form Nil, (Cons a Nil), (Cons a (Cons b Nil)), ... dürfen mit $[]$, $[a]$, $[a, b], \ldots$ bezeichnet werden.
- Statt $(\alpha, \beta)prod$ darf $\alpha * \beta$ geschrieben werden.

2.2 Primitive Rekursion

In PML wird mit jeder Datentypdeklaration automatisch eine zugehörige Grundfunktion mit dem Namen PRIMREC_*type* eingeführt. Diese Funktion beschreibt die primitive Rekursion über diesem Datentyp. Die Definition der PRIMREC–Funktion soll zunächst nur für die Datentypen *bool*, *prod*, *list*, *num* und *partial* erklärt werden. Eine allgemeine Definition der PRIMREC–Funktion folgt in Kapitel 3.

PRIMREC_bool T a b $= a$
PRIMREC_bool F a b $= b$

PRIMREC_prod (Comma x y) f $= f\ x\ y$

PRIMREC_list Nil a f $= a$
PRIMREC_list (Cons x y) a f $= f\ x\ y$ (PRIMREC_list y a f)

PRIMREC_num Zero a f $= a$
PRIMREC_num (Suc n) a f $= f\ n$ (PRIMREC_num n a f)

PRIMREC_partial (Defined x) f a $= f\ x$
PRIMREC_partial Undefined f a $= a$

Die Verwendung der PRIMREC–Funktionen soll am Beispiel der Funktion PRIMREC_num erläutert werden. Man betrachte die folgenden beiden Gleichungen, die eine Funktion g durch primitive Rekursion über dem Datentyp *num* definieren:

g Zero $= a$
g (Suc x) $= f\ x$ (g x)

Man erkennt, daß die primitive Rekursion eindeutig durch a und f charakterisiert wird. a und f können beliebig sein — abgesehen davon natürlich, daß ihre Typen passend sein müssen. Die Funktion PRIMREC_num hat die folgende Wirkung: Sie erhält als Parameter eine Variable x vom Typ *num* und die beiden Funktionen a und f und liefert als Funktionswert jenes (gx), welches durch das obige Gleichungssystem in eindeutiger Weise beschrieben wird.

In PML darf die Funktion g nicht in der oben dargestellten Form definiert werden, da zum einen PML–Funktionsdefinitionen aus genau einer Gleichung bestehen müssen und zum anderen die neu zu definierende Funktion nicht auf der rechten Seite der Gleichung vorkommen darf. Die Implementierung von g sieht in PML folgendermaßen aus:

```
fun g x = PRIMREC_num x a f;
```

Oft wird ein besonderer Spezialfall der primitiven Rekursion benötigt, bei dem die rechten Seiten des Gleichungssystems nicht von der zu definierenden Funktion abhängen. Dieser Speziallfall der primitiven Rekursion wird als Fallunterscheidung bezeichnet. Für *num* sieht eine derartige Fallunterscheidung so aus:

$$\begin{aligned} \mathsf{h}\ \mathsf{Zero} &= a \\ \mathsf{h}\ (\mathsf{Suc}\ x) &= f\ x \end{aligned}$$

Mit PRIMREC_num könnte h folgendermaßen implementiert werden:

```
fun h x = PRIMREC_num x a (fn x => fn y => f x);
```

Für die Fallunterscheidung gibt es in PML aber auch eine gesonderte Funktion mit dem Namen CASE_*type*. Die CASE–Funktionen werden wie die PRIMREC–Funktionen bei einer Datentypdeklaration automatisch mitdefiniert. Für die Typen *bool*, *prod*, *list*, *num* und *partial* sind die zugehörigen CASE–Funktionen folgendermaßen definiert:

$$\begin{aligned} \mathsf{CASE\_bool}\ \mathsf{T}\ a\ b &= a \\ \mathsf{CASE\_bool}\ \mathsf{F}\ a\ b &= b \end{aligned}$$

$$\mathsf{CASE\_prod}\ (\mathsf{Comma}\ x\ y)\ f = f\ x\ y$$

$$\begin{aligned} \mathsf{CASE\_list}\ \mathsf{Nil}\ a\ f &= a \\ \mathsf{CASE\_list}\ (\mathsf{Cons}\ x\ y)\ a\ f &= f\ x\ y \end{aligned}$$

$$\begin{aligned} \mathsf{CASE\_num}\ \mathsf{Zero}\ a\ f &= a \\ \mathsf{CASE\_num}\ (\mathsf{Suc}\ n)\ a\ f &= f\ n \end{aligned}$$

$$\begin{aligned} \mathsf{CASE\_partial}\ (\mathsf{Defined}\ x)\ f\ a &= f\ x \\ \mathsf{CASE\_partial}\ \mathsf{Undefined}\ f\ a &= a \end{aligned}$$

Verwendet man CASE_num statt PRIMREC_num, so kann man h in PML folgendermaßen implementieren:

```
fun h x = CASE_num a f;
```

2.3 μ–Rekursion

Bisher sind lediglich zwei Arten von Grundfunktionen vorgestellt worden, nämlich die PRIMREC– und die CASE–Funktionen. Diese beiden Grundfunktionen bilden die Basis für eine in sich abgeschlossene Teilsprache von PML, in der alle primitiv rekursiven Funktionen über den deklarierten Datentypen definiert werden können.

In diesem Abschnitt wird eine weitere Grundfunktionen namens WHILE vorgestellt. Mit dieser Erweiterung ist es möglich, allgemeine μ-rekursive Funktionen, also beliebige berechenbare Funktionen, zu beschreiben.

Im Gegensatz zu den bisher betrachteten primitiv rekursiven Funktionen besteht bei μ–rekursiven Funktion die Möglichkeit, daß sie nicht terminieren. Zur Beschreibung der Funktionswerte partieller Funktionen wird in PML der Datentyp *partial* verwendet:

```
datatype  " partial = Undefined | Defined 'a " ;
```

Während primitiv rekursive Funktionen in PML ganz allgemein den Typ $\alpha \to \beta$ haben, haben μ–rekursive Funktionen in PML den Typ $\alpha \to (\beta)partial$.

Der wesentliche Unterschied zu ML ist der, daß in PML auch für den Fall, daß eine μ–rekursive Funktion nicht terminiert, explizit ein Funktionswert, nämlich Undefined, festgelegt ist. In dem Fall, daß der Funktionswert einer μ–rekursiven Funktion in PML definiert ist, die Evaluierung also terminiert, ist der Funktionswert (Defined y). In ML dagegen gibt es Funktionen, denen für bestimmte Argumente keine Funktionswerte zugeordnet sind. Solche ,,echt partiellen" Funktionen gibt es in PML nicht. In PML ist jeder Funktionswert eindeutig festgelegt.

Die Grundfunktion WHILE dient zur Beschreibung von Schleifen. Sie hat drei Parameter: eine Schleifenbedingung $g_{\alpha \to bool}$, eine μ–rekursive Funktion $f_{\alpha \to (\alpha)partial}$ und einen Anfangswert x_α. Die Funktion WHILE iteriert die Funktion f so oft, bis die Schleifenbedingung g nicht mehr erfüllt ist.

Man betrachte die nachstehende Folge:

$$x_0 = x, \quad x_1 = f(x), \quad x_2 = f^2(x), \quad x_3 = f^3(x), \ \ldots$$

Gibt es ein x_i mit $(g\, x_i) = \mathsf{F}$, dann gibt es auch ein eindeutig definiertes kleinstes j mit $(g\, x_j) = \mathsf{F}$. In diesem Fall hat (WHILE $g\ f\ x$) den Funktionswert (Defined x_j). Gibt es dagegen kein x_i, für das $(g\, x_i)$ den Wert F hat, dann hat (WHILE $g\, f\, x$) den Funktionswert Undefined.

2.4 Abgeleitete Funktionen und Konstanten

Bisher wurden nur Grundfunktionen vorgestellt. Grundfunktionen sind: die PRIMREC– und CASE–Funktionen und die Funktion WHILE. Die Definition von abgeleiteten Funktionen und Konstanten erfolgt, ähnlich wie in ML, durch fun– bzw. val–Anweisungen.

```
fun name  par par ... par  = term ;
val name  = term  ;
```

Als Terme auf der rechten Seite einer Funktions– oder Konstantendefinition sind nur die nach folgendem Schema konstruierten Terme zulässig:

1. Variablen, Konstruktoren und bereits definierte Funktoren sind zulässige Terme

2. Ist x eine Variable und ist q ein zulässiger Term, dann ist auch die λ–Abstraktion (fn x => q) ein zulässiger Term.

3. Sind $p_{\alpha\rightarrow\beta}$ und q_α zulässige Terme, dann ist die Funktionsanwendung (p q) ein zulässiger Term.

In dem Term auf der rechten Seite einer Funktionsdefinition dürfen nur die in der Parameterliste vorkommenden Variablen frei vorkommen. Bei Konstantendefinitionen muß auf der rechten Seite ein geschlossener Term stehen. In der Parameterliste der Funktionsdefinition sind nur Variablen und gepaarte Variablen zulässig. Im Term auf der rechten Seite einer Funktions– bzw. Konstantendefinition dürfen lediglich Grundfunktionen und bereits definierte Funktionen, verwendet werden. Insbesondere darf, im Gegensatz zu ML, die neu zu definierende Funktion selbst nicht auf der rechten Seite vorkommen. Dies ist in PML auch nicht notwendig, da zur Bildung rekursiver Funktionen geeignete Grundfunktionen zur Verfügung stehen.

Typvariablen

In ML werden den Typvariablen polymorpher Terme automatisch Namen zugeordnet. In PML müssen dagegen (wie in HOL) alle vorkommenden Typvariablen explizit benannt werden. Beispielsweise ist

```
fun id x = x;
```

nicht zulässig. Richtig wäre:

```
fun id (x:'a) = x;
```

Gepaarte λ–Abstraktion

Neben der ,,gewöhnlichen" λ–Abstraktion, bei der auf der linken Seite eine Variable steht, ist noch eine abgewandelte Form, die λ–Abstraktion über mehrere, durch Paarbildung zusammengefügte Variablen, erlaubt. Bei der gepaarten λ–Abstraktion handelt es sich lediglich um eine Schreibweise, die auf CASE_prod zurückgeführt werden kann. So darf beispielsweise der Ausdruck

(fn (x, y) => q)

verwendet werden. Dieser Ausdruck steht für:

(CASE_prod (x, y) q)

In analoger Weise lassen sich alle λ–Terme über beliebig geschachtelten Paarungen von Variablen als eine andere Schreibweise eines äquivalenten Terms mit CASE_prod–Funktionen interpretieren.

let–Terme

Unter einem β–Redex versteht man einen Term der Form:

$$((\mathsf{fn}\ x => q)\ a)$$

Tief geschachtelte β–Redizes sind schwer lesbar, da die Variable x und der ihr zugeordnete Term a dann räumlich weit auseinander zu stehen kommen. Aus diesem Grund gibt es in ML (und auch in HOL) sogenannte let–Terme. Auch in PML wird diese Schreibweise erlaubt. let–Terme sind lediglich andere Schreibweisen für β–Redizes mit einer umgestellten Reiheinfolge der Teilterme: Der Term a steht bei let–Termen direkt hinter der Variablen, der er zugeordnet ist. Der nachfolgende let–Term ist (definitionsgemäß) äquivalent zu dem oben erwähnten β–Redex:

$$(\mathsf{let\ val}\ x\ =\ a\ \mathsf{in}\ q\ \mathsf{end})$$

geschrieben werden. Die let–Term–Schreibweise ist nicht nur in dem Fall erlaubt, daß der linke Teilterm des β–Redexes eine gewöhnliche λ–Abstraktion ist, sondern auch dann, wenn dieser Term eine gepaarter λ–Abstraktion ist. An der Stelle der Variablen x darf also auch eine beliebig geschachtelte Paarung von Variablen steht.

2.5 Ein Beispielprogramm

2.5.1 Aufgabenstellung

Die Verwendung der Sprache PML soll jetzt anhand eines Beispiels illustriert werden. Die zu implementierende Funktion sei

$$f(a,b) = \sqrt{a^2 + b^2}$$

Dabei sollen beide Parameter natürliche Zahlen sein, und auch der Funktionswert soll eine natürliche Zahl sein. Der berechnete Funktionswert soll eine möglichst gute Näherung von $\sqrt{a^2+b^2}$ sein. Zur Berechnung der Quadratwurzel wird die Iterationsgleichung des Heronverfahrens benutzt:

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}$$

Abbildung 2.2 zeigt die PML–Implementierung. Die Implementierung komplexer rekursiver Funktionen in PML ist aus programmiertechnischer Sicht schwierig, da nur sehr einfache Grundfunktionen zur Verfügung stehen. PML–Programme sind oft schwer lesbar. Aus diesem Grunde wird zur Illustration eine „entsprechende" Implementierung in ML vorgestellt (Abbildung 2.1). Die beiden Implementierungen in PML und ML sind jedoch nicht echt äquivalent. Wie bereits gesagt, haben in PML–Programmen die Funktionswerte μ–rekursiver Funktionen den Typ $(\beta)partial$. Es unterscheiden sich deshalb die Typen der Funktionswerte von f in den beiden Implementierungen: In der PML–Implementierung hat f den Typ $num \rightarrow (num)partial$ und in der ML–Implementierung den Typ $num \rightarrow num$.

2.5.2 Anmerkungen zur Implementierung in PML

Bei der Implementierung werden vier Typen verwendet: *bool*, *num*, *prod* und *partial*. In PML sind diese beiden Datentypen bereits definiert.

Zunächst werden für die natürlichen Zahlen einfache arithmetische Funktionen definiert: die Addition (add), die Multiplikation (mult), die Subtraktion (sub) und die kleiner–gleich–Relation (less_than). Diese vier Funktionen werden durch primitive Rekursion definiert.

Bei der Funktion add kann man von der folgenden Spezifikation ausgehen:

$$\begin{aligned} \mathsf{add}\ \mathsf{Zero}\ b &= b \\ \mathsf{add}\ (\mathsf{Suc}\ a)\ b &= \mathsf{Suc}(\mathsf{add}\ a\ b) \end{aligned}$$

Diese Gleichungen erinnert bereits an das Gleichungssystem, mit dem PRIMREC_num definiert wurde. Der einzige Unterschied besteht darin, daß in der zweiten Zeile die äußerste Funktion nicht nur von (add a b), sondern auch noch von b abhängt. Durch eine kleine Umformung kommt man zu der folgenden Darstellung:

$$\begin{aligned} \mathsf{add}\ \mathsf{Zero}\ b &= b \\ \mathsf{add}\ (\mathsf{Suc}\ a)\ b &= (\mathsf{fn}\ n => \mathsf{Suc})\ b\ (\mathsf{add}\ a\ b) \end{aligned}$$

Jetzt hat das Gleichungssystem die gewünschte Form. Die PML–Implementierung von add lautet folgendermaßen:

```
fun add a b  =  PRIMREC_num a b (fn n => Suc);
```

Analog kann man auch die Funktion mult programmieren. Schwieriger ist die Implementierung der Funktion sub, die den Betrag der Differenz zweier Zahlen berechnen soll. Wie man bereits an der ML–Implementierung erkennt, handelt es sich um eine geschachtelte primitive Rekursion über beide Argumente. Bei der Implementierung in PML wird folgendermaßen vorgegangen: Zunächst wird ein Term gebildet, der zu einer frei vorkommenden Variablen b eine Funktion beschreibt, die zu einem Argument x den Betrag der Differenz aus b und x berechnet. Dies geschieht durch primitive Rekursion über b.

```
PRIMREC_num b
   (fn x => x)
   (fn n => fn r => fn x => CASE_num x n (fn m => r m))
```

Wendet man nun auf diesen Term ein Variable a an, dann erhält man einen Term, der den Betrag der Differenz aus a und b beschreibt. Auf diese Weise kann dann sub definiert werden.

```
fun sub a b =
   PRIMREC_num b
      (fn x => x)
      (fn n => fn r => fn x => CASE_num x n (fn m => r m))
         a;
```

Analog zu sub verläuft die Implementierung von less_than.

Als nächstes werden noch die Division div und die Quadratwurzel sqrt definiert. Diese beiden Funktionen werden als μ–rekursive Funktionen mit Hilfe der Funktion WHILE implementiert. Bei div wird der Divisor wird so oft vom Dividenden subtrahiert, bis dieser kleiner als der Divisor geworden ist, und bei sqrt wird die oben beschriebene Iterationsgleichung so oft angewandt, bis die Differenz zweier benachbarter Folgeglieder kleiner oder gleich 1 ist.

```
datatype num = Zero | Suc of num;

datatype bool = T | F;


fun add Zero b    =  b  |
    add (Suc a) b =  Suc (add a b);

fun mult Zero b    =  Zero  |
    mult (Suc a) b =  add b (mult a b);

fun sub a Zero          =  a  |
    sub Zero b          =  b  |
    sub (Suc a) (Suc b) =  sub a b;

fun less_than a Zero          =  F  |
    less_than Zero (Suc b)    =  T  |
    less_than (Suc a) (Suc b) =  less_than a b;

fun div a b =
   case (less_than a b) of
      T => Zero  |
      F => (Suc (div (sub a b) b));

val two = Suc(Suc Zero);

fun sqrt_approx prox x =
   let
      val new_prox = div (add prox (div x prox)) two
   in
      case (less_than (sub new_prox prox) two) of
         T => new_prox |
         F => (sqrt_approx new_prox x)
   end;

fun sqrt x = sqrt_approx x x;

fun f a b = sqrt (add (mult a a) (mult b b));
```

Abbildung 2.1: Die Implementierung in ML

```
fun add a b  = PRIMREC_num a b (fn n => Suc);

fun mult a b = PRIMREC_num a Zero (fn n => add b);

fun sub a b =
   PRIMREC_num b
      (fn x => x)
      (fn n => fn r => fn x => CASE_num x n (fn m => r m))
         a;
fun less_than a b =
   PRIMREC_num b
      (fn x => F)
      (fn n => fn r => fn x => CASE_num x T (fn m => r m))
         a;

fun div a b =
   let val c =
      WHILE
         (fn ((x,y),z) => CASE_bool (less_than x y) F T)
         (fn ((x,y),z) => Defined ((sub x y,y),Suc z))
         ((a,b),Zero)
   in
      PRIMREC_partial c (fn (xy,z) => Defined z) Undefined
   end;

val two = Suc(Suc Zero);

fun sqrt a =
   let val c =
      WHILE
         (fn (x,oldx) => less_than two (sub x oldx))
         (fn (x,oldx) =>
            let val d = div a x in
            let val e =
               CASE_partial d (fn y => Defined(add x y)) Undefined in
            let val g = CASE_partial e (fn y => div y two) Undefined in
               CASE_partial g (fn y => Defined(y,x)) Undefined
            end end end)
         (Suc Zero,a)
   in
      CASE_partial c (fn (x,y) => Defined x) Undefined
   end;

fun f a b = sqrt (add (mult a a) (mult b b));
```

Abbildung 2.2: Die Implementierung in PML

2.6 Zusammenfassung

Es wurden alle Konstrukte der Sprache PML vorgestellt und es wurde anhand eines Beispiels gezeigt, wie mächtig die Sprache PML ist und inwieweit Funktionen, die in einer üblichen funktionalen Programmiersprache wie ML beschrieben werden können, auch in PML dargestellt werden können. Die folgende Auflistung faßt die wichtigsten Eigenschaften von PML zusammen:

- PML ist Turing–vollständig. Die in der Sprache PML beschreibbaren Funktionen decken die gesamte Menge der μ–rekursiven Funktionen ab.
- Alle Funktionen in PML sind total. Bei den μ–rekursiven Funktionen von PML ist der Funktionswert zwar i.a. nicht immer berechenbar — formal gesehen ist der Funktionswert jedoch immer eindeutig definiert.
- Das Typenkonzept von PML ist weniger mächtig als das von ML. Lediglich direkt rekursive Datentypdeklarationen sind in PML zugelassen. Eine Erweiterung des Datentypen ist denkbar, einschränkend ist jedoch zu bemerken, daß hier therotische Grenzen bestehen. Das Typenkonzept formal verankerter Sprachen kann nie derart mächtig sein wie beispielsweise das von ML ([Gunt92]).
- In der Parameterliste einer ML–Funktionsdefinition dürfen Terme stehen, in denen beliebige Konstruktoren vorkommen. Bei einem Typ mit mehreren Konstruktoren kann das zu Funktionen führen, deren Funktionswert nicht immer definiert ist. In PML ist in diesen Termen lediglich der Konstruktor Comma erlaubt.
- Es wird in PML keine Ausnahmebehandlung (exceptions) angeboten.
- Das ML–Konstrukt case ... of ... gibt es in PML nicht.
- Das Schreiben von rekursiven Funktionen ist in PML bei weitem schwieriger, da nur sehr einfache, elementare Grundfunktionen zur Verfügung stehen. Obwohl die Sprache ML nicht mächtiger ist — auch mit ihr kann man „nur" berechenbare Funktionen beschreiben — so ist es in ihr doch einfacher, rekursive Funktionen zu beschreiben, da man in ihr die rekursiven Funktionen nicht auf elementare primitive Rekursion und μ–Rekursion zurückführen muß.
- Funktionen können in PML keine Seiteneffekte haben. In ML gibt es zahlreiche Funktionen mit Seiteneffekten, also Funktionen, bei denen während der Evaluierung globale Größen verändert werden. Diese globalen Größen können dann wieder andere Funktionen während der Evaluierung beeinflussen. Typisch sind hierfür Ausgabefunktionen und alle Funktionen, die mit Zeigertypen zu tun haben.

  Durch die Verwendung von Funktionen mit Seiteneffekten ergeben sich jedoch für die Verifikation zwei gravierende Probleme. Zum einen muß zur Beschreibung der Funktion nicht nur der Zusammenhang zwischen den Parametern und dem Funktionswert betrachtet werden, sondern es muß auch die Beeinflussung durch und die

Auswirkungen auf globale Größen untersucht werden. Das zweite Problem besteht darin, daß die Wirkung einer Funktion mit Seiteneffekten auch von der Evaluierungsreihenfolge abhängt.

Kapitel 3

Die Semantik von PML

3.1 Grundlagen des Beweissystems HOL

Im folgenden wird die Logik HOL soweit vorgestellt, wie dies für die Formalisierung der Sprache PML notwendig ist. Für eine allgemeine Einführung in HOL sei auf [HOL93] und speziell zum Thema Datentypdeklarationen sei auf [Melh88] verwiesen.

3.1.1 Grundelemente

Die Logik von HOL baut auf einigen wenigen Grundelementen auf. Diese Grundelemente sind:

- die beiden atomaren Typen *bool* und *ind*
- der zweistellige Typoperator $\rightarrow$, der in Infixschreibweise verwendet wird.
- die drei Konstanten $\Rightarrow_{bool\rightarrow bool\rightarrow bool}$, $=_{\alpha\rightarrow\alpha\rightarrow bool}$ und $\varepsilon_{(\alpha\rightarrow bool)\rightarrow bool}$.

Der Typ *bool* steht für die Menge der Wahrheitswerte $\{\mathsf{T},\mathsf{F}\}$. Der Typ *ind* steht für eine nicht näher spezifizierte Individuenmenge. Von dieser Individuenmenge ist lediglich eine Eigenschaft bekannt: Sie ist nicht endlich. Der Typenoperator $\rightarrow$ ordnet zwei Typen α und β einen neuen Typ $\alpha\rightarrow\beta$ zu, dessen Elemente die Funktionen von α nach β sind. Die Konstanten $\Rightarrow$, $=$ und ε stehen in üblicher Weise für die Implikation, die Gleichheit und den Hilbert-Operator.

Die Bedeutung dieser Typen, Typenoperatoren und Konstanten wird in HOL formal durch Axiome und Regeln definiert. In den Axiomen kommen neben den Konstanten $\Rightarrow$, $=$ und ε noch weitere, abgeleitete Konstanten vor. Die Bedeutung der Grundelemente wird durch die Axiome also in einer indirekten Weise beschrieben. Diese Vorgehensweise hat rein technische Gründe: die Axiome werden dadurch leichter lesbar. Genauso gut wäre es aber auch möglich gewesen, sich bei den Axiomen allein auf Grundelemente zu beschränken.

3.1.2 Konstantendefinitionen

In HOL besteht die Möglichkeit, eine Konstante einzeln einzuführen. Damit eine Konstante verwendet werden kann, muß einfach ein neuer Konstantenname und ein Typ angegeben werden. Es gibt dann aber noch keine Theoreme, die dieser Konstante eine Eigenschaft zuordnen, d.h. es können keinerlei Eigenschaften der Konstanten bewiesen werden.

Bei einer *Konstantendefinition* wird deshalb, neben der Konstanten selbst, eine Formel eingeführt, die diese Konstante beschreibt. Die Formel muß den Charakter einer ,,Abkürzung" haben: eine Gleichung zwischen der neuen Konstanten und einem beliebigen geschlossenen Term. Durch eine Konstantendefinition entsteht aus der Abkürzungsgleichung ein Theorem, das als *Definition* bezeichnet wird. Der gesamte Vorgang einer Konstantendefinition, also die gleichzeitige Einführung einer Konstantne und ihrer Definition, ist konsistenzerhaltend.

In HOL sind, aufbauend auf den Konstanten $\Rightarrow$, $=$ und ε, die folgenden Konstantendefinitionen vorgenommen worden:

$$\vdash \mathsf{T} = ((\lambda x_{bool}.\, x) = (\lambda x_{bool}.\, x))$$

$$\vdash \forall = (\lambda P_{\alpha\to bool}.\, P = (\lambda x.\, \mathsf{T}))$$

$$\vdash \exists = \lambda P_{\alpha\to bool}.\, P(\varepsilon\, P)$$

$$\vdash \mathsf{F} = \forall b_{bool}.\, b$$

$$\vdash \neg = \lambda b.\, b \Rightarrow \mathsf{F}$$

$$\vdash \wedge = \lambda b_1\, b_2.\, \forall b.\, (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$$

$$\vdash \vee = \lambda b_1\, b_2.\, \forall b.\, (b_1 \Rightarrow b) \Rightarrow (b_2 \Rightarrow b) \Rightarrow b)$$

$$\vdash \mathsf{One\_One} = \lambda f_{\alpha\to\beta}.\, \forall x_1\, x_2.\, (f\, x_1 = f\, x_2) \Rightarrow (x_1 = x_2)$$

$$\vdash \mathsf{Onto} = \lambda f_{\alpha\to\beta}.\, \forall y.\exists x.\, y = f\, x$$

$$\vdash \mathsf{Type\_Definition} = \lambda\, P_{\alpha\to bool}\, REP_{\beta\to\alpha}.\, \mathsf{One\_One}\, REP \wedge (\forall x.\, P\, x = (\exists y.\, x = REP\, y))$$

Die Konstanten T, $\forall$, $\exists$, F, $\neg$, $\wedge$ und $\vee$ haben die in der Logik üblichen Bedeutung: Die beiden Konstanten T und F stehen für die beiden Wahrheitswerte ,,wahr" bzw. ,,falsch", $\neg$ steht für die Negation, $\wedge$ und $\vee$ stehen für Konjunktion bzw. die Disjunktion, und $\forall$ und $\exists$ stehen für den All– bzw. Existenzquantor.

Die drei Konstanten One_One, Onto und Type_Definition werden bei der Einführung neuer Typen benötigt. Die Konstante One_One steht für die Injektivität und die Konstante Onto für die Surjektivität. Auf die Bedeutung der Funktion Type_Definition wird später eingegangen werden.

Es sind ein paar Besonderheiten in der Schreibweise zu beachten: $\Rightarrow$, $=$, $\vee$ und $\wedge$ werden in Infix-Schreibweise verwendet. Statt $\forall(\lambda x.\ldots)$ darf auch $\forall x.\ldots$ und statt

$\forall x_1.(\forall x_2.(\forall x_3.\ \ldots))$ darf auch $\forall x_1\, x_2\, x_3.\ \ldots$ geschrieben werden. Entsprechendes gilt auch für den Existenzquantor $\exists$ und den Hilbert-Operator ε.

Konstanten mit einer Schreibweise wie bei $\forall$, $\exists$ und ε werden als ,,Binder" bezeichnet. Bei ihnen darf, wenn sie auf einen λ–Term angewandt werden, das λ–Symbol weggelassen werden, und bei direkt ineinander geschachtelten Bindern dürfen die inneren Konstantennennungen weggelassen werden. Zu jeder in HOL definierten Konstante wird festgelegt, ob sie Infix–Status und oder oder Binder–Status haben soll oder nicht.

Diese Besonderheiten in der Schreibweise führen dazu, daß die HOL–Terme in der allgemein üblichen Weise notiert werden können. Es ist aber zu beachten, daß sich beispielsweise hinter $a = b$ der Term $((= a)b)$ und hinter $\exists x.p$ der Term $\exists(\lambda x.p)$ verbirgt.

3.1.3 Der Kalkül von HOL

Axiome sind Theoreme. Sie entstehen dadurch, daß eine beliebige Formel angegeben wird, die per Definition allgemeingültig sein soll. Damit ist diese Formel als Theorem verfügbar und es können aus diesem Theorem weitere Theoreme abgeleitet werden. In HOL gibt es die folgenden fünf Axiome:

BOOL_CASES_AX	$\vdash\ \forall b.\,(b = \mathsf{T})\ \vee\ (b = \mathsf{F})$
IMP_ANTISYM_AX	$\vdash\ \forall b_1\, b_2.\,(b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_2 = b_1)$
ETA_AX	$\vdash\ \forall f_{\alpha\to\beta}.\,(\lambda x.\, f\ x) = f$
SELECT_AX	$\vdash\ \forall P_{\alpha\to bool}.\, P\ x \Rightarrow P(\varepsilon\, P)$
INFINITY_AX	$\vdash\ \exists f_{ind\to ind}.\,\mathsf{One\_One}\ f \wedge \neg(\mathsf{Onto}\ f)$

Man kann jederzeit weitere Axiome hinzufügen; die Konsistenz des Systems kann dabei allerdings verlogengehen. In dieser Arbeit werden daher keine weiteren Axiome eingeführt.

Regeln beschreiben Mechanismen, mit denen neue Theoreme abgeleitet werden können. Beispielsweise bildet die Regel ASSUME jede Formel t auf das Theorem $t \vdash t$ ab. Andere Regeln setzen bereits bewiesene Theoreme voraus. In der nachfolgenden schematischen Schreibweise der Regeln sind die vorausgesetzten Theoreme und das daraus abgeleitete Theorem in der Form eines Bruches dargestellt: die vorausgesetzten Theorem befinden sich im ,,Zähler", das abgeleitete Theorem im ,,Nenner".

ASSUME	$\dfrac{}{t \vdash t}$
REFL	$\dfrac{}{\vdash t = t}$
BETA_CONV	$\dfrac{}{\vdash (\lambda x.\, t_1)t_2 = t_1[t_2/x]}$
SUBST	$\dfrac{\Gamma_1 \vdash t_1 = t_1' \ \cdots\ \Gamma_n \vdash t_n = t_n' \quad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1', \ldots, t_n']}$

ABS $$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\, t_1) = (\lambda x.\, t_2)}$$

INST_TYPE $$\frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n / \alpha_1, \ldots \alpha_n]}$$

DISCH $$\frac{\Gamma \vdash t_2}{\Gamma \setminus \{t_1\} \vdash t_1 \Rightarrow t_2}$$

MP $$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

Für diese Regeln gelten folgende Einschränkungen:

- In BETA_CONV und SUBST muß durch geeignete Variablenumbenennungen verhindert werden, daß durch die Substitution freie Variablen gebunden werden.
- BETA_CONV ist nur zulässig, falls x nicht frei in Γ vorkommt.
- Bei der Regel INST_TYPE dürfen die Typenvariablen $\alpha_1, \ldots, \alpha_n$ nicht in Γ vorkommen, und es muß verhindert werden, daß zwei verschiedene Variablen, die sich nicht im Namen, sondern nur im Typ unterscheiden, durch die Typensubstitution ,,gleichgemacht" werden.

3.1.4 Standardtheorien

Die bisher beschriebene Sprache wird als die Theorie INIT bezeichnet. INIT ist bewiesenermaßen konsistent. Der Beweis der Konsistenz wurde dadurch erbracht werden, daß ein Modell — das sogenannte *Standardmodell* — angegeben wurde.

In HOL kann die Theorie INIT so erweitert werden, daß die Konsistenz nicht verloren geht. Dazu gibt es vier Mechanismen:

1. Konstantendefinition
2. Konstantenspezifikation
3. Typendefinition
4. Typenspezifikation

Der Begriff *Standardtheorie* bezeichnet all jene Theorien, die durch die Anwendung dieser Mechanismen aus der Theorie INIT abgeleitet wurden. Standardtheorien sind konsistent. Das kann man dadurch beweisen, daß man zeigt, wie das Standardmodell erweitert werden kann, wenn die Standardtheorie durch einen der Mechanismen erweitert wird.

Die Erweiterung durch Konstantendefinition wurde bereits vorgestellt. Die Erweiterung durch Konstantenspezifikation ist ein allgemeinerer Mechanismus, bei dem zum einen mehrere Konstanten gleichzeitig eingeführt werden können und zum anderen die Formel diese Konstanten nicht in eindeutiger Weise beschreiben muß — es muß lediglich bewiesen werden, daß derartige Werte existieren.

Zur Erweiterung einer Theorie um einen Typ gibt es ebenfalls zwei Möglichkeiten: Typendefinition un Typenspezifikation. Durch eine Typendefinition wird ein Typ in eindeutiger Weise auf eine fest vorgegebene nichtleere Teilmenge eines bereits bestehenden Typs zurückgeführt. Bei einer Typenspezifikation wird der neu eingeführte Typ durch eine allgemeinere Eigenschaft beschrieben. Wie schon bei der Konstantenspezifikation, so muß auch bei der Typenspezifikation ein Beweis erbracht werden, durch den abgesichert wird, daß die Typenspezifikation auch tatsächlich konsistenzerhaltend ist.

Die Mechanismen Konstantenspezifikation und Typendefinition werden in den nächsten beiden Abschnitten erläutert. Da in dieser Arbeit lediglich die drei Mechanismen Konstantendefinition, Konstantenspezifikation und Typendefinition zur Anwendung kommen werden, soll auf den Typenspezifikationsmechanismus nicht näher eingegangen werden. Der interessierte Leser sei auf [HOL93] verwiesen.

3.1.5 Konstantenspezifikation

Bei einer Konstantenspezifikation werden mehrere Konstanten $c_1, c_2, \ldots c_n$ und ein Theorem

$$\vdash p(c_1, c_2, \ldots c_n)$$

eingeführt, das diese Konstanten beschreibt. Das Prädikat p kann dabei jedoch nicht beliebig gewählt werden. Ein ,,ungünstig" gewähltes p könnte dazu führen, daß die Einführung des Theorems $\vdash p(c_1, c_2, \ldots c_n)$ die Konsistenz verletzt. Das Prädikat p könnte so gewählt sein, daß $p(c_1, c_2, \ldots c_n)$ widersprüchlich ist oder daß $\vdash p(c_1, c_2, \ldots c_n)$ zu einer Inkonsistenz mit den in der bisherigen Theorie ableitbaren Theoremen führt. Es wird deshalb bei einer Konstantenspezifikation vorausgesetzt, daß das Theorem

$$\vdash \exists c_1, c_2, \ldots c_n . p(c_1, c_2, \ldots c_n)$$

bewiesen wurde.

Zur Veranschaulichung dieser Forderung beachte man, daß aus dem neu einzuführenden Theorem $\vdash p(c_1, \ldots c_n)$ das Theorem $\vdash \exists c1, \ldots c_n . p(c1, \ldots c_n)$ abgeleitet werden kann. Da vor einer Konstantenspezifikation der Beweis von $\vdash \exists c1, \ldots c_n . p(c1, \ldots c_n)$ gefordert wird, kann sichergestellt werden, daß aus der bisherigen Theorie das Theorem

$$\vdash \neg(\exists c1, \ldots c_n . p(c1, \ldots c_n))$$

nicht ableitbar ist, welches zusammen mit dem neu einzuführenden Theorem $\vdash p(c1, \ldots c_n)$ zu einem Widerspruch führen würde. Damit ist natürlich lediglich ausgesagt, daß die Forderung dieses Beweises mindestens notwendig ist, um die Konsistenz zu erhalten und nicht, daß diese Forderung auch die Konsistenzerhaltung des Konstantenspezifikationsmechanismuses garantiert. Die Erhaltung der Konsistenz bei einer Konstantenspezifikation kann durch eine entsprechende Erweiterung des Standardmodells nachgewiesen werden.

3.1.6 Typendefinitionen

Bei einer Typendefinition wird ein in der bisherigen Theorie noch nicht definierter, neuer Typ *newtype* eingeführt und wie bei der Konstantendefinition und der Konstantenspezifikation wird gleichzeitig auch ein Axiom eingeführt, das die Eigenschaften des neuen Typs beschreibt.

Um diesen Typ zu beschreiben, muß zunächst ein bereits definierter Typ *oldtype* ausgewählt werden. Als Grundelemente sind in INIT bereits die atomaren Typen *bool* und *ind* definiert. Zusammen mit dem ebenfalls in INIT definierten zweistelligen Typoperator $\rightarrow$, mit dem aus zwei bereits definierten Typen wieder ein Typ gebildet werden kann, ergeben sich bereits in INIT zahlreiche Möglichkeiten zur Auswahl eines Typs *oldtype*. Alle abgeleiteten Typen bauen auf *bool*, *ind* und $\rightarrow$ auf. Bei einer Typendefinition muß der Typ, auf den der neue Typ zurückgeführt werden soll jedoch nicht unbedingt nur aus *bool*, *ind* und $\rightarrow$ aufgebaut sein, sondern es können auch solche Typen verwendet werden, die selbst durch Typendefinition oder Typenspezifikation abgeleitet wurden.

Nachdem ein geeigneter Typ ausgewählt wurde, wird ein Prädikat $P_{oldtype \rightarrow bool}$ festgelegt, das eine nichtleere Teilmenge $\{x|P\,x\}$ von *oldtype* beschreibt. Der neu einzuführende Typ soll in einer Isomorphiebeziehung zu dieser Teilmenge stehen. Die Forderung daß diese Teilmenge nichtleer sein muß, gründet auf der Forderung, daß der neue Typ — wie alle Typen in HOL — eine nichtleere Menge darstellen soll. Der Beweis, daß diese Menge nichtleer ist, muß explizit erbracht werden. Daher muß zunächst das folgende Theorem gezeigt werden:

$$\vdash\ \exists x_{oldtype}.\ P\ x$$

Schließlich wird eine Isomorphiebeziehung zwischen $newtype$ und $\{x|P\,x\}$ aufgebaut. Diese Beziehung wird durch ein Theorem beschrieben, das die Aussage macht, daß es eine injektive Funktion $f_{newtype \rightarrow oldtype}$ gibt, deren Wertebereich $\{x|P\ x\}$ ist. Abbildung 3.1 veranschaulicht diesen Zusammenhang.

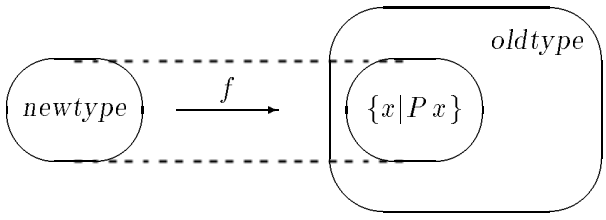

Abbildung 3.1: Typendefinition

Zur Unterstützung von Typendefinitionen gibt es in der Theorie INIT eine Konstantendefinition, die bisher noch nicht erläutert wurde:

$$\vdash\ \mathsf{Type\_Definition} =$$
$$\lambda\, P_{\alpha \rightarrow bool}\ REP_{\beta \rightarrow \alpha}.\ \mathsf{One\_One}\ REP \wedge (\forall x.\ P\ x = (\exists y.\ x = REP\ y))$$

Der Term (One_One REP) steht für die Aussage, daß REP injektiv ist und $\forall x.\, P\, x = (\exists y.\, x = REP\, y)$ besagt, daß $\{x|P\, x\}$ die Bildmenge von REP ist. Die in Abbildung 3.1 skizzierte Isomorphiebeziehung zwischen $\{x|P\, x\}$ und *newtype* wird durch das Theorem newtype_DEF beschrieben:

$$\text{newtype\_DEF} \quad \vdash \exists f_{newtype \to oldtype}.\ \text{Type\_definition}\ P\ f$$

Die Typendefinition ist damit abgeschlossen. Es kann gezeigt werden, daß die Konsistenz durch die gleichzeitige Erweiterung um einen Typ *newtype* und die gleichzeitige Einführung von newtype_DEF nicht verletzt wird.

In den nachfolgenden beiden Schritten werden noch zwei Konstanten mit den Namen REP_newtype und ABS_newtype eingeführt, die nützlich für die Verwendung des neuen Typs sind. Diese Konstanten werden bei der Definition eines neuen Typs im HOL–Beweiser automatisch durchgeführt. Sie sind für die Charakterisierung des neuen Typs jedoch nicht notwendig — sie haben rein technische Bedeutung.

Da durch newtype_DEF die Existenz eines Isomorphismus postuliert wird, ist es legitim, eine Konstante REP_newtype einzuführen, die gerade die Eigenschaft hat, eine solche Funktion zu sein. Das geschieht durch die folgende Konstantenspezifikation:

$$\vdash \text{Type\_definition}\ P\ \text{REP\_newtype}$$

Es soll besonders darauf hingewiesen werden, daß diese Konstantenspezifikation nur deshalb zulässig war, weil die Existenz eines solchen Wertes bewiesen werden kann und zwar deshalb, weil durch die Typendefinition das Theorem newtype_DEF eingeführt wurde.

Als nächstes wird zu der Funktion $\text{REP\_newtype}_{newtype \to oldtype}$ eine ,,inverse" Funktion $\text{ABS\_newtype}_{oldtype \to newtype}$ eingeführt. Auch das geschieht durch eine Konstantenspezifikation.

$$\vdash \forall x.\, \text{ABS\_newtype}(\text{REP\_newtype}\ x) = x$$

Auch bei dieser Konstantenspezifikation muß die Existenz einer derartig beschriebenen Konstanten ABS_newtype erst nachgewiesen werden. Gezeigt werden muß:

$$\vdash \exists f. \forall x.\, \text{ABS\_newtype}(f\ x) = x$$

Dieses Theorem kann aus dem Theorem, das bei der Konstantenspezifikation von REP_newtype entstanden ist, abgeleitet werden. Aus diesem Theorem folgt, daß die Funktion REP_newtype injektiv ist, und aus der Injektivität kann die Existenz der Funktion ABS_newtype abgeleitet werden. Auf den detaillierten Beweis soll hier verzichtet werden.

Abbildung 3.2 skizziert den Zusammenhang zwischen den Konstanten REP_newtype und ABS_newtype. Zu beachten ist, daß ABS_newtype eine totale Funktion von *oldtype* nach *newtype* ist und daß keine Aussage über die Funktionswerte von ABS_newtype außerhalb von $\{x|P\, x\}$ gemacht wurde. ABS_newtype ist deshalb i.a. nicht eindeutig definiert. Man beachte auch, daß die Funktionen REP_newtype und ABS_newtype nicht wirklich invers zueinander sind. REP_newtype ist injektiv aber i.a. nicht surjektiv und ABS_newtype ist surjektiv aber i.a. nicht injektiv. Die erwähnte Isomorphie besteht nicht zwischen *newtype* und *oldtype*, sondern lediglich zwischen *newtype* und $\{x|P\, x\}$.

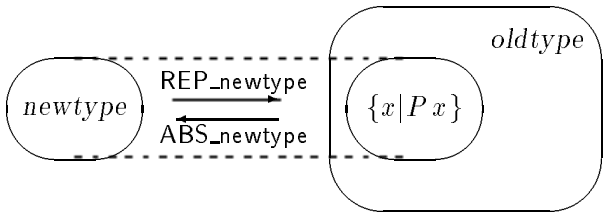

Abbildung 3.2: Die Funktion REP_newtype und ABS_newtype

3.2 PML-Datentypen

Das im folgenden zur Formalisierung der Semantik von PML–Programmen beschriebene Schema basiert auf den Grundmechanismen zur konsistenzerhaltenden Erweiterung von Theorien. Jedes PML–Konstrukt, Datentypdeklaration, Funktionen– und Konstantendefinition, wird in HOL durch Konstantendefinitionen, Konstantenspezifikationen und Typdefinitionen formal beschrieben. Die Typen in HOL entsprechen dabei den Typen in PML, die Konstanten in HOL stehen für die Konstanten und Funktionen in PML und die Theoreme beschreiben die Semantik. Verknüpft man die einzelnen dabei entstehenden Theoreme konjunktiv, so erhält man ein Theorem, das die Semantik des Gesamtprogramms beschreibt.

Die Formalisierung eines PML–Konstruktes teilt sich in zwei Teilaufgaben: Zum einen muß festgelegt werden, wie die jeweilige Formel aussieht, die ein PML–Konstrukt beschreibt, und zum anderen muß gezeigt werden, wie diese Formel als Theorem in konsistenzerhaltender Weise eingeführt werden kann. Für die formale Festlegung der Semantik der Sprache PML ist eigentlich nur der erste Punkt von Belang. Da die Konsistenz erhalten bleibt, ist jedoch auch gewährleistet, daß die Semantik nicht widersprüchlich ist.

Zur Beschreibung der Semantik der PML–Datentypen wird ein Verfahren zur Typdefinition verwendet, das von T.F. Melham entwickelt wurde [Melh88]. Das Verfahren erläutert, wie man zu einer Datentypbeschreibung eine entsprechende Umsetzung in HOL findet. Eine Implementierung des Verfahrens findet man im *Type Definition Package* von HOL. Die zentrale Funktion ist dabei die Funktion new_type, die zu einer vorgegebenen Datentypdeklaration die aktuelle Theorie um den entsprechenden Typ, seine Konstruktoren und seine Semantik erweitert. Die Datentypdeklaration wird dabei in Form einer Zeichenkette als Parameter angegeben. Die in PML verwendeten Datentypen lehnen sich an dieses Verfahren an. Bei der primitive_datatype–Datentypdeklaration in PML wurde gerade die Syntax dieser Zeichenkette übernommen.

In diesem Abschnitt wird erläutert, wie die Semantik der PML-Datentypen in HOL durch Theoreme ausgedrückt wird. Dieser Aspekt ist relevant für die Semantik der Sprache PML. Das Verfahren, mit dem Melham diese Theoreme in konsistenzerhaltender Weise einführt, ist recht komplex und kann hier nur skizziert werden. Für eine genaue Beschreibung sei auf [HOL93] und insbesondere auch auf [Melh88] verwiesen.

3.2.1 Die Semantik der PML-Datentypdeklaration

Jeder Datentyp wird in HOL durch eine Typenkonstante ϑ repräsentiert. Der Begriff Typenkonstante wird in HOL als Oberbegriff für atomare Typen (nullstellige Typenkonstanten) und Typenoperatoren (mehrstellige Typenkonstanten) verwendet. Die Stelligkeit von ϑ sei mit $|\vartheta|$ bezeichnet. Man erhält die Stelligkeit eines Typenoperators als die Anzahl der in der Datentypdeklaration in PML vorkommenden Typenvariablen. Beispielsweise sind *bool* und *num* atomare Typen, *list* und *partial* sind einstellige Typenoperatoren und *prod* ist ein zweistelliger Typenoperator.

Mit den Typenvariablen α^1, $\alpha^2, \ldots \alpha^{|\vartheta|}$ wird für jede Typenkonstante ϑ der Typ

$$\widetilde{\vartheta} \;=\; (\alpha^1,\, \alpha^2, \ldots \alpha^{|\vartheta|})\, \vartheta$$

gebildet. Jedem Datentyp ist eine Menge von Konstruktoren zugeordnet. Diese Konstruktoren werden in HOL durch Konstanten dargestellt. Die Anzahl der Konstruktoren sei c und die Menge der Konstruktoren sei

$$\{\gamma^1,\, \gamma^2, \ldots \gamma^c\}$$

Jeder Konstruktor γ^i soll in Curry–Form sein und der Funktionswert soll vom Typ $\widetilde{\vartheta}$ sein. Die Stelligkeit des Konstruktors γ^i sei mit $|\gamma^i|$ bezeichnet. Der Typ von γ^i hat die folgende Form:

$$\tau^{i,1} \rightarrow \tau^{i,2} \rightarrow \ldots \rightarrow \tau^{i,|\gamma^i|} \rightarrow \widetilde{\vartheta}$$

Die Namen der Konstruktoren, ihre Stelligkeit und die Typen ihrer Argumente können direkt aus der Datentypdeklaration entnommen werden. Wie bereits in Kapitel 2 erwähnt, dürfen die Typen der Argumente $\tau^{i,j}$ nicht beliebig sein. Als Typ ist nur erlaubt:

1. ein bereits definierter, atomarer Typ
2. eine der Typvariablen $\alpha^1, \alpha^2, \ldots \alpha^{|\vartheta|}$
3. der Typ $\widetilde{\vartheta}$ selbst.

Gibt es für einen Datentyp ϑ einen Konstruktor γ^i, bei dem mindestens ein Argument den Typ $\widetilde{\vartheta}$ hat, dann handelt es sich um einen *rekursiven* Datentyp. Beispiele für rekursive Typen sind *num* und *list*. Dagegen sind *bool*, *partial* und *prod* nichtrekursiv. Den Argumenten vom Typ $\widetilde{\vartheta}$ kommt im folgenden eine besondere Bedeutung zu. Zu einem Konstruktor γ^i sei r_i die Anzahl der Argumente vom Typ $\widetilde{\vartheta}$. Mit $\rho_{i,1}, \rho_{i,2}, \ldots \rho_{i,r_i}$ seien, in aufsteigender Reihenfolge, die Indizes k mit $\tau^{i,k} = \widetilde{\vartheta}$ bezeichnet.

Das Theorem $\mathcal{D}(\vartheta)$, das die Semantik des Datentyps ϑ beschreibt, ist wie folgt definiert:

$$\begin{aligned}\mathcal{D}(\vartheta) \;\vdash\; & \forall\, h_1 h_2 \ldots h_c\,.\;\; \exists_1\, g\,. \\ & \bigwedge_{i=1}^{c} \forall\, x_1 x_2 \ldots x_{|\gamma^i|}\,. \\ & \quad g\,(\gamma^i\, x_1 x_2 \ldots x_{|\gamma^i|}) \;=\; h_j\,(g\, x_{\rho_{i,1}})\,(g\, x_{\rho_{i,2}}) \;\ldots\; (g\, x_{\rho_{i,r_i}})\, x_1 x_2 \ldots x_{|\gamma^i|}\end{aligned}$$

Das Theorem $\mathcal{D}(\vartheta)$ besagt im wesentlichen, daß die primitive Rekursion über ϑ eindeutig ist. Daß die Semantik eines Datentyps auf diese Weise definiert wird, mag verblüffen, und es ist auch sicher nicht sofort einsichtig, wie man mit Hilfe der vorgestellten Mechanismen eine Theorie in konsistenzerhaltender Weise um dieses Theorem erweitern kann.

Beispiel

Die Semantik der Datentypen soll am Beispiel des Datentyps *num* erläutert werden. Die Datentypdeklaration von *num* lautet:

```
datatype  " num = Zero | Suc num " ;
```

In der Datentypdeklaration kommen keine Typvariablen vor. Damit wird *num* in HOL durch einen atomaren Typ (eine nullstellige Typenkonstante) repräsentiert. Es gilt:

$$\begin{aligned} |num| &= 0 \\ \widetilde{num} &= num \end{aligned}$$

Es wurden für *num* die beiden Konstruktoren Zero und Suc festgelegt. Der Konstruktor Zero hat keine Argumente und damit ist sein Typ $\widetilde{num}$ also *num*. Der Konstruktor Suc hat genau ein Argument und dieses hat den Typ *num*. Der Typ von Suc ist damit $num \to num$. Es gilt:

$$\begin{aligned} c &= 2 \\ \gamma^1 &= \mathsf{Zero} \\ \gamma^2 &= \mathsf{Suc} \\ |\mathsf{Zero}| &= 0 \\ |\mathsf{Suc}| &= 1 \\ \tau^{2,1} &= num \end{aligned}$$

Zero hat keine Argumente also auch keine vom Typ $\widetilde{num}$. Suc dagegen hat genau ein Argument und dieses hat den Typ *num*. Damit folgt nun, daß *num* rekursiv ist. Es gilt:

$$\begin{aligned} r_1 &= 0 \\ r_2 &= 1 \\ \rho_{2,1} &= 1 \end{aligned}$$

Die Semantik der Datentypdeklaration $\mathcal{D}(num)$ kann jetzt aus der allgemeinen Form $\mathcal{D}(\vartheta)$ abgeleitet werden:

$$\mathcal{D}(num) \ \vdash \ \forall a\, f.\, \exists_1 g.\, (g\ \mathsf{Zero} = a) \ \wedge \ \forall n.(g(\mathsf{Suc}\, n) = f\ (g\ n)\ n)$$

Die Aussage dieser Formel wird deutlich, wenn man die allgemeine Form einer primitiven Rekursion über dem Typ *num* betrachten:

$$\begin{aligned} g\ \mathsf{Zero} &= a \\ g\ (\mathsf{Suc}\, n) &= f\ (g\ n)\ n \end{aligned}$$

Durch dieses Gleichungssystem wird für gegebene a und f die Funktion g eindeutig definiert — und gerade das ist die Aussage von $\mathcal{D}(num)$!

Man wird zur Beschreibung der Semantik eher Aussagen in Form der Peano–Axiome wie

$$\vdash\ n = \mathsf{Zero} \vee \exists m.\, n = \mathsf{Suc}\, m$$
$$\vdash\ \neg(\mathsf{Zero} = \mathsf{Suc}\, n)$$
$$\vdash\ (\mathsf{Suc}\, n = \mathsf{Suc}\, m) \Rightarrow (n = m)$$

erwartet haben. In [Melh88] wird erläutert, daß durch $\mathcal{D}(num)$ tatsächlich der Datentyp in der gewünschten Weise charakterisiert wird und es wird gezeigt, wie derartige Eigenschaften aus $\mathcal{D}(num)$ abgeleitet werden können.

3.2.2 Die Semantik der vordefinierten Datentypen

Wie bereits in Kapitel 2 erwähnt, sind in PML die Datentypen *bool*, *prod*, *list*, *num* und *partial* bereits vordefiniert.

```
primitive_datatype  " bool    = T | F " ;
primitive_datatype  " prod    = Comma of 'a # 'b " ;
primitive_datatype  " list    = Nil | Cons of 'a # list " ;
primitive_datatype  " num     = Zero | Suc of num " ;
primitive_datatype  " partial = Defined 'a | Undefined" ;
```

Die Semantik dieser Typen wird durch die Theoreme $\mathcal{D}(bool)$, $\mathcal{D}(prod)$, $\mathcal{D}(list)$, $\mathcal{D}(num)$ und $\mathcal{D}(partial)$ beschrieben:

$$\mathcal{D}(bool) \quad \vdash \forall a\, b.\, \exists_1 g.\, (g\, \mathsf{T} = a) \wedge (g\, \mathsf{F} = b)$$
$$\mathcal{D}(prod) \quad \vdash \forall f.\, \exists_1 g.\, \forall m\, n.\, g(\mathsf{Comma}\, m\, n) = f\, m\, n$$
$$\mathcal{D}(list) \quad \vdash \forall a\, f.\, \exists_1 g.\, (g\, \mathsf{Nil} = a) \wedge (\forall h\, t.\, g\, (\mathsf{Cons}\, h\, t) = f\, (g\, t)\, h\, t)$$
$$\mathcal{D}(num) \quad \vdash \forall a\, f.\, \exists_1 g.\, (g\, \mathsf{Zero} = a) \wedge (\forall n.\, g\, (\mathsf{Suc}\, n) = f\, (g\, n)\, n)$$
$$\mathcal{D}(partial) \vdash \forall a\, f.\, \exists_1 g.\, (g\, \mathsf{Undefined} = a) \wedge (\forall n.g\, (\mathsf{Defined}\, n) = f\, n)$$

3.2.3 Datentypdefinitionen in HOL

Bisher wurde nur erläutert, wie die Theoreme aussehen, die die Semantik der PML–Datentypen beschreiben. In diesem Abschnitt soll skizziert werden, wie eine Datentypdefinition in HOL in konsistenzerhaltender Weise durchgeführt werden kann. Eine Datentypdefinition ist ein Mechanismus, der sich aus mehreren Konstantendefinitionen, Konstantenspezifikationen und Typendefinitionen zusammensetzt. Bei einer Datentypdefinition werden gleichzeitig eine neue Typenkonstante ϑ, mehrere Konstanten γ^1, $\gamma^2, \ldots \gamma^c$ und ein die Semantik des Datentyps beschreibendes Theorem $\mathcal{D}(\vartheta)$ eingeführt. Die nachfolgende Darstellung dieses Verfahrens ist sehr grob gehalten. Für eine detaillierte Beschreibung sei auf [Melh88] verwiesen.

Der Typ $(\alpha)Tree$

Am Anfang einer Datentypdefinition steht eine Typendefinition. Für eine Typendefinition wird immer ein bereits existierender Typ *oldtype* benötigt, auf den der neue Typ zurückgeführt wird. Für den Datentypdeklarationsmechanismus wurde in HOL ein spezieller Typ namens $(\alpha)Tree$ eingeführt. $(\alpha)Tree$ steht für einen Baum, bei dem sich jeder Knoten $\mathsf{Node}(a, b)$ aus dem Inhalt des Knotens a vom Typ α und einer Liste von Unterbäumen b zusammensetzt.

Bemerkung: In der Syntax von ML könnte man einen Datentyp dieser Art durch die folgende Typdeklaration beschreiben:

```
datatype 'a Tree = Node of ('a * (('a Tree) list));
```

Der Typ $(\alpha)Tree$ ist mächtiger als alle Datentypen, die durch Datentypdeklarationen nach diesem Mechanismus entstehen können. Auf ihm bauen alle Datentypdefinitionen auf. Der Typ *oldtype* ist eine Ausprägung von Tree. *oldtype* entsteht durch Typeninstanziierung aus $(\alpha)Tree$, indem α durch einen geeigneten Typ τ substituiert wird. τ hängt nur von der Datentypdeklaration des neuen Datentyps ϑ ab und wird direkt aus ihm berechnet. In τ kommen der atomare Typ *one* (eine einelementige Menge) und die Typenoperatoren $\times$ (kartesisches Mengenprodukt) und $+$ (Mengenvereinigung) vor.

Bevor eine Datentypdefinition durchgeführt werden kann, müssen die Typen $(\alpha)Tree$, Node, *one*, $\times$ und $+$ bereits mit den jeweiligen Konstruktoren und einer zugehörigen Semantik eingeführt worden sein. In der Implementierung von HOL ist deshalb die Standardtheorie INIT bereits um diese Konstrukte erweitert worden. Bei dieser Erweiterung mußte natürlich auch wieder auf die Grundmechanismen Typendefinition, Konstantendefinition und Konstantenspezifikation zurückgegriffen werden. Diese Schritte sind manuell ausgeführt worden. Wie dies im einzelnen geschehen ist, soll hier nicht erläutert werden.

Das Teilmengenprädikat $P_{oldtype \to bool}$

Um eine Typendefinition durchführen zu können, wird dann aus der Datentypdeklaration ein Teilmengenprädikat $P_{oldtype \to bool}$ berechnet, und der Beweis für das Theorem

$$\vdash \exists x_{oldtype}.\ P\ x$$

wird automatisch abgeleitet. Jetzt kann die Datentypdefinition für ϑ durchgeführt werden.

Die Definition der Konstruktoren

Auf dem Typ *oldtype* ist bereits der Konstruktor Node definiert, und auch für die Typen $\times$, $+$ und *one* sind bereits entsprechende Konstruktoren definiert. Bei der Typendefinition von ϑ entstehen die beiden Konstanten REP_ϑ und ABS_ϑ. Die Konstruktoren des neuen Datentyps entstehen durch Konstantendefinitionen, wobei mit Hilfe von REP_ϑ und ABS_ϑ die Konstruktor des Datentyps ϑ auf Node zurückgeführt werden.

3.3 Primitive Rekursion über PML–Datentypen

Im letzten Abschnitt wurde gezeigt, wie zu einer vorgegebenen Datentypdeklaration die Semantik des Datentyps definiert werden kann. In diesem Abschnitt werden nun zwei Grundfunktionen zu diesem Datentyp definiert. Von dem Datentyp sind wieder gegeben:

ϑ	die Typenkonstante mit der Stelligkeit $\|\vartheta\|$
γ^i	die Konstruktoren mit den Stelligkeiten $\|\gamma^i\|$
$\tau^{i,j}$	die Typen der Argumente der Konstruktoren
r_i	die Anzahl der Argumente von γ^i mit dem Typ ϑ
$\rho_{i,j}$	die Positionen dieser Argumente

Die Funktion PRIMREC_ϑ wird durch das folgende Theorem beschrieben:

$$\begin{array}{l}\vdash\ \forall\, h_1 h_2 \ldots h_c\,.\\ \quad \bigwedge_{i=1}^{c} \forall\, x_1 x_2 \ldots x_{|\gamma^i|}\,.\\ \qquad \mathsf{PRIMREC\_}\vartheta\ h_1 \ldots h_c\ (\gamma^i\ x_1 x_2 \ldots x_{|\gamma^i|}) \ =\\ \qquad h_i\ x_1 x_2 \ldots x_{|\gamma^i|}\,(\mathsf{PRIMREC\_}\vartheta\ h_1 \ldots h_c\ x_{\rho_{i,1}}) \ldots (\mathsf{PRIMREC\_}\vartheta\ h_1 \ldots h_c\ x_{\rho_{i,r_i}})\end{array}$$

Beweis

In HOL wird PRIMREC_ϑ durch Konstantenspezifikation eingeführt. Dazu muß das folgende Theorem bewiesen werden.

$$\begin{array}{l}\vdash\ \exists\, g\,.\ \forall\, h_1 h_2 \ldots h_c\,.\\ \quad \bigwedge_{i=1}^{c} \forall\, x_1 x_2 \ldots x_{|\gamma^i|}\,.\\ \qquad g\ h_1 \ldots h_c\ (\gamma^i\ x_1 x_2 \ldots x_{|\gamma^i|}) \ =\\ \qquad h_i\ x_1 x_2 \ldots x_{|\gamma^i|}\,(g\ h_1 \ldots h_c\ x_{\rho_{i,1}}) \ldots (g\ h_1 \ldots h_c\ x_{\rho_{i,r_i}})\end{array}$$

Dieses Theorem kann aus $\mathcal{D}(\vartheta)$ abgeleitet werden: Als erstes muß die Parameterreihenfolge in $\mathcal{D}(\vartheta)$ umgestellt werden. Dazu wird in $\mathcal{D}(\vartheta)$ jede Variable h_j durch

$$(\lambda\, x_1 x_2 \ldots x_{|\gamma^i|}\, y_1\, y_2 \ldots y_{r_j}\,.\, h_j\, y_1\, y_2 \ldots y_{r_j}\, x_1 x_2 \ldots x_{|\gamma^i|})$$

spezialisiert. Dann werden die Variablen $h_1 h_2 \ldots h_c$ allquantifiziert, und es wird schließlich mit β-Konversion vereinfacht. Dadurch kommt man zu:

$$\begin{array}{l}\vdash\ \forall\, h_1 h_2 \ldots h_c\,.\ \exists_1\, g\,.\\ \quad \bigwedge_{i=1}^{c} \forall\, x_1 x_2 \ldots x_{|\gamma^i|}\,.\\ \qquad g\ (\gamma^i\ x_1 x_2 \ldots x_{|\gamma^i|}) \ =\ h_i\ x_1 x_2 \ldots x_{|\gamma^i|}\ (g\ x_{\rho_{i,1}})\ (g\ x_{\rho_{i,2}})\ \ldots\ (g\ x_{\rho_{i,r_i}})\end{array}$$

Dann wird die folgende Beziehung[1] benutzt:

$$\vdash\ (\forall x.\, \exists_1 y.\, P\, x\, y) \ =\ (\exists_1 y.\, \forall x.\, P\, x\, (y\, x))$$

[1] Wäre in diesem Theorem jedes $\exists_1$–Symbol durch ein $\exists$–Symbol ersetzt, so würde das Theorem gerade das Schema einer gewöhnlichen Skolemisierung beschreiben. Das Theorem sagt aus, daß die Skolemisierung für die eindeutige Existenz $\exists_1$ zulässig ist.

Wendet man diese Beziehung mehrfach auf das Theorem an, so kommt man zu:

$$\begin{array}{l}\vdash\ \exists_1 g\,.\ \forall h_1 h_2 \ldots h_c\,. \\ \quad \bigwedge_{i=1}^{c} \forall x_1 x_2 \ldots x_{|\gamma^i|}\,. \\ \qquad g\ h_1 \ldots h_c\ (\gamma^i\ x_1 x_2 \ldots x_{|\gamma^i|}) \ = \\ \qquad h_i\ x_1 x_2 \ldots x_{|\gamma^i|}\,(g\ h_1 \ldots h_c\ x_{\rho_{i,1}}) \ldots (g\ h_1 \ldots h_c\ x_{\rho_{i,r_i}})\end{array}$$

Aus der Definition der eindeutigen Existenz $\exists_1$ kann abgeleitet werden, daß die eindeutige Existenz $\exists_1$ die Existenz $\exists$ impliziert. Damit ist die Existenz einer solchen Funktion gesichert, wie es für die Definition von PRIMREC_ϑ erforderlich war. ■

Für die vordefinierten Datentypen *bool*, *prod*, *list*, *num* und *partial* ergeben sich folgende Definitionen:

$$\vdash\ \forall a\ b.\ \ (\mathsf{PRIMREC\_bool\ T}\ a\ b = a) \wedge (\mathsf{PRIMREC\_bool\ F}\ a\ b = b)$$

$$\vdash\ \forall f.\quad (\forall m\ n.\ \mathsf{PRIMREC\_prod\ (Comma}\ m\ n)\ f = f\ m\ n)$$

$$\vdash\ \forall a\ f.\ \ (\mathsf{PRIMREC\_list\ Nil}\ a\ f = a) \wedge (\forall h\ t.\ \mathsf{PRIMREC\_list\ (Cons}\ h\ t)\ a\ f = f\ (\mathsf{PRIMREC\_list}\ t\ a\ f)\ h\ t)$$

$$\vdash\ \forall a\ f.\ \ (\mathsf{PRIMREC\_num\ Zero}\ a\ f = a) \wedge (\forall n.\ \mathsf{PRIMREC\_num\ (Suc}\ n)\ a\ f = f\ (\mathsf{PRIMREC\_num}\ n\ a\ f)\ n)$$

$$\vdash\ \forall f\ a.\ \ (\forall x.\ \mathsf{PRIMREC\_partial\ (Defined}\ f\ a\ x) = f\ x) \wedge (\mathsf{PRIMREC\_partial\ Undefined}\ f\ a = a)$$

Ein wichtiger Spezialfall der primitiven Rekursion ist die Fallunterscheidung. Aus diesem Grund wurde in PML für die Fallunterscheidung eine separate Grundfunktion definiert. Die Semantik der Funktion CASE_ϑ erhält man durch Spezialisierung von PRIMREC_ϑ. Die Einführung von CASE_ϑ erfolgt durch eine Konstantendefinition:

$$\begin{array}{l}\vdash\ \mathsf{CASE\_}\vartheta\ = \\ \quad \lambda x\ h_1 \ldots h_c\,. \\ \qquad \mathsf{PRIMREC\_}\vartheta\ x \\ \qquad\quad (\lambda x_1 x_2 \ldots x_{|\gamma^i|}.\ h_1 x_{\rho_{1,1}}\ \ldots x_{\rho_{1,r_i}}) \\ \qquad\quad (\lambda x_1 x_2 \ldots x_{|\gamma^i|}.\ h_2 x_{\rho_{2,1}}\ \ldots x_{\rho_{2,r_i}}) \\ \qquad\qquad \vdots \\ \qquad\quad (\lambda x_1 x_2 \ldots x_{|\gamma^i|}.\ h_c\ x_{\rho_{c,1}} \ldots x_{\rho_{c,r_i}})\end{array}$$

Die CASE–Funktionen der vordefinierten Datentypen *bool*, *prod*, *list*, *num* und *partial* sind somit folgendermaßen definiert:

$$\vdash\ \forall a\ b\ x.\ \mathsf{CASE\_bool}\ x\ a\ b = \mathsf{PRIMREC\_bool}\ x\ a\ b$$

$$\vdash\ \forall\ f\ x.\ \mathsf{CASE\_prod}\ x\ h = \mathsf{PRIMREC\_prod}\ x\ (\lambda r\ s.\ h\ r\ s)$$

$$\vdash\ \forall a\ f\ x.\ \mathsf{CASE\_list}\ x\ a\ f = \mathsf{PRIMREC\_list}\ x\ a\ (\lambda r\ s\ t.\ f\ r\ s)$$

$\vdash \forall a\, f\, n.\, \mathsf{CASE\_num}\, n\, a\, f = \mathsf{PRIMREC\_num}\, n\, (\lambda r\, s.\, f\, r)$

$\vdash \forall f\, a\, x.\, \mathsf{CASE\_partial}\, x\, a\, f = \mathsf{PRIMREC\_partial}\, x\, (\lambda r.\, f\, r)\, a$

3.4 μ–Rekursion

3.4.1 Die Funktion WHILE

Zur Beschreibung μ-rekursiver Funktionen gibt es in PML eine zusätzliche Grundfunktion mit dem Namen WHILE. Die Funktionen WHILE wird durch die nachfolgenden Konstantendefinitionen[2] von iota, terminates, mu, power und WHILE beschrieben. Dabei sind iota, terminates, mu und power lediglich Hilfskonstruktionen, die die Definition von WHILE etwas anschaulicher machen sollen.

$\vdash$ iota f = CASE_bool $(\exists_1 f)$ (Defined(εf)) Undefined

$\vdash$ terminates $(f, n) = (f\, n) \wedge (\forall m.\, m < n \Rightarrow \neg(f\, m))$

$\vdash$ mu f = iota $(\lambda m.$ terminates$(f, m))$

$\vdash$ power $f\, n\, x =$
PRIMREC_num n (Defined x) $(\lambda a\, b.$ CASE_partial $b\, f$ Undefined)

$\vdash$ WHILE $g\, f\, x =$
CASE_partial
(mu $(\lambda n.$ CASE_partial (power $f\, n\, x$) $(\lambda y.$ PRIMREC_bool y F T) F))
$(\lambda n.$ power $f\, n\, x)$
Undefined

Die Funktion iota ähnelt dem Hilbert–Operator ε. In dem Fall, daß der Parameter (ein Prädikat $f_{\alpha\to bool}$) ein Element x_α eindeutig beschreibt ($\exists_1 f$ gilt), dann ist der Funktionswert $\varepsilon\, f =$ Defined x_α. Ansonsten hat iota den Wert Undefined. Zum Vergleich: der Hilbert–Operator ε hat in dem Fall, daß $f_{\alpha\to bool}$ ein Element x_α eindeutig bestimmt, den Funktionswert $\varepsilon f = x_\alpha$, in dem Fall, daß es kein x_α mit der Eigenschaft $f_{\alpha\to bool}$ gibt, einen beliebigen aber festen Wert vom Typ α und in dem Fall, daß es mehrere Werte mit der Eigenschaft $f_{\alpha\to bool}$ gibt, einen beliebigen aber festgelegten Wert mit dieser Eigenschaft.

Das Prädikat terminates hat zwei Parameter: $f_{num\to bool}$ und n_{num}. Es gibt an, ob n die kleinste natürliche Zahl ist, so daß $(f\, n)$ wahr wird.

Die Funktion mu ist eine leicht modifizierte Variante des μ–Operators. Sie berechnet ähnlich wie der μ–Operator das kleinste m, bei der die Funktion, die als Parameter angegeben wird, wahr wird. Im Gegensatz zum μ–Operator wird aber auch für den Fall, daß

[2] An dieser Stelle tritt eine allgemeinere Form von Konstantendefinitionen auf: Auf der linken Seite der Gleichung darf nicht nur eine Konstante stehen, sondern es dürfen auch Variablen als Parameter vorkommen und diese Variablen dürfen auch gepaart werden. Diese erweiterte Konstantendefinitionsmechanismus kann auf die ursprünglich eingeführte Form der Konstantendefinition zurückgeführt werden.

es kein solches m gibt, explizit ein Funktionswert, nämlich Undefined, angegeben, und in dem Fall, daß es ein solches m gibt, ist der Funktionswert nicht m wie beim μ–Operator, sondern (Defined m).

Im Gegensatz zu ML sollen in PML μ–rekursive Funktionen immer vom Typ $\alpha \rightarrow (\beta)partial$ sein. Die Funktion power potenziert eine Funktionen vom Typ $\alpha \rightarrow (\alpha)partial$. Der Term (power$fn$) steht für den mathematischen Ausdruck f^n, wobei jedoch zu beachten ist, daß der Typ von f nicht $\alpha \rightarrow \alpha$ sondern $\alpha \rightarrow (\alpha)partial$ ist. Damit hat auch (powerfn) den Typ $\alpha \rightarrow (\alpha)partial$ und nicht $\alpha \rightarrow \alpha$.

Die Funktion WHILE hat drei Parameter: $g_{\alpha \rightarrow bool}$, $f_{\alpha \rightarrow (\alpha)partial}$ und x_α. Sie berechnet den Term $f^n(x)$ mit dem kleinsten n, sodaß $g(f^n(x))$ wahr wird. Der Funktionswert ist (Defined (power f n x)) falls es ein solches n gibt und ansonsten Undefined.

3.4.2 Vergleich: WHILE und μ

Es wurde bereits gefordert, daß die Sprache PML Turing–vollständig sein soll, daß also alle berechenbaren Funktionen in PML ausgedrückt werden können. Um dies zu beweisen, soll gezeigt werden, daß mit PML–Programme beliebige μ–rekursive Funktionen dargestellt werden können. Die Menge der μ–rekursiven Funktionen ist Turing–vollständig. Üblicherweise kommt man von der primitiven Rekursion zur μ–Rekursion durch Einführung des μ–Operators. Es muß gezeigt werden, daß die Funktion WHILE ein gleichwertiger Ersatz für μ ist. Das wird dadurch bewiesen, daß μ unter Verwendung von WHILE ausgedrückt wird:

$$\vdash \ \mu \ = \ \lambda g.\, \mathsf{WHILE}\ g\ (\lambda n.\, \mathsf{Defined}(\mathsf{Suc}\ n))\ \mathsf{Zero}$$

Um den Übergang von den primitiv rekursiven zu den berechenbaren Funktionen zu realisieren, hätte man auch den μ–Operator als PML–Grundfunktion ergänzen können. Man könnte dann die Funktion WHILE als eine von μ abgeleitete Funktion betrachten. Daß dieser Weg in PML nicht eingeschlagen wurde, hat einen rein technischen Grund: Die Evaluierung einer mit WHILE konstruierten Schleife würde weniger ineffizient ablaufen. Bei der Evaluierung einer so gebildeten Schleife müßte nämlich zuerst festgestellt werden, wie oft die Schleifenfunktion iteriert werden muß, bis das Abbruchkriterium erreicht ist, und dann würde die Schleifenfunktion so oft iteriert, wie man dies gerade berechnet hat. Jede Schleife müßte dann doppelt abgearbeitet werden. Bei geschachtelten WHILE-Konstrukten wäre dieser zusätzliche Aufwand gravierend: der zeitliche Aufwand würde mit der Schachtelungstiefe exponentiell anwachsen. Die Funktion WHILE als Grundfunktion in einem PML–Interpreter führt hingegen zu einer effizienteren Abarbeitung der Schleifen, bei der jede Schleife nur einmal durchlaufen wird.

3.4.3 Vergleich: μ–rekursive Funktionen in PML und ML

Die Beschreibung rekursiver Funktionen in PML erscheint auf den ersten Blick ein wenig umständlich. Im Gegensatz zu PML gibt es in ML keine explizite Unterscheidung zwischen

primitiver Rekursion und μ–Rekursion und insbesondere gibt es auch keinen gesonderten Typ zur Darstellung der Funktionswerte μ–rekursiver Funktionen.

Rekursion wird in ML durch Gleichungen und Gleichungssysteme ausgedrückt, bei denen die zu definierende Funktion auch auf der rechten Seite vorkommen darf. Beispiel:

```
fun concat Nil         y = y    |
    concat (Cons h t) y = Cons h (concat t y);
```

Eine derartige Funktionsbeschreibung hat zwar nicht die Form einer Konstantendefinition, man könnte diese Funktion jedoch durch Konstantenspezifikation in HOL einführen. Bei einer Konstantenspezifikation muß nachgewiesen werden, daß es eine Funktion mit der beschriebenen Eigenschaft auch gibt. Im obigen Beispiel muß das folgende Theorem bewiesen werden:

$$\vdash \quad \exists g.\,(\forall y.\,g\,\mathsf{Nil}\,y = y)\ \wedge\ (\forall y.\,g\,(\mathsf{Cons}\,h\,t)\,y = \mathsf{Cons}\,h\,(g\,t\,y))$$

Dieses Theorem ist ableitbar. In diesem Fall wäre diese Vorgehensweise tatsächlich sinnvoll.

Im allgemeinen sind jedoch ML–Funktionen — im Gegensatz zu PML–Funktionen — nur partiell definiert. Für den Fall, daß ein Programm nicht terminiert, ist der Funktionswert durch das Gleichungssystem des ML–Programmtextes nicht festgelegt. Beispiel:

```
fun fa T = F |
    fa F = fa F;
```

Die Funktion fa terminiert für F nicht. Formalisiert man fa in der oben beschriebenen Weise durch Konstantenspezifikation, dann ist fa nicht eindeutig definiert. Das zugehörige Theorem sagt lediglich aus, daß fa entweder die konstant–F–Funktion oder die Negation ist. Das führt zum ersten Problem dieser ML–Methodik: Definiert man nämlich eine zweite Funktion fb in gleicher Weise,

```
fun fb T = F |
    fb F = fb F;
```

dann kann nicht gesagt werden, daß die beiden Funktionen fa und fb identisch sind. Man weiß lediglich, daß beide Funktionen jeweils entweder die konstant–F–Funktion oder die Negation sind.

In PML sind dagegen alle Funktionen total. Bei entsprechenden Implementierungen in PML, wären die Funktionswerte von fa und fb für beide Werte T und F eindeutig festgelegt: Für T wäre er (Defined F) und für F wäre er Undefined. Die beiden Funktionen wären definitiv gleich.

Aber es gibt noch ein zweites Problem mit der ML–Methodik: Es kann sein, daß syntaktisch korrekte ML–Funktionsdefinitionen durch ein formal gesehen widersprüchliches Gleichungssystem beschrieben werden. Beispiel:

```
fun g x = not(g x);
```

Die Funktion g terminiert nie, ist aber eine korrekt gebildete ML–Funktion. Wollte man g, durch Konstantenspezifikation einführen, so müßte vorab

$$\vdash \exists g.\, \forall x.\, g\ x = \mathsf{not}(g\ x);$$

bewiesen werden. Es kann leicht gezeigt werden, daß diese Aussage widersprüchlich ist (Beweisschema: Fallunterscheidung über $(g\,x)$). Damit ist dieses Theorem nicht ableitbar und diese Konstantenspezifikation ist nicht möglich.

In PML wäre die entsprechende Funktion eindeutig definiert. Ihr Funktionswert wäre immer Undefined. Widersprüchliche Beschreibungen gibt es in PML nicht. Das ist dadurch abgesichert, daß alle Konstrukte aus PML durch Konstantendefinitionen, Konstantenspezifikationen und Typdefinitionen formalisiert werden.

Die Evaluierung μ-rekursiver Funktionen

Ein Interpreter, der immer den Funktionswert einer μ–rekursiven Funktion berechnet, also auch dann, wenn er Undefined ist, gibt es nicht (Halteproblem). Man kann einen PML–Interpreter implementieren, der den Funktionswert einer μ–rekursiven Funktion immer dann berechnet, wenn er (Defined y) ist und der nicht terminiert, wenn der Funktionswert Undefined ist.

Das heißt allerdings nicht, daß man, in den Fällen in denen der Funktionswert einer μ–rekursiven Funktion Undefined ist, den Funktionswert prinzipiell nicht bestimmen könnte. Zum einen kann man versuchen, den Funktionswert ,,von Hand" zu ermitteln. Zum anderen kann man versuchen, einen möglichst guten Interpreter zu schreiben, der in dem Fall, daß der Funktionswert (Defined y) ist, den Funktionswert immer bestimmt und der auch in möglichst vielen Fällen erkennt, daß der Funktionswert Undefined ist und dann diesen Wert ausgibt.

3.4.4 Anmerkungen zur systematischen Verwendung der μ–Rekursion

Es ist vorgesehen, daß der Typ *partial* nur zur Beschreibung der Funktionswerte von μ–rekursiven Funktionen verwendet wird. Es sollen zwei verschiedene Gruppen von Funktionen unterschieden werden:

1. Primitive rekursive Funktionen: Funktionen, die ohne die Verwendung von WHILE gebildet wurden. Bei diesen Funktionen soll der Typ *partial* überhaupt nicht verwendet werden.

2. μ–rekursive Funktionen. Alle μ–rekursiven Funktionen sollen den Typ $\alpha \rightarrow (\beta)partial$ haben, wobei der Typoperator *partial* in den Typen α und β nicht vorkommen soll. Außerdem soll gefordert werden, daß der Funktionswert einer Funktion genau dann den Funktionswert Undefined hat, wenn eine Teilfunktion den Funktionswert Undefined hat.

Um diese Forderungen systematisch umzusetzen, kann man wie folgt vorgehen: Die Funktionen und Konstruktoren PRIMREC_partial, CASE_partial, Undefined und Defined werden nicht mehr benutzt. Als Ersatz werden die beiden Funktionen PARTIALIZE und PAPPLY definiert:

$$\vdash \mathsf{PARTIALIZE}\ f\ x\ =\ \mathsf{defined}\ (f\ x)$$

$$\vdash \mathsf{PAPPLY}\ f\ x\ =\ \mathsf{CASE\_partial}\ x\ f\ \mathsf{undefined}$$

Durch PARTIALIZE wird eine primitiv rekursive Funktion vom Typ $\alpha \rightarrow \beta$ in die entsprechende μ–rekursive Funktion vom Typ $\alpha \rightarrow (\beta)partial$ umgewandelt. Mit der Funktion PAPPLY ist es möglich, eine μ–rekursive Funktion auf das Ergebnis einer anderen μ–rekursiven Funktion anzuwenden.

Die einzigen Grundfunktionen für *partial* sind dann: WHILE, PARTIALIZE und PAPPLY. Zusätzlich wird noch die Einschränkung gemacht, daß die Funktion PARTIALIZE nicht auf solche Funktionen angewandt werden soll, die bereits den Typ $\alpha \rightarrow (\beta)partial$ haben. Beachtet man diese Einschränkungen, dann werden die oben gestellten Forderungen eingehalten, und es ist trotzdem noch gewährleistet, daß beliebige μ–rekursive Funktionen dargestellt werden können.

3.5 Abgeleitete Funktionen und Konstanten

3.5.1 Der erweiterte Konstantendefinitionsmechanismus

Funktions- und Konstantendefinitionen in PML entsprechen Konstantendefinitionen in HOL. Tabelle 3.1 zeigt die syntaktischen Unterschiede. Zur Umsetzung der PML–Funktionsdefinitionen muß jedoch eine etwas erweiterte Form der HOL–Konstantendefinition verwendet werden.

PML-Definition	Semantik in HOL
`fun` *name par par ... par* `=` *term* `;` `val` *name* `=` *term* `;`	$\vdash$ *name par par ... par = term* $\vdash$ *name = term*

Tabelle 3.1: Funktions– und Konstantendefinitionen in PML und HOL

In der bisher betrachtete Form der HOL–Konstantendefinition waren lediglich Gleichungen mit der neu einzuführenden Konstante auf der linken Seite und einem geschlossenen Term auf der anderen Seite erlaubt. Es werden jetzt zwei Erweiterungen eingeführt:

1. Auf der linken Seite der Konstantendefinition dürfen Parameter (Variablen) vorkommen. Die Parameter auf der linken Seite der Gleichung dürfen im Term auf der rechten Seite frei vorkommen.

2. Statt einzelner Variablen dürfen als Parameter auf der linken Seite der Gleichung auch mehrere, durch Paarbildung zusammengefaßte Variablen stehen.

Die erweiterte Form der Konstantendefinition kann auf eine Konstantendefinition in der ursprünglichen Form zurückgeführt werden. Das geschieht dadurch, daß man die Gleichung nach der neuen Konstanten auflöst. Die dabei notwendigen Umformungen sollen jetzt formal beschrieben werden. Zunächst werden dazu die beiden HOL–Funktionen CURRY und UNCURRY vorgestellt, die zur Umwandlung von Funktionen mit gepaarten Parametern in Funktionen in Curry–Form bzw. umgekehrt verwendet werden:

$$\vdash \mathsf{CURRY} = \lambda f.\, \lambda x\, y.\, f(x, y)$$

$$\vdash \mathsf{UNCURRY} = \lambda f.\, \lambda z.\, \mathsf{PRIMREC\_prod}\ z\ (\lambda x\, y.\, f\ x\ y)$$

Die notwendigen Umformungen werden durch die drei folgenden Beziehungen beschrieben:

$$\vdash (p\ x = q[x]) \ = \ (p = \lambda x.\, q[x])$$

$$\vdash (p\ (r, s) = q) \ = \ ((\mathsf{UNCURRY}\ p)\ r\ s = q)$$

$$\vdash (\mathsf{UNCURRY}\ p = q) \ = \ (p = \mathsf{CURRY}\ q)$$

Die erste Gleichung beschreibt, wie ein Parameter x, der eine einzelne Variable sein muß, von der linken Seite auf die rechte Seite gebracht werden kann (η–Abstraktion). Mit der zweiten Gleichung wir ein Parameterpaar (r, s) in Curry–Form umgewandelt. Dabei dürfen r und s beliebige Terme sein. Durch Anwendung der zweiten Regel kommt es jedoch zu UNCURRY–Operatoren auf der linken Seite. Mit der dritten Gleichung können diese UNCURRY–Operatoren wieder entfernt werden.

3.5.2 Terme in PML und HOL

Auf der rechten Seite von Funktions– und Konstantendefinitionen in PML und entsprechend bei einer Konstantendefinition in HOL befinden sich Terme. Zwischen Termen in PML und HOL gibt es keine strukturellen Unterschiede. In beiden Sprachen handelt es sich um typisierte λ–Terme, die sich aus Variablen, Konstanten, Funktionsanwendungen und λ–Abstraktionen zusammensetzen. Unterschiede gibt es lediglich bei der Schreibweise für die λ–Abstraktion. Tabelle 3.2 zeigt die syntaktischen Unterschiede zwischen Termen in PML und HOL. Dabei steht x für eine beliebige Variable, c für eine beliebige Konstante und p und q stehen für beliebige Terme. Die Funktionsanwendung ist nur für den Fall erlaubt, daß p den Typ $\alpha \to \beta$ und q den Typ α hat.

Es soll jetzt noch auf ein paar Schreibweisen eingegangen werden, die man in PML und auch in HOL findet. Es soll erläutert werden, für welche λ–Terme sie stehen und wie sie in HOL implementiert wurden. Diese Schreibweisen sind aus formaler Sicht nicht notwendig, sie dienen lediglich einer anschaulicheren Darstellung. Man könnte, ohne die Mächtigkeit der beiden Sprachen zu beschneiden, auf diese Schreibweisen verzichten und sich allein auf „reine" λ–Terme beschränken.

	PML	HOL
Variable	`x`	x
Konstante	`c`	`c`
Funktionsanwendung	`(p q)`	$(p\,q)$
λ–Abstraktion	`(fn x => p)`	$(\lambda x.\,p)$

Tabelle 3.2: Terme in PML und HOL

Schreibweisen für Paare und Listen

Der Typenoperator für Paare heißt sowohl in PML als auch in HOL *prod*. Statt $(\alpha, \beta)prod$ darf in PML $\alpha * \beta$ und in HOL $\alpha\#\beta$ geschrieben werden.[3] Der Typ $(\alpha, \beta)prod$ wird durch den Term–Parser als $\alpha\#\beta$ dargestellt und bei einer Eingabe von $\alpha\#\beta$ wird diese Eingabe durch den Term–Parser in $(\alpha, \beta)prod$ umgewandelt.

Als Konstruktor für den Typs *prod* stehen zwei äquivalente Konstanten zur Verfügung: der Präfixoperator Comma und der Infixoperator `,`. Man beachte, daß `,` nicht etwa eine andere Schreibweise für Comma im Sinne von Pretty–Printing ist.

Die vordefinierte Typen

Die vordefinierten Typen von PML wurden auf die bereits existierenden Typen in HOL aufgesetzt. Dazu mußten in HOL ein paar kleine Anpassungen gemacht werden. Zu diesen Anpassungen gehören die folgenden Konstantendefinitionen.

- $\vdash$ Comma = COMMA
- $\vdash$ Nil = NIL
- $\vdash$ Cons = CONS
- $\vdash$ Zero = ZERO
- $\vdash$ Suc = SUC

Die benutzerfreundlicheren Schreibweisen beruhen auf einem Mechanismus in HOL, der als Pretty–Printing bezeichnet wird. Unter Pretty–Printing versteht man folgendes: Für bestimmte, genau festgelegte Terme werden alternative Darstellungsformen festgelegt. Bei der Eingabe können die Terme in dieser alternativen Form angegeben werden. Der Term–Parser löst diese Schreibweisen automatisch auf und erzeugt reine λ–Terme. Soll ein Term dargestellt werden, für den eine solche Schreibweise definiert wurde, so benutzt der Term–Parser wieder diese Schreibweise. Der Term wird also nicht so ausgegeben, wie er tatsächlich aussieht, sondern in dieser alternativen Darstellungsform.

[3] In Datentypdeklarationen von HOL kommt das Zeichen # ebenfalls vor. Hier wird es als Trennzeichen zwischen den Typen der Argumente der Konstruktoren verwendet. Diese Schreibweise ist verwirrend. Sie erweckt den Eindruck als handle es sich auch hier um diesen Typoperator. Das ist jedoch gerade nicht der Fall. Die Argumente der Konstruktoren werden nicht durch Paarbildung zu einem Tupel zusammengefaßt, sondern sind in Curry–Form.

Für bestimmte Terme des Typs *list* existiert ein Pretty–Printing. Listen der Form

Nil, $(\mathsf{Cons}\, a\, \mathsf{Nil})$, $(\mathsf{Cons}\, a(\mathsf{Cons}\, b\, \mathsf{Nil}))$,...

werden per Pretty–Printing als

$[]$, $[a]$, $[a, b]$,...

dargestellt.[4] Das bedeutet, daß wenn beispielsweise der Term [] eingegeben wird, daß dieser dann vom Term–Parser in Nil umgewandelt wird und in dieser Form gespeichert wird. Bei der Ausgabe wird dieser Term vom Term–Parser wieder in [] umgewandelt.

λ–Abstraktion über Paaren und let–Terme

Die λ–Abstraktion über Paaren gibt es sowohl in PML als auch in HOL. In HOL steht der Term

$\lambda(a, b).\, p$

für

$\mathsf{UNCURRY}(\lambda a\, b.\, p)$

Durch den Term–Parser wird der Term $\mathsf{UNCURRY}(\lambda a\, b.\, p)$ nicht so dargestellt wird, wie er tatsächlich aussieht, sondern als $\lambda(a, b).\, p$ und wenn der Term $\lambda(a, b).\, p$ eingegeben wird, dann wird diese Eingabe automatisch in $\mathsf{UNCURRY}(\lambda a\, b.\, p)$ übersetzt (Pretty–Printing).

Zur Darstellung von let–Termen ist in HOL die Konstante LET definiert:

$\mathsf{LET} = \lambda f\, x.\, f\, x$

Let–Terme sind lediglich andere Formen von β–Redizes. Unter einem β–Redex versteht man einen Term der Form

$(\lambda x.\, p[x])\, y$

Der zu diesem β–Redex äquivalente let–Term heißt in der Syntax von PML

$\mathsf{let\ val}\ x = y\ \mathsf{in}\ p[x]\ \mathsf{end}$

und in der Syntax von HOL

$\mathsf{let}\ x = y\ \mathsf{in}\ p[x]$

Ein let–Term in HOL–Schreibweise steht für den Term

$\mathsf{LET}\ (\lambda x.\, p[x])\, y$

Der Term–Parser stellt diesen Term als

$\mathsf{let}\ x = y\ \mathsf{in}\ p[x]$

dar (Pretty–Printing).

Neben diesen einfachen let–Termen sind (sowohl in PML als auch in HOL) gepaarte let–Terme erlaubt, bei denen anstelle der einzelnen Variable x mehrere gepaarte Variablen stehen. Gepaarte let–Terme entsprechen β–Redizes über gepaarten λ–Abstraktionen.

[4] Genau genommen sind diese Schreibweisen nicht die Konstanten Cons und Nil, sondern für die äquivalenten Konstanten CONS und NIL definiert.

Numerale

Es liegt nahe, zu vermuten, daß die Numerale lediglich andere Darstellungsformen der Terme

Zero, Suc(Zero), Suc(Suc(Zero)),...

sind, und daß für diesen Wechsel der Darstellungsform der Term–Parser verantwortlich ist. Dem ist nicht so.

Die Numerale sind in HOL auf eine andere Weise implementiert worden: Es gibt eine unendlich große Menge von Konstanten vom Typ *num* mit den Namen "0","1","2",.... Die Konstante mit dem Typ *num* und dem Namen "3" wird automatisch als ein Term interpretiert, der äquivalent zu Suc(Suc(Suc Zero)) ist.

Ungewöhnlich ist an diesem Vorgehen, daß dies mit der bisherigen Logik nicht möglich ist. Durch eine endliche Menge von Konstantendefinitionen ist es nicht möglich, die unendliche Menge der Numerale so zu beschreiben. Aus diesem Grund wird, nur für diesen Zweck, die Logik um eine weitere Regel mit dem Namen NUM_CONV erweitert. NUM_CONV berechnet zu einem Numeral n, das ungleich 0 ist, das Theorem

$$\vdash n = \mathsf{Suc}\, m$$

wobei m (ein Numeral) der Vorgänger von n ist. Das Numeral 0 ist durch

$$\vdash 0 = \mathsf{Zero}$$

definiert worden. Mit der Regel NUM_CONV und der Konstantendefinition von Zero lassen sich beliebige Numerale in äquivalente Terme mit Suc und Zero umwandeln.

Der Grund für dieses Vorgehen bei der Einführung der Numerale ist rein technischer Natur. Die Numerale sind bei weitem kompakter als die entsprechenden Terme aus Suc und Zero. Das macht sich vor allem bei großen Zahlenwerten bemerkbar.

3.6 Die Semantik des Beispielprogramms

Jedem Konstrukt aus PML ist eine Semantik in Form eines Theorems zugeordnet worden. Damit ist es möglich, die Semantik eines PML-Programms zu beschreiben. Das nachfolgende Theorem beschreibt die Semantik des Beispielprogramms aus Kapitel 2. Dieses Theorem entsteht durch Konjunktion aller Theoreme, die die Semantik der einzelnen Konstrukte (Variablendeklarationen, Funktions– und Konstantendefinitionen) beschreiben.

Aus Gründen der Übersichtlichkeit sind jene Teile des Theorems ausgelassen worden, die bereits erwähnt wurden. Ausgelassen wurden:

- die Theoreme, die die Semantik der verwendeten Datentypen *bool*, *num*, *prod* und *partial* beschreiben (siehe dazu 3.2.2)
- die Theoreme, die die PRIMREC– und CASE–Funktionen zu diesen Typen beschreiben (siehe dazu 3.3)

- die Konstantendefinitionen zur Einführung von WHILE (siehe dazu 3.4.1)

Die Semantik des Beispielprogramms aus Kapitel 2 lautet:

$$
\begin{array}{l}
\vdash (\mathsf{add}\ a\ b\ =\ \mathsf{PRIMREC\_num}\ a\ b\ (\lambda n.\,\mathsf{Suc})\ )\ \wedge \\
\quad (\mathsf{mult}\ a\ b\ =\ \mathsf{PRIMREC\_num}\ a\ \mathsf{Zero}\ (\lambda n.\,\mathsf{add}\,b)\ )\ \wedge \\
\quad (\mathsf{sub}\ a\ b\ =\ \mathsf{PRIMREC\_num}\ b\ (\lambda x.\,x)\ (\lambda n\ r\ x.\,\mathsf{CASE\_num}\ x\ n\ (\lambda m.\,r\ m))\ a\ )\ \wedge \\
\quad (\mathsf{less\_than}\ a\ b\ = \\
\qquad \mathsf{PRIMREC\_num}\ b\ (\lambda x.\,\mathsf{F})\ (\lambda n\ r\ x.\,\mathsf{CASE\_num}\ x\ \mathsf{T}\ (\lambda m.\,r\ m))\ a\ )\ \wedge \\
\quad (\mathsf{div}\ a\ b\ = \\
\qquad (\mathsf{let}\ c = \\
\qquad\quad \mathsf{WHILE} \\
\qquad\qquad (\lambda((x,y),z).\mathsf{CASE\_bool}\ (\mathsf{less\_than}\ x\ y)\ \mathsf{F}\ \mathsf{T}) \\
\qquad\qquad (\lambda((x,y),z).\mathsf{Defined}\ ((\mathsf{sub}\ x\ y,y),\mathsf{Suc}\ z)) \\
\qquad\qquad ((a,b),\mathsf{Zero}) \\
\qquad \mathsf{in}\ \mathsf{CASE\_partial}\ c\ (\lambda(xy,z).\mathsf{Defined}\ z)\ \mathsf{Undefined}\ )\ )\ \wedge \\
\quad (\mathsf{two} =\ \mathsf{Suc}(\mathsf{Suc}\ \mathsf{Zero})\ )\ \wedge \\
\quad (\mathsf{sqrt}\ a\ = \\
\qquad \mathsf{let}\ c = \\
\qquad\quad \mathsf{WHILE} \\
\qquad\qquad (\lambda\,(x,oldx).\,\mathsf{less\_than}\ \mathsf{two}\ (\mathsf{sub}\ x\ oldx)) \\
\qquad\qquad (\lambda\,(x,oldx). \\
\qquad\qquad\quad \mathsf{let}\ d = \mathsf{div}\ a\ x\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ e = \mathsf{CASE\_partial}\ d\ (\lambda y.\,\mathsf{Defined}(\mathsf{add}\ x\ y))\ \mathsf{Undefined}\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ g = \mathsf{CASE\_partial}\ e\ (\lambda y.\,\mathsf{div}\ y\ \mathsf{two})\ \mathsf{Undefined}\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{CASE\_partial}\ c\ (\lambda\,(x,y).\,\mathsf{Defined}\ x)\ \mathsf{Undefined}\ ) \\
\qquad\qquad (\mathsf{Suc}\ \mathsf{Zero},a) \\
\qquad \mathsf{in}\ \mathsf{CASE\_partial}\ c\ (\lambda\,(x,y).\,\mathsf{Defined}\ x)\ \mathsf{Undefined}\ )\ \wedge \\
\quad (\mathsf{f}\ a\ b\ =\ \mathsf{sqrt}\ (\mathsf{add}\ (\mathsf{mult}\ a\ a)\ (\mathsf{mult}\ b\ b))\ )
\end{array}
$$

Kapitel 4

Funktionale Schaltungsbeschreibung

4.1 Signale

Zur formalen Beschreibung der Signale werden PML–Datentypen verwendet. Den Datentypen kommen dabei drei Teilaufgaben zu:

1. die Beschreibung zeitunabhängiger Einzelsignale
2. die Bündelung der zeitunabhängigen Einzelsignale
3. die Beschreibung der Zeit

Es stellt sich heraus, daß einige Datentypen für mehrere der Teilaufgaben geeignet wären. Um Mißverständnisse zu vermeiden, soll an dem Prinzip festgehalten werden, daß jedem Datentyp immer nur eine der drei oben aufgeführten Teilaufgaben zugeordnet wird.

4.1.1 Einzelsignale

Welche Typen zur Beschreibung der zeitunabhängigen Einzelsignale verwendet werden, hängt davon ab, auf welcher Ebene die Schaltungen beschrieben werden sollen. In den Beispielen dieser Arbeit wird ausschließlich der Typ *bool* mit den Werten T und F verwendet.

Denkbar sind jedoch auch andere Typen. Möchte man beispielsweise Tri–State–Signale beschreiben, so kann man dafür einen eigenen Datentyp *tri* definieren:

```
primitive_datatype "tri = High | Low | Undefined";
```

Auch zusammengesetzte Typen sind möglich. Zur Beschreibung von Tri–State–Signalen hätte man auch den Typ $(bool * bool)$ verwenden können: die erste Komponente des Paares gibt an, ob der Wert definiert ist und die zweite Komponente enthält den Wert des Signals.

Dabei wird jedoch deutlich, wie wichtig es ist, die Typen zur Beschreibung von Einzelsignalen von denen zur Beschreibung von Signalbündeln zu trennen. Es muß eindeutig

erkennbar sein, wie ein Term vom Typ $(bool * bool)$ zu interpretieren ist, ob es sich um ein einzelnes Tri–State–Signal handelt, oder ob es sich um ein Paar boolescher Einzelsignale handelt. Aus diesem Grund sollen Datentypen, wie bereits gefordert wurde, immer nur einer Teilaufgabe zugeordnet werden. Wird der Datentyp *prod* (mit dem Infixtypoperator *) bei der Beschreibung der Signalbündelung verwendet, dann soll er nicht bei der Beschreibung von Einzelsignalen verwendet werden. Diese Forderung bedeutet keine schwerwiegende Einschränkung, da man einfach einen neuen, äquivalenten Datentyp definieren kann.

Andere Typen sind erforderlich, wenn man die Schaltung auf der elektrischen Ebene beschreiben möchte. Möchte man die Spannungspegel durch einen diskreten Wertebereich modellieren, dann eignet sich dazu beispielsweise der Datentyp *num*, der als ein Spannungswert in mV interpretiert wird.

Statt durch natürliche Zahlen, könnte man die Spannungspegel auch durch rationale Zahlen beschreiben. Rationale Zahlen können beispielsweise durch den Typ $bool * num * num$ modelliert werden. Ein Term $(sign, a, b)$ von diesem Typ ist als die rationale Zahl $\frac{a}{Suc\, b}$ für $sign = \mathsf{F}$ bzw. $-\frac{a}{Suc\, b}$ für $sign = \mathsf{T}$ zu interpretieren.

Bisher wurde bei der Auswahl eines geeigneten Typs zur Beschreibung der Einzelsignale lediglich beachtet, ob der Typ für die Problemstellung angemessen ist, ob also mit ihm auf der betrachteten Beschreibungsebene die Signale in adäquater Weise dargestellt werden können. Ein anderer Aspekt ist jedoch der Aufwand, der mit der Schaltungsbeschreibung und der Beweisführung verbunden sein wird. Je präziser und damit auch aufwendiger die Beschreibung der Einzelsignale ist, desto komplexer wird auch i.a. die Schaltungsbeschreibung und die Beweisführung.

4.1.2 Signalbündelung

Es soll möglich sein, mehrere zeitunabhängige Einzelsignale zu Signalstrukturen zusammenzufassen. Eine sehr einfache und auch zweckmäßige Möglichkeit zur Bündelung von Signalen besteht darin, die Einzelsignale durch Paarbildung zusammenzufassen. Das kartesische Mengenprodukt wird in PML durch den Datentyp *prod* repräsentiert. Zwei beliebige Signale x_α und y_β können zu $(x, y)_{\alpha * \beta}$ zusammengefaßt werden. Dabei können x_α und y_β sowohl Einzelsignale als auch bereits gebündelte Signale sein.

Eigentlich wäre der Datentyp *prod* bereits ausreichend, um beliebige Signalbündelungen auszudrücken. Es gibt jedoch Gründe, auch andere Datentypen als *prod* für die Signalbündelung zu verwenden. Kommen in einer Schaltung beispielsweise oft 8 Bit breite Worte vor, so führt dies, wenn man lediglich *prod* zur Signalbündelung verwendet, zu einem recht großen zusammengesetzten Typ.

$$bool * bool * bool * bool * bool * bool * bool * bool$$

Mit der nachfolgenden Datentypdefinition kommt man zu einem atomaren Typ, mit dem ebenfalls die Bündelung von 8 booleschen Einzelsignalen ausgedrückt werden kann.

```
primitive_datatype
   "word = Word of bool#bool#bool#bool#bool#bool#bool#bool";
```

Es gibt noch einen anderen Grund, für die Signalbündelung andere Typen als nur *prod* zu verwenden: Verwendet man für die in einer Schaltung vorkommenden Signalarten verschiedene Typen, dann kann vermieden werden, daß unterschiedliche Signalarten unbeabsichtigt miteinander verbunden werden. Haben beispielsweise der Datenbus und der Adreßbus eines Mikroprozessors beide die gleiche Breite von 16 Bit, dann könnte man, wenn man beide durch den gleichen Typ beschreibt, versehentlich bei einem Systembusteilnehmer die Signale vertauschen. Gibt man dagegen dem Datenbussignal einen anderen Typ als dem Adreßbussignal, dann wird dieser Fehler syntaktisch unmöglich.

Bisher wurden zur Signalbündelung nur nichtrekursive Datentypen mit genau einem Konstruktor verwendet. Das führt dazu, daß aus dem Typ des Signals direkt die Struktur des Signals abgelesen werden kann. Bei anderen Datentypen hängt die Struktur des Signals i.a. auch vom konkreten Wert des Terms ab. Verwendet man beispielsweise den rekursiven Datentyp *list* zur Bündelung boolescher Signale, dann kann ein Term vom Typ (*bool*)*list* sowohl den Wert [T] als auch den Wert [T, F] haben.

Daß in realen Schaltungen ein Signal aus einem Einzelsignal bestehen kann, das den Wert T hat und daß dieses gleiche Signal auch aus zwei Einzelsignalen bestehen kann, die die Werte T und F haben, widerspricht der technischen Realität. Bei realen Schaltungen ist die Anzahl der Einzelsignale fest. Aus diesem Grund werden variable Signalstrukturen nicht zugelassen.

Regel 1 Zur Signalbündelung dürfen nur nichtrekursive Datentypen mit genau einem Konstruktor verwendet werden.

Bemerkung: Diese Einschränkung soll später wieder etwas gelockert werden, wenn abstrakte Schaltungen beschrieben werden.

4.1.3 Zeitabhängige Signale

Als zeitabhängige Signale sollen Funktionen verstanden werden, die einem Zeitpunkt vom Typ *time* einen Signalwert vom Typ *sig* zuordnet. Zeitabhängige Signale haben den folgenden Typ:

$$time \rightarrow sig$$

Dabei ist *sig* der Typ eines zeitunabhängigen Signals und *time* ist ein Typ, mit dem die Zeit beschrieben wird. Welcher Typ zur Beschreibung der Zeit geeignet ist, hängt von der Aufgabenstellung ab.

In dieser Arbeit sollen synchrone Schaltwerke beschrieben werden. Zur Beschreibung der Zeit wird dabei ausschließlich der Typ *num* verwendet. Die Signale der synchronen Schaltwerke haben den Typ $num \rightarrow sig$ und ordnen jedem Taktzeitpunkt einen Signalwert zu.

Sollen Schaltungen auf der elektrischen Ebene beschrieben werden, dann kann gefordert werden, daß die Signale zeitkontinuierlich sein sollen. In diesem Fall könnte man die Zeit beispielsweise durch rationale Zahlen darstellen.

4.2 Schaltnetze

Als Schaltnetze sollen Funktionen verstanden werden, die ein zeitunabhängiges Eingangssignal vom Typ *siga* auf ein zeitunabhängiges Ausgangssignal vom Typ *sigb* abbilden. Schaltnetze sind also Funktionen mit dem folgenden Typ:

$siga \rightarrow sigb$

Schaltnetze können durch PML–Funktionen beschrieben werden. Beispiel: ein and–Gatter mit zwei Eingängen (Abbildung 4.1).

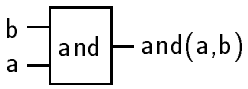

Abbildung 4.1: and–Gatter

Die zughörige PML–Implementierung lautet:

```
fun and (a,b) = CASE_bool a b F;
```

Die Schaltnetzfunktionen können in beliebiger Weise auf bereits definierten Schaltnetzfunktionen aufbauen. Auf diese Weise können Schaltnetzstrukturen ausgedrückt werden. Abbildung 4.2 zeigt die Struktur eines Volladdierers, der sich aus den Teilbausteinen and, or und xor zusammensetzt.

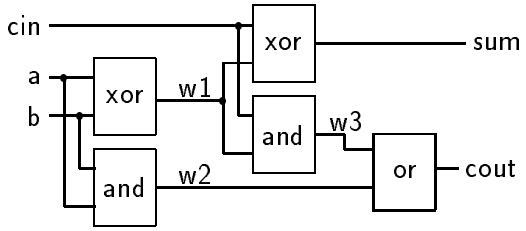

Abbildung 4.2: Struktur von fulladder

Die Implementierung in PML:

```
fun fulladder (cin,(a,b)) =
   let val w1 = xor(a,b) in
   let val w2 = and(a,b) in
   let val sum = xor(w1,cin) in
   let val w3 = and(w1,cin) in
   let val cout = or(w2,w3) in
      (sum,cout)
   end end end end end;
```

Die bei diesem Beispiel gewählte Form der Funktionsdefinition macht den Bezug zur Schaltungsstruktur besonders gut deutlich. Es gibt mehrere äquivalente Funktionsdefinitionen, die alternativ verwendet werden können. Ersetzt man beispielsweise alle let–Terme durch β–Redizes, dann entsteht eine äquivalente Funktionsdefinition, die aber schwerer zu lesen ist:

```
fun fulladder (cin,(a,b)) =
   (fn w1 =>
      (fn w2 =>
         (fn sum =>
            (fn w3 =>
               (fn cout => (sum,cout))
               (w2,w3) )
            (and(w1,cin)) )
         (xor(w1,cin)) )
      (and(a,b)) )
   (xor(a,b));
```

Eine weitere, äquivalente Funktionsdefinition erhält man aus der vorangegangenen Funktionsdefinition durch β–Reduktion:

```
fun fulladder (cin,(a,b)) =
   ( xor(xor(a,b),cin), or(and(a,b),and(xor(a,b),cin)) );
```

In dieser Form wird jedoch nicht mehr deutlich, daß das Ausgangssignal des linken xor–Gatters zweimal verwendet werden soll und nicht etwa statt dessen das Gatter zweimal realisiert werden soll. Der Bezug zur Schaltungsstruktur ist nicht mehr gegeben. Aus diesem Grunde hätte auf den β–Redex mit der Variable `w1` keine β–Reduktion angewandt werden sollen. Besser ist deshalb:

```
fun fulladder (cin,(a,b)) =
   let val w1 = xor(a,b) in
      ( xor(w1,cin), or(and(a,b),and(w1,cin)) )
   end;
```

Prinzipiell gilt, daß die mehrfache Verwendung eines Ausgangssignals durch einen β–Redex bzw. einen äquivalenten let–Term ausgedrückt werden soll. Die Einhaltung dieser Regel führt auch zu Vorteilen bei der Evaluierung: Es wird dadurch vermieden, daß mehrfach der gleiche Teilterm evaluiert werden muß.

4.3 Synchrone Schaltwerke

Synchrone Schaltwerke sind Funktionen, die ein zeitabhängiges Eingangssignal auf ein zeitabhängiges Ausgangssignal abbilden. Sie haben daher den folgenden Typ:

$$(num \rightarrow siga) \rightarrow (num \rightarrow sigb)$$

Die Beschreibung von Schaltwerkstrukturen wirft ein Problem auf: Schaltwerkstrukturen haben Zyklen und Zyklen können nicht funktional beschrieben werden (Erläuterungen dazu folgen in Abschnitt 4.4). Um dieses Problem zu umgehen soll wie folgt vorgegangen werden: Die synchronen Schaltwerke sollen nicht direkt zu Strukturen verbunden werden, sondern jedes Schaltwerk soll durch Aus– und Übergangsschaltnetze repräsentiert werden, und diese Schaltnetze sollen dann miteinander verknüpft werden.

Durch ein Tripel (f, g, q) bestehend aus einem Ausgangsschaltnetz f, einem Übergangsschaltnetz g und einem Anfangszustand q kann ein Schaltwerk eindeutig definiert werden (Abbildung 4.3). Das Ausgangsschaltnetz bildet das aktuelle Eingangssignal vom Typ *siga* und den aktuellen Zustand vom Typ *sigc* auf den aktuellen Ausgangswert vom Typ *sigb* ab. Das Ausgangsschaltnetz bildet das aktuelle Eingangssignal vom Typ *siga* und den aktuellen Zustand vom Typ *sigc* auf den Nachfolgezustand vom Typ *sigc* ab. Der Anfangszustand hat den Typ *sigc*. Die Typen *siga*, *sigb* und *sigc* stehen für zeitunabhängige Signale.

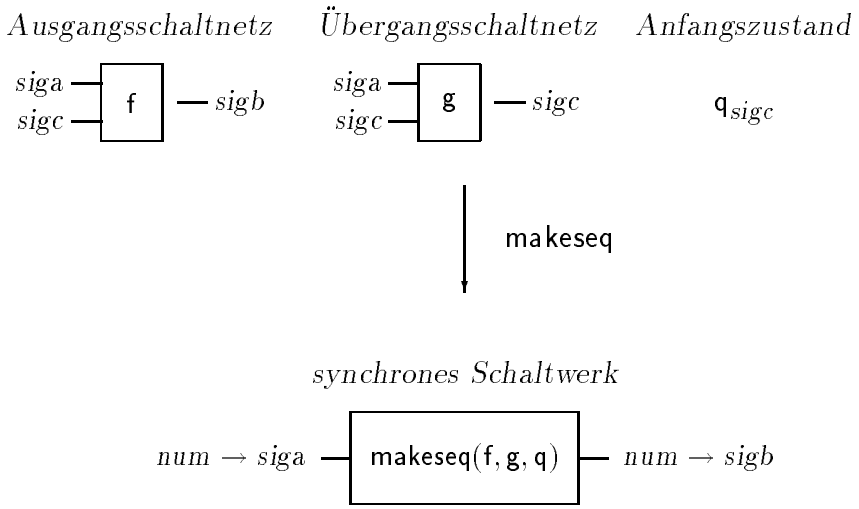

Abbildung 4.3: Die Funktion makeseq

Synchrone Schaltwerke können aus dem Tripel Ausgangszustand, Übergangsschaltnetz und Anfangszustand abgeleitet werden. Für diese Aufgabe wird die Funktion makeseq definiert:

```
fun makeseq (f,g,q) a t =
   f (PRIMREC_num t q (fn n => fn r => g(a n,r)));
```

Die Vorgehensweise bei der Darstellung von Schaltwerken und Schaltwerkstrukturen soll an einem Beispiel illustriert werden: Die Schaltung count stellt einen 1–Bit–Zähler dar, der sich aus einem xor–Gatter und einem D–Flipflop zusammensetzt (Abbildung 4.4).

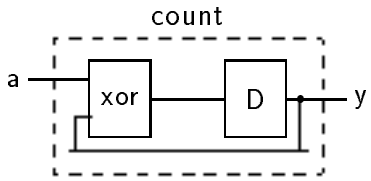

Abbildung 4.4: Schaltwerk count

Die Schaltung count ist ein synchrones Schaltwerk, das ein zeitabhängiges Eingangssignale vom Typ *num* → *bool* auf ein zeitabhängiges Ausgangssignal vom Typ *num* → *bool* abbildet. Um dieses Schaltwerk in Schaltwerkstrukturen verwenden zu können, müssen die Ausgangsfunktion fcount, Übergangsfunktionen gcount und Anfangszustand qcount bestimmt werden. Der Anfangszustand sei F. Aus– und Übergangsfunktion werden aus der Schaltungsbeschreibung extrahiert (Abbildung 4.5).

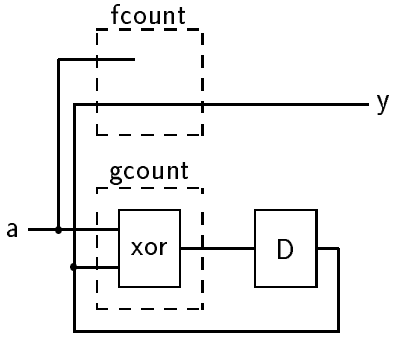

Abbildung 4.5: Ausgangsschaltnetz und Übergangsschaltnetz von count

Die PML–Implementierung gemäß Abbildung 4.5 lautet:

```
fun fcount(a,q) = q;
fun gcount(a,q) = xor(a,q);
val qcount = F;
```

Dieses Tripel stellt nur ein Hilfskonstrukt dar, durch das die eigentlich interessante Schaltwerkfunktion beschrieben wird. Die Schaltwerkfunktion erhält man, indem man auf das Tripel makeseq anwendet. Bemerkung: Es gibt i.a. mehrere verschiedene Tripel, die das gleiche Schaltwerk repräsentieren.

Das Schaltwerk count wird jetzt in einer Schaltungsstruktur verwendet. Die Schaltung mod4 gibt am Ausgang an, ob die Anzahl der T–Werte, die bereits am Eingang anlagen, durch 4 teilbar ist (Abbildung 4.6).

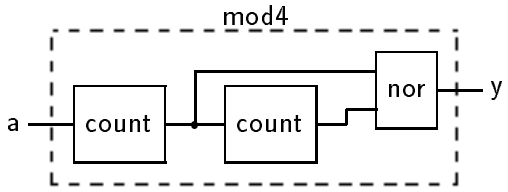

Abbildung 4.6: Struktur von mod4

Diese Struktur wird dadurch beschrieben, daß aus Ausgangsfunktion, Übergangsfunktion und Anfangszustand der Teilschaltwerke (fcount, gcount, qcount) und aus der Schaltnetzfunktion nor Ausgangsfunktion, Übergangsfunktion und Anfangszustand der Gesamtschaltung (fmod4, gmod4, qmod4) abgeleitet werden. In Abbildung 4.7 sind die beiden count–Schaltwerke jeweils durch ihre Ausgangsfunktion und Übergangsfunktion dargestellt. In Abbildung 4.8 sind Ausgangsfunktion und Übergangsfunktion der Gesamtschaltung zusammengefaßt.

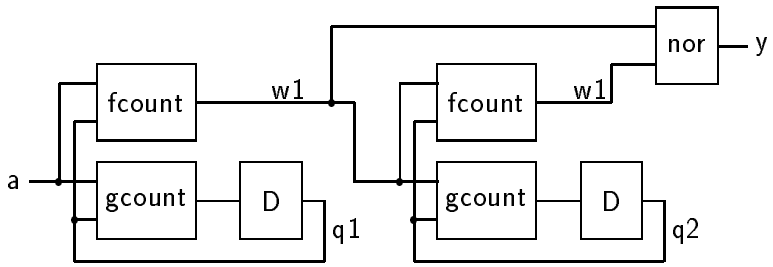

Abbildung 4.7: Struktur von mod4

Die Implementierung in PML gemäß Abbildung 4.8:

```
fun fmod4 (a,(q1,q2)) =
   let val w1 = fcount(a,q1) in
   let val w2 = fcount(w1,q2) in
   let val y = nor(w1,w2) in
      y
   end end end;

fun gmod4 (a,(q1,q2)) =
   let val w1 = fcount(a,q1) in
   let val q1s = gcount(a,q1) in
   let val q2s = gcount(w1,q2) in
      (q1s,q2s)
   end end end;
```

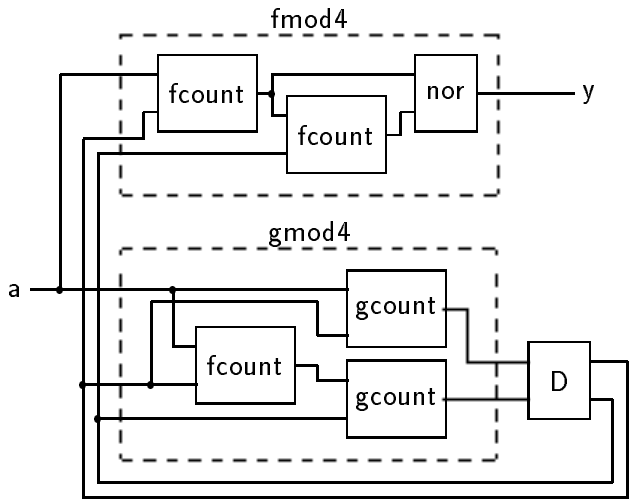

Abbildung 4.8: Struktur von mod4

```
val qmod4 = (qcount,qcount);
```

Auch aus dem Tripel der Gesamtschaltung (fmod4, gmod4, qmod4) kann wieder mit Hilfe von makeseq die Schaltwerkfunktion bestimmt werden.

4.4 Grenzen funktionaler Schaltungsbeschreibung

4.4.1 Funktional beschreibbare Schaltungsstrukturen

Hat man mehrere Schaltungen durch PML–Funktionen beschrieben, die jeweils die Eingangssignale auf die Ausgangssignale abbilden, dann ist es nicht immer möglich, eine aus diesen Schaltungen gebildete Schaltungsstruktur durch eine PML–Funktionsdefinition auszudrücken. Es gilt folgender Zusammenhang:

> Die Struktur einer Schaltung, die sich aus mehreren durch PML–Funktionen beschriebenen Teilschaltungen zusammensetzt, kann genau dann durch eine PML–Funktion beschrieben werden, wenn in der Struktur keine Zyklen vorkommen.

Das soll jetzt gezeigt werden.

Bei einer Funktionsdefinition kann die Gesamtfunktion dadurch ausgewertet werden, daß man die Teilfunktionen in einer bestimmten Reihenfolge sequentiell nacheinander evaluiert. Es gibt immer mindestens eine solche Reihenfolge und i.a. sind mehrere Reihenfolgen möglich. Besitzt eine Strukturbeschreibung einen Zyklus, dann kann eine Teilschaltung, die sich in einem Zyklus befindet, nicht ausgewertet werden, bevor nicht alle Schaltungen in dem Zyklus ausgewertet wurden. Aus diesem Widerspruch folgt, daß es

keine solche Evaluierungsreihenfolge gibt und da es für jede PML–Funktionsdefinition eine Evaluierungsreihenfolge gibt, ist es nicht möglich, daß diese Struktur durch eine PML–Funktionsdefinition ausgedrückt werden kann (Abbildung 4.9).

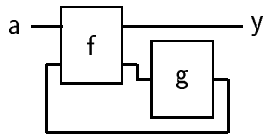

Abbildung 4.9: Verzögerungsfreier Zyklus

Enthält die Schaltungsstruktur dagegen keinen Zyklus, dann gibt es eine Teilschaltung f, deren Eingangssignale nur vom Eingangssignal der Gesamtschaltung und nicht von Ausgangssignalen anderer Teilschaltungen abhängt. Die Schaltungsstruktur für die Gesamtschaltung g kann dann in zwei Teile aufgetrennt werden: in eine Teilschaltung f, die nur von Eigangssignalen abhängt und eine Restschaltung h (Abbildung 4.10).

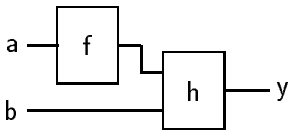

Abbildung 4.10: Zyklenfreie Schaltungsstruktur

Die Implementierung in PML lautet:

```
fun g (a,b) = h(f a,b);
```

Die Restschaltung ist wieder zyklenfrei und kann in analoger Weise weiter zerlegt werden. Auf diese Weise kann die gesamte Strukturbeschreibung von g in eine Funktionsdefinition umgesetzt werden.

4.4.2 Folgerungen

Daß durch die PML–Funktionen nur zyklenfreie Schaltnetzstrukturen beschrieben wurden bedeutet nicht, daß es synchrone Schaltwerke gibt, die funktional nicht beschrieben werden können. Jede Aus– und Übergangsfunktion kann durch eine PML–Funktion beschrieben werden. Deshalb ist es auch möglich, beliebige synchrone Schaltwerke zu beschreiben.

Schaltungen können nicht in beliebiger Weise miteinander verbunden werden. Bei einer Schaltungstruktur, bei der die Teilschaltungen synchrone Schaltwerke sind, kommt es zwangsläufig zu Zyklen. Aus diesem Grund wird der Umweg über die Aus– und Übergangsfunktionen gewählt. Synchronen Schaltwerke werden nicht direkt miteinander zu größeren Schaltungen kombiniert, sondern es werden ihre Aus– und Übergangsfunktionen

zu größeren Aus– und Übergangsfunktionen kombiniert. Parallel dazu werden die Anfangszustände der Teilschaltungen zu einem Anfangszustand zusammengefaßt. Erst die Aus– und Übergangsfunktion der Gesamtschaltung wird zusammen mit ihrem Anfangszustand durch makeseq auf ein synchrones Schaltwerk abgebildet.

Aber auch diese Vorgehensweise garantiert noch nicht, daß keine Zyklen in den Strukturbeschreibungen entstehen können. Geht man von einfachen Grundschaltnetzen wie and, or und inv aus und beschreibt man damit ohne Zwischenschritt ein beliebiges Schaltnetz, dann kann man diese Strukturbeschreibung direkt in eine PML–Funktionsdefinition umsetzen, da die Schaltnetzstruktur zyklenfrei ist. Dagegen kann es zu Zyklen in den Strukturbeschreibungen kommen, wenn zunächst Teilschaltnetze zu Modulen zusammengefaßt werden.

Beispiel: Man möchte die drei Schaltnetze f, g und h in Serie schalten. Das geht ohne Probleme, wenn man die Gesamtschaltung in einem Schritt aus den Teilschaltungen ableitet (Abbildung 4.11).

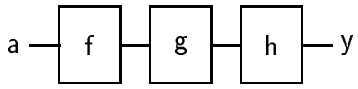

Abbildung 4.11: Serienschaltung 1

Die Implementierung in PML lautet:

```
fun p a = f(g(h a));
```

Man kann diese Struktur auch dann funktional ausdrücken, wenn man zunächst die beiden Schaltungen f und g zu einer Schaltung j zusammenfaßt, um dann j mit h zusammenzufassen (Abbildung 4.12).

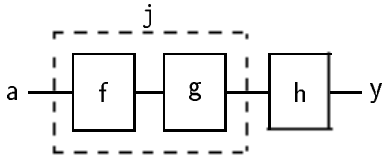

Abbildung 4.12: Serienschaltung 2

Die Implementierung in PML lautet:

```
fun j a = g (f a);
fun p a = h (j a);
```

Eine funktionale Strukturbeschreibung ist aber unmöglich, wenn man zunächst f und h zu einer Schaltung k zusammenfaßt, da dann die Struktur aus k und g einen Zyklus bildet (Abbildung 4.13).

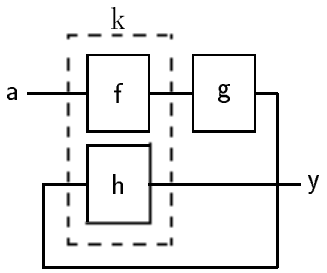

Abbildung 4.13: Serienschaltung 3

Anmerkung

Wenn man Schaltnetze verzögerungsfrei beschreibt, dann führen Zyklen in der Schaltnetzstruktur zu Schaltungen, die aus formaler Sicht widersprüchlich sind. Das Schaltnetz in Abbildung 4.14 führt im Fall a=T zu einem Widerspruch.

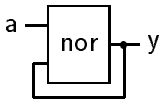

Abbildung 4.14: Widersprüchliches Schaltnetz

Da in dem hier vorgestellten Schema der funktionalen Schaltungsbeschreibung Zyklen prinzipiell nicht beschrieben werden können, ist es auch nicht möglich, derartige widersprüchliche Schaltungen zu beschreiben. Dieser Fehler ist durch die syntaktischen Beschränkungen nicht möglich. Es erweist sich hier als Vorteil, daß nur eindeutige Schaltungsbeschreibungen möglich sind.

Bei einer relationalen Form der Schaltungsbeschreibung können dagegen auch verzögerungsfreie Zyklen und damit widersprüchliche Schaltungen beschrieben werden. Widersprüchliche Schaltungsbeschreibungen haben in der Verifikation schwerwiegende Folgen: Schaltungen mit einer widersprüchlichen Schaltungsbeschreibung besitzen keine technische Realisierung, und man kann für sie beliebige Eigenschaften beweisen (ex falso quodlibet).

4.4.3 Moore–Schaltwerke

Es stellt sich jetzt die Frage nach einer Methodik, mit der man Schaltwerke, die durch Aus– und Übergangsfunktionen beschrieben wurden, derart zusammenfügen kann, daß keine Zyklen in den Schaltnetzstrukturen entstehen. In diesem Ansatz werden lediglich Moore–Schaltwerke als Teilschaltungen zugelassen. Bei Moore–Schaltwerken hängt — im Gegensatz zu Mealy–Schaltwerken — das Ausgangsschaltnetz nicht direkt von den Eingangssignalen ab.

Der Verzicht auf Mealy–Schaltwerke ist schon deshalb keine große Einschränkung, weil Mealy–Schaltwerke nicht in beliebiger Weise kombiniert werden können, ohne daß es zu verzögerungsfreien Zyklen in der Struktur der Grundschaltnetze und damit zu Widersprüchen kommt. Beim Entwurf von Mealy–Schaltwerken müssen diese direkten Abhängigkeiten genau beachtet werden, um verzögerungsfreie Zyklen zu vermeiden. Dagegen ist ein beliebiger Graph aus Moore–Schaltwerken immer frei von verzögerungsfreien Zyklen. Die Synthese mit Moore–Teilschaltungen ist deshalb einfacher zu handhaben als mit Mealy–Teilschaltungen.

Es soll jetzt gezeigt werden, daß eine Schaltungsstruktur bestehend aus zwei Moore–Schaltwerken immer funktional beschrieben werden kann. Genauer ausgedrückt: Es soll gezeigt werden, daß wenn die beiden Moore–Teilschaltwerke durch Ausgangsfunktion, Übergangsfunktion und Anfangszustand gegeben sind, daß dann Ausgangsfunktion, Übergangsfunktion und Anfangszustand des Moore–Gesamtschaltwerkes bestimmt werden
können. Dies soll für beliebige Schaltungsstrukturen aus zwei Moore–Teilschaltungen gezeigt werden.

Abbildung 4.15 zeigt die allgemeine Form einer Schaltungsstruktur aus zwei beliebigen Moore–Schaltwerken A und B. Zu A und B seien jeweils die Ausgangsfunktionen fa bzw. fb, die Übergangsfunktionen ga bzw. gb und die Anfangszustände qa bzw. qb gegeben. Es wird durch PML–Funktionsdefinitionen angegeben, wie zur Gesamtschaltung C die Ausgangsfunktion fc, die Übergangsfunktion gc und der Anfangszustand qc auf Ausgangsfunktion, Übergangsfunktionen und Anfangszustand der Teilschaltungen zurückgeführt werden können.

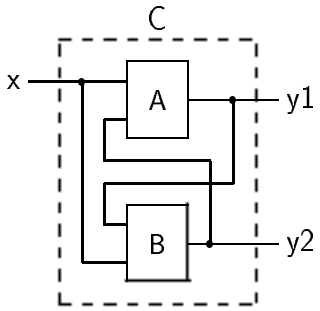

Abbildung 4.15: Struktur aus zwei Moore–Schaltwerken

Die PML–Implementierung gemäß Abbildung 4.15:

```
fun fc (q1,q2) = (fa q1,fb q2);

fun gc (x,(q1,q2)) =
   let val (y1,y2) = fc(qa,qb) in
      ( ga((x,y2),q1), gb((x,y1),q2) )
```

```
    end;

val qc = (qa,qb);
```

Da zwei beliebige Moore–Schaltwerke zusammengefaßt werden können, ist es auch möglich beliebige Schaltwerkstrukturen aus beliebig vielen Moore–Schaltwerken zu beschreiben. Dazu muß diese Schaltwerkstruktur nur sukzessive geteilt werden.

4.5 Vergleich mit relationalen Schaltungsbeschreibungen

Das verwendete Formalisierungsschema beschreibt Schaltungen durch eine Funktion, die den Eingangssignalen ein Ausgangssignal zuordnet. Im Gegensatz dazu werden Schaltungen oft durch Relationen dargestellt. Die Relationen beschreiben den Zusammenhang zwischen den Signalen in einer allgemeineren Weise. Zwischen Ein– und Ausgangssignalen muß dabei nicht unterschieden werden, und es muß auch nicht unbedingt ein funktionaler Zusammenhang zwischen den Signalen bestehen. Die Signale können sich in beliebiger Weise gegenseitig beeinflussen.

In HOL werden die Relationen durch Prädikate, also durch Funktionen mit der Bildmenge *bool* dargestellt. Eine Schaltung mit dem Eingangssignaltyp α und dem Ausgangssignaltyp β hat bei einer funktionalen Beschreibung den Typ $\alpha \to \beta$ und bei einer relationalen Beschreibung den Typ $\alpha * \beta \to bool$. Zu einer beliebigen funktionalen Schaltungsbeschreibung $f_{\alpha\to\beta}$ kann durch $g(x,y) = (y = f\,x)$ eine relationale Schaltungsbeschreibung $g_{\alpha*\beta\to bool}$ definiert werden. Umgekehrt läßt sich jedoch nicht jede Relation in eine Funktion umwandeln.

Bei der funktionalen Form der Schaltungsbeschreibung können Schaltungsstrukturen mit Zyklen nicht beschrieben werden, und es mußte deshalb zur Beschreibung von Schaltwerken ein Umweg über die Aus– und Übergangsschaltnetze genommen werden. Beim relationalen Ansatz ist die Beschreibung von Schaltungsstrukturen einfacher. Eine beliebige Schaltungsstruktur, bei der die Teilschaltungen durch Relationen beschrieben werden, kann direkt in eine Formel umgesetzt werden, die die Gesamtschaltung relational beschreibt. Das Schema soll am Beispiel der Struktur von Abbildung 4.16 erläutert werden. Die relationale Strukturbeschreibung sieht folgendermaßen aus:

$$C(a,b,x,y) \;=\; \exists\, w1\; w2\; w3.\; A(b,w3,w1) \wedge B(a,w1,x) \wedge C(w3,w2,y)$$

Die größeren Freiheitsgrade bei der relationalen Schaltungsbeschreibung haben jedoch auch Nachteile. Meist soll zwischen den Ein– und Ausgangssignalen ein eindeutiger Zusammenhang hergestellt werden. Bei Relationen ist dies jedoch nur ein möglicher Fall. Relationale Schaltungsbeschreibungen können insbesondere auch mehrdeutig und auch widersprüchlich sein.

Ein zweites Problem der relationalen Form der Schaltungsbeschreibung besteht darin, daß bei der relationalen Strukturbeschreibung auch „unerwünschte" Strukturen ausge-

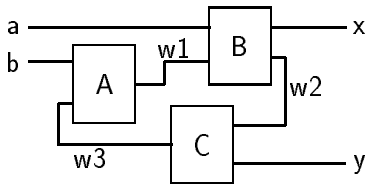

Abbildung 4.16: Schaltungsstruktur mit Zyklen

drückt werden können. Insbesondere können, im Gegensatz zur funktionalen Strukturbeschreibung, auch Kurzschlüsse und verzögerungsfreie Zyklen beschrieben werden. Selbst dann, wenn die Relationen der Teilschaltungen nicht widersprüchlich sind, können durch Schaltungsstrukturen, die Kurzschlüsse oder verzögerungsfreien Zyklen enthalten, widersprüchliche Gesamtschaltungen ausgedrückt werden. Übersieht man derartige Fehler beim Entwurf, so entstehen gravierende Konsequenzen für die Verifikation: Für widersprüchliche Schaltungsbeschreibungen lassen sich beliebige Eigenschaften beweisen. Derartige Fehler in der Struktur lassen sich bei einer funktionalen Strukturbeschreibung schon syntaktisch nicht ausdrücken.

Ein weiterer Vorteil der funktionalen Schaltungsbeschreibung ist die Ausführbarkeit. Es ist nicht nur bekannt, daß zwischen Ein– und Ausgangssignalen ein Zusammenhang besteht, sondern es ist auch sichergestellt, daß es zu einer beliebigen Eingangssignalbelegung ein eindeutig bestimmbares Ausgangssignal gibt. Dieses Ausgangssignal kann durch Evaluierung bestimmt werden — zumindest stimmt das dann, wenn der Funktionswert nicht Undefined ist. Beschränkt man sich auf primitiv rekursive Funktionen, wie dies bisher geschehen ist, dann terminiert die Evaluierung immer. Außerdem gibt es für λ–Terme Optimierungsalgorithmen, die den λ–Term vor der Evaluierung in einen äquivalenten, aber effizienter zu evaluierenden λ–Term umwandeln ([Jone87]).

Der Vorteil der Evaluierbarkeit der funktional beschriebenen Schaltungen kann auch für die Verifikation von Nutzen sein. Möchte man beispielsweise die Äquivalenz zweier Schaltnetze zeigen, so kann man dies durch Fallunterscheidung über den Eingangssignalen und anschließender Evaluierung geschehen. Vorausgesetzt werden muß natürlich, daß die Menge der Eingangssignale endlich ist, und sinnvoll ist diese Methode auch nur dann, wenn die Menge der Eingangssignale klein ist.

Zusammenfassend eine Übersicht über die wichtigsten Vor– und Nachteile von funktionaler bzw. relationaler Schaltungsbeschreibung:

- relationale Schaltungsbeschreibung
  - \+ beliebige, auch mehrdeutige Schaltungen sind beschreibbar
  - \+ beliebige Schaltungsstrukturen darstellbar
  - – widersprüchliche Schaltungen können beschrieben werden
  - – Kurzschlüsse und verzögerungsfreie Zyklen darstellbar

- funktionale Schaltungsbeschreibung
  - \+ Evaluierbarkeit
  - \+ Kurzschlüsse und verzögerungsfreie Zyklen sind durch syntaktische Beschränkungen unmöglich
  - ± alle Schaltungsbeschreibungen sind eindeutig
  - – Einschränkungen bei der Strukturbeschreibung

Kapitel 5

Abstrakte Schaltungen

Bisher wurden Funktionen beschrieben, die jeweils eine einzelne Schaltung repräsentieren. Im Gegensatz dazu werden in diesem Kapitel Funktionen dazu verwendet, Schaltungsmengen zu beschreiben. Diese Funktionen werden als *abstrakte Schaltungen* bezeichnet, wohingegen die bisher betrachteten Funktionen, die jeweils einzelne reale Schaltungen repräsentieren, als *konkrete Schaltungen* bezeichnet werden. Eine konkrete Schaltung, die ein Element der durch eine abstrakten Schaltung beschriebenen Menge von Schaltungen ist, wird als eine *Ausprägung* der abstrakten Schaltung bezeichnet.

In diesem Kapitel wird zunächst erläutert, welche Form abstrakte Schaltungen haben und wie aus ihnen konkrete Schaltungen abgeleitet werden können. Der zweite Teil widmet sich der Beschreibung regulärer Schaltungsstrukturen. Es soll gezeigt werden, wie mit Hilfe abstrakter Schaltungen reguläre Schaltungsstrukturen ausgedrückt werden können.

5.1 Die Form abstrakter Schaltungen

Aufbauend auf dem bisherigen Konzept der funktionalen Schaltungsbeschreibung werden im folgenden Erweiterungen eingeführt, die zu abstrakten Schaltungen führen. Abstrakte Schaltungen unterscheiden sich in drei Punkten von konkreter Schaltungen:

1. Sie dürfen polymorph sein, d.h. sie dürfen Typenvariablen enthalten.
2. Sie können parameterbehaftet sein.
3. Es dürfen, entgegen der im vorigen Kapitel definierten Regel 1, auch rekursive Datentypen zur Signalbündelung verwendet werden.

Für alle abstrakten Schaltungen wird eine einheitliche Form festgelegt:

$$par1 \rightarrow par2 \rightarrow \ldots parn \rightarrow siga \rightarrow sigb$$

par1, *par2*,...*parn* sind die Typen der Parameter der abstrakten Schaltung, *siga* ist der Typ des Eingangssignals und *sigb* ist der Typ des Ausgangssignals. Die Parameter abstrakter Schaltungen dürfen beliebig sein. Insbesondere dürfen die Parameter auch selbst Schaltungen (konkrete und abstrakte) sein.

Abstrakte Schaltungen haben eine allgemeinere Form als konkrete Schaltungen. Durch Spezialisierung können aus abstrakten Schaltungen konkrete Schaltungen erzeugt werden. Es gibt für alle abstrakten Schaltungen zwei Verfahren, um aus abstrakten Schaltungen konkrete Schaltungen zu gewinnen. Diese sind:

- Variablensubstitution
- Typeninstanziierung

Jede abstrakte Schaltung repräsentiert eine Menge von konkreten Schaltungen, die aus ihr mittels Variablensubstitution und Typeninstanziierung abgeleitet werden können.

5.1.1 Polymorphe Funktionen

Abstrakte Schaltungen dürfen, im Gegensatz zu konkreten Schaltungen, polymorph sein, d.h. sie dürfen Typenvariablen haben. An Stelle der Typenvariablen dürfen bei den Ausprägungen der abstrakten Schaltung beliebige Typen von Signalen stehen.

Über variablen Typen können keine „echte" Operationen wie die Addition oder die Konjunktion definiert werden. Diese Operationen entstehen durch primitive Rekursion und setzen immer Operanden mit festen Datentypen voraus.

Abbildung 5.1 zeigt einen 2:1–Multiplexer mit der Bezeichnung mux, bei dem mit einem booleschen Steuersignal aus zwei Eingangssignalen eines ausgewählt wird.

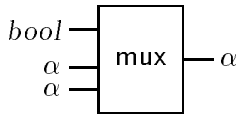

Abbildung 5.1: 2:1–Multiplexer

Die Implementierung in PML:

```
fun mux((s:bool),(a:'a),(b:'a)) = CASE_bool s b a;
```

Die beiden Eingangssignale und auch das Ausgangssignal haben den gleichen Typ α. Die Typvariable α steht für einen beliebigen Typ, der mittels Typeninstanziierung durch einen beliebigen Signaltyp ersetzt werden kann. Abbildung 5.2 zeigt, wie aus der abstrakten Schaltung mux per Typeninstanziierung konkrete Schaltungen abgeleitet werden.

Die Typeninstanziierung erfolgt automatisch bei der Verwendung der Funktion in PML–Termen. Wird beispielsweise der Term $\mathsf{mux}(\mathsf{F},(\mathsf{T},\mathsf{T}),(\mathsf{F},\mathsf{F}))$ angegeben, dann wird automatisch die Typenvariable α durch $bool * bool$ ersetzt. Außerdem kann der Typ α auch explizit durch Constraints spezialisiert werden.

Auch abstrakte Schaltungen können (wie konkrete Schaltungen) zu Strukturen zusammengefügt werden. Es entstehen neue, abstrakte Schaltungen. In Abbildung 5.3 baut der polymorphe 4:1-Multiplexer mux4 auf dem polymorphen 2:1-Multiplexer mux auf.
Die Implementierung in PML:

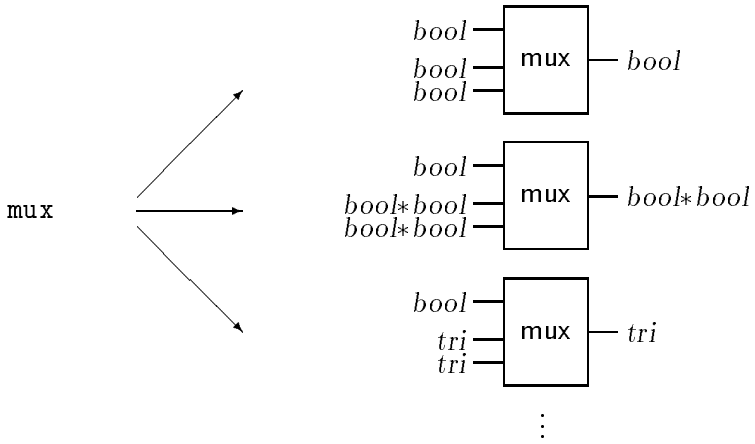

Abbildung 5.2: Ausprägungen von mux

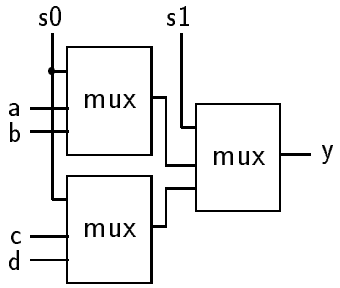

Abbildung 5.3: Schaltungsstruktur mit abstrakten Teilschaltungen

```
fun mux4 (s0,s1,a,b,c,d) = mux(mux(s0,a,b),mux(s1,c,d));
```

5.1.2 Parameterbehaftete Funktionen

Der Typ eines Parameters einer Schaltung kann beliebig sein. Wichtig ist, daß zwischen Parametern und Signalen klar unterschieden wird. Es empfiehlt sich, für Parameter und Signale verschiedene Typen zu verwenden.

Es soll das Übergangsschaltnetz fcounter eines 1–Bit–Zählers mit Reset-Eingang beschrieben werden. Von fcounter soll es zwei Ausprägungen geben: eine mit active–low und eine mit active–high Reset–Eingang. Um welche der beiden Ausprägungen es sich handelt, wird durch einen Parameter angegeben. Dieser Parameter muß einen Typ haben, der genau zwei verschiedene Werte annehmen kann. Es wäre möglich, für diesen Parameter den Typ *bool* zu benutzen. Das könnte dann allerdings zu Verwechslungen zwischen Parametern und Signalen führen. Aus diesem Grund wird für den Parameter ein eigener

Typ definiert:

```
primitive_datatype  " active = Activelow | Activehigh ";
```

Die abstrakte Schaltung wird wie folgt implementiert:

```
fun fcounter act ((a,res),q) =
   and(xor(a,q),CASE_active act res (inv res));
```

Abbildung 5.4 zeigt die beiden Ausprägungen von fcounter.

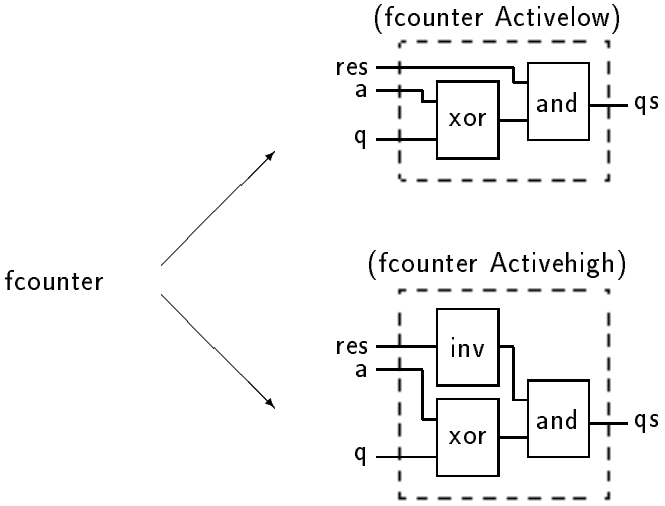

Abbildung 5.4: Ausprägungen von fcounter

Es gilt folgender Zusammenhang:

$$\begin{aligned} \text{fcounter Activelow } ((a, res), q) &= \text{and}(\text{xor}(a, q), res) \\ \text{fcounter Activehigh } ((a, res), q) &= \text{and}(\text{xor}(a, q), \text{inv } res) \end{aligned}$$

Als Parameter für abstrakte Schaltungen sind auch Schaltungsfunktionen erlaubt. Bei Schaltungsfunktionen als Parametern kann eine Verwechslung mit Signalen nicht passieren, da Schaltungsfunktionen immer Funktionen von Signalen auf Signale sind. Zeitabhängige Signale können nicht den gleichen Typ haben, da diese keine Funktionen sind und auch mit zeitabhängigen Signalen können sie nicht verwechselt werden, da in zeitabhängigen Signalen der Typ *num* zur Beschreibung der Zeit vorkommt und *num* für keine andere Funktion verwendet werden soll.

Die abstrakte Schaltung double f entsteht durch Serienschaltung zweier Schaltungen $f_{\alpha\to\alpha}$. Die Schaltung f ist ein Parameter von double.

```
fun double f x = f(f x);
```

Da als Parameter eine beliebige Schaltung vom Typ $\alpha \to \alpha$ in Frage kommt, gibt es unendliche viele Ausprägungen von double. Beispiele für Ausprägungen von double zeigt Abbildung 5.5.

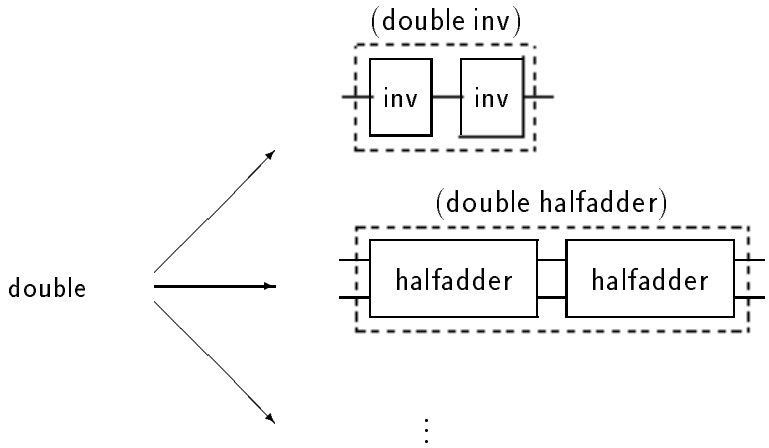

Abbildung 5.5: Ausprägungen von double

5.1.3 Rekursive signalbündelnde Datentypen

In der Regel führen Schaltungsstrukturen zu regulären Signalstrukturen. Soll beispielsweise eine n–fach–Parallelschaltung beschrieben werden, dann entstehen dadurch n–fach gebündelte Ein- und Ausgangssignale (Abbildung 5.6).

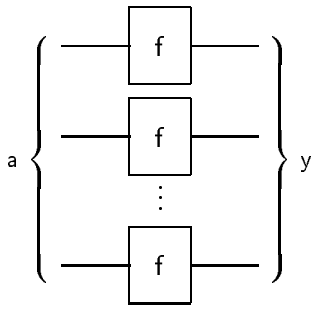

Abbildung 5.6: n–fache Parallelschaltung

Für konkrete Schaltungen wurde gefordert, daß die Datentypen zur Signalbündelung nicht rekursiv sein dürfen. Zur Bündelung der Datentypen werden bei konkreten Schaltungen lediglich Paare und Tupel verwendet. Jede konkrete Schaltung bildet ein Eingangssignal mit einer fest vorgegebenen Signalstruktur auf ein Ausgangssignal mit einer fest vorgegebenen Signalstruktur ab. Die Struktur der Ein– und Ausgangssignale kann aus dem Typ der Funktion abgelesen werden.

Möchte man die in Abbildung 5.6 skizzierte Schaltung durch eine Funktion ausdrücken, dann müssen die Strukturen der Ein– und Ausgangssignale flexibel sein. Ein und dieselbe Funktion soll für verschiedene n ein n–Bit breites Eingangssignal auf ein n–Bit breites Ausgangssignal abbilden. Beschränkt man sich bei der Signalbündelung auf Paare und

Tupel, dann kann man diese Funktion nur jeweils für ein einzelnes, fest vorgegebenes n realisieren.

Zur Bewältigung dieser Problemstellung wird die Beschränkung auf nichtrekursive Datentypen bei der Signalbündelung wieder aufgehoben. Bei abstrakten Schaltungen dürfen zur Signalbündelung auch rekursive Typen (Listen, Bäume, etc.) verwendet werden. Die Ein– und Ausgangssignale aus Abbildung 5.6 können beispielsweise durch Listen gebündelt werden. Die Gesamtschaltung ist dann eine Funktion mit dem Typ $(\alpha)list \rightarrow (\beta)list$. Die Struktur der Ein– und Ausgangssignale und damit die Anzahl der Teilsignale ist durch den Typ der Funktion noch nicht festgelegt. Als Ein– und Ausgangssignale sind Listen beliebiger Länge möglich.

Im vorangegangenen Kapitel wurde bereits angedeutet, daß die Verwendung rekursiver, signalbündelnder Typen dazu führen kann, daß Funktionen entstehen, die aus technischer Hinsicht keine sinnvolle Schaltungsbeschreibungen darstellen. Dazu nun ein Beispiel:

```
fun bad x =  mux(x,[T],[T,F]);
```

Es gilt $\mathsf{bad}\,\mathsf{F} = [\mathsf{T}]$ und $\mathsf{bad}\,\mathsf{T} = [\mathsf{T},\mathsf{F}]$. Je nachdem, welchen Wert das Eingangssignal annimmt, ändert sich die Struktur des Ausgangssignals. Diese Funktion ist zur Beschreibung einer real existierenden Schaltung nicht geeignet. Aus diesem Grund wird die Verwendung rekursiver, signalbündelnder Typen, eingeschränkt.

Regel 2 Bei abstrakten Schaltungen darf die Struktur des Ausgangssignals nur von den Parametern und von der Struktur des Eingangssignals abhängen, nicht aber von den Werten der Einzelsignale des Eingangssignals.

Die Schaltung aus Abbildung 5.6 erfüllt diese Forderung. Die Strukturen der Ein– und Ausgangssignale sind immer gleich, es sind immer zwei gleich lange Listen. Bereits aus der Länge des Eingangssignals ergibt sich die Länge des Ausgangssignals. Die Werte der Einzelsignale des Eingangssignals haben auf die Struktur des Ausgangssignals keinen Einfluß.

Auf ein zweites Problem führt die folgende Funktion:

```
fun possiblybad (x:bool,y) = CONS x y;
```

Das Ausgangssignal hat immer genau ein Listenelement mehr als die zweite Komponente des Eingangssignals. Diese Funktion erfüllt Regel 2 und sie eignet sich tatsächlich zur Beschreibung eines Schaltnetzes.

Verwendet man diese Funktion aber als Übergangsschaltnetz, dann repräsentiert sie ein synchrones Schaltwerk, bei dem der innere Zustand des Schaltwerks von Takt zu Takt um ein Listenelement zunimmt. Ein derartiges Schaltwerk ist technisch nicht realisierbar. Das führt zu einer zweiten Einschränkung:

Regel 3 Bei Übergangsfunktionen müssen die Struktur von altem und neuem Zustand identisch sein.

Wird der Zustand durch eine Liste boolescher Werte dargestellt, dann heißt das, daß die Übergangsfunktion die Länge dieser Liste erhalten muß. Wenn das Übergangsschaltnetz die Länge der Liste nicht ändert, dann kann ein Schaltwerk, das auf dieser Übergangsfunktion aufbaut, nur jene Zustände annehmen, bei denen die Liste genauso lang ist wie im Anfangszustand.

5.2 Reguläre Schaltungsstrukturen

Abstrakte Schaltungen sollen insbesondere dazu verwendet werden, Regularität in Schaltungen auf eine systematische Weise auszudrücken. Es wird folgende Vorgehensweise gewählt: Ausgangspunkt für die Beschreibung einer rekursiven Schaltung ist ein Rekursionsschema — eine rekursive Definition der Schaltungsstruktur. Dieses Rekursionsschema wird in eine abstrakter Schaltung umgesetzt, und aus dieser können dann verschiedene Ausprägungen abgeleitet werden.

Durch konkrete Schaltungen lassen sich bereits beliebige Schaltungsstrukturen, also insbesondere auch reguläre Schaltungsstrukturen, ausdrücken. Bei einer direkten Umsetzung einer regulären Schaltungsstruktur in eine Netzliste und dann in eine konkrete Schaltung geht die Information über die Regularität verloren und kann nicht mehr genutzt werden. Dagegen wird bei der hier gewählten Vorgehensweise die Regularität explizit ausgedrückt. Reguläre abstrakten Schaltungen entstehen durch primitive Rekursion. Bei der Verifikation können, statt zahlreicher gleichartiger Fallunterscheidungen auf der Ebene konkreter Schaltungen zu führen, schnellere Induktionsbeweise auf der Ebene abstrakten Schaltungen angewandt werden. Ein zweiter Vorteil besteht darin, daß aus abstrakten Schaltungen jeweils mehrere konkrete Schaltungen abgeleitet werden. Statt separat einen 1–Bit Volladdierer, dann einen 2–Bit Volladdierer, dann einen 3–Bit Volladdierer etc. zu implementieren, wird zunächst ein n–Bit Volladdierer beschrieben, aus dem dann all diese Schaltungen abgeleitet werden können.

Dazu jetzt ein einführendes Beispiel. Es soll eine abstrakte Schaltung entworfen werden, die zu einer natürlichen Zahl n_{num} und einer Schaltungsfunktion $f_{\alpha\to\alpha}$ die n–fache Serienschaltung von f beschreibt (Abbildung 5.7). Die Gesamtschaltung ist die abstrakte Schaltung ser mit den Parametern n und f.

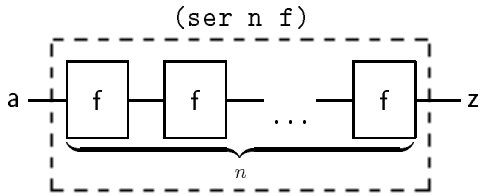

Abbildung 5.7: n–fache Serienschaltung

Abbildung 5.8 zeigt das Rekursionsschema, durch das die Schaltungsstruktur von ser

beschrieben wird. Es werden zwei Fälle unterschieden: $n = 0$ und $n = \mathsf{Suc}\, m$. Für beide Fälle wird durch eine Schaltungsstruktur $(\mathsf{ser}\, n\, f)$ beschrieben. Das besondere an diesen Schaltungsstrukturen ist, daß sie rekursiv sind. Die zu definierende Schaltung kommt in ihrer eigenen Strukturbeschreibung wieder vor.

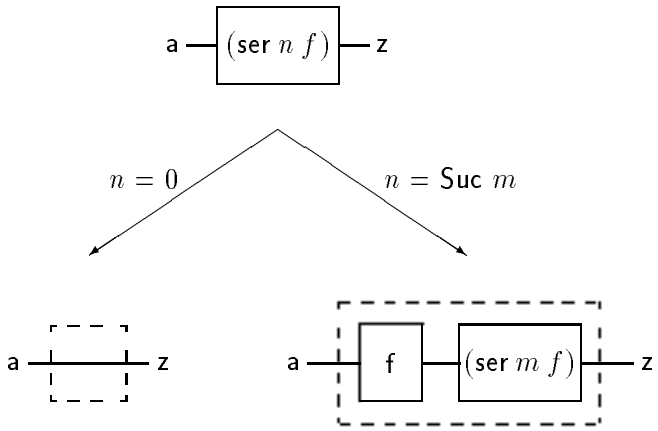

Abbildung 5.8: Rekursionsschema von ser

Die abstrakte Schaltung ser kann durch die folgenden beiden Gleichungen definiert werden. Die beiden Gleichungen sind Umsetzungen der beiden Schaltungsstrukturen für die Fälle $n = 0$ und $n = \mathsf{Suc}\, m$ aus Abbildung 5.8.

$$\mathsf{ser}\, 0\, f\, a = a$$
$$\mathsf{ser}\, (\mathsf{SUC}\, n)\, f\, a = f\, (\mathsf{ser}\, n\, f\, a)$$

Dieses Gleichungssystem beschreibt eine primitive Rekursion über den natürlichen Zahlen. Bei der Implementierung in PML wird die primitive Rekursion durch die Grundfunktion PRIMREC_num ausgedrückt:

```
fun ser n f a =
   PRIMREC_num a (fn m => fn r => f r);
```

Im allgemeinen führen reguläre Schaltungsstrukturen zu regulären Strukturen in den Signalstrukturen. Ein Beispiel für eine Schaltung mit regulären Signalstrukturen ist die n–fache Parallelschaltung gemäß Abbildung 5.9. Die abstrakte Schaltung par hat einen Parameter: die Schaltung f. Die Ein– und Ausgangssignale werden durch Listen gebündelt. Bei ser wurde die Anzahl der in Serie zu schaltenden Schaltungen f durch einen Parameter festgelegt. Im Gegensatz dazu wird die Anzahl der parallel zu schaltenden Teilschaltungen f nicht explizit angegeben — die Anzahl ergibt sich aus der Länge der Liste des Eingangssignals.

In Abbildung 5.10 ist das Rekursionsschema für par dargestellt. Es findet eine Fallunterscheidung über der Struktur des Eingangssignals statt: Die Liste a kann leer sein

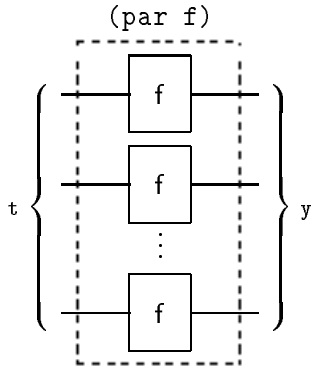

Abbildung 5.9: n–fache Parallelschaltung

(a = Nil) oder sich aus einem ersten Element h und einer Restliste t zusammensetzen (a = Cons h t). Das Rekursionsschema beschreibt die Struktur von (par f) für diese beiden Fälle.

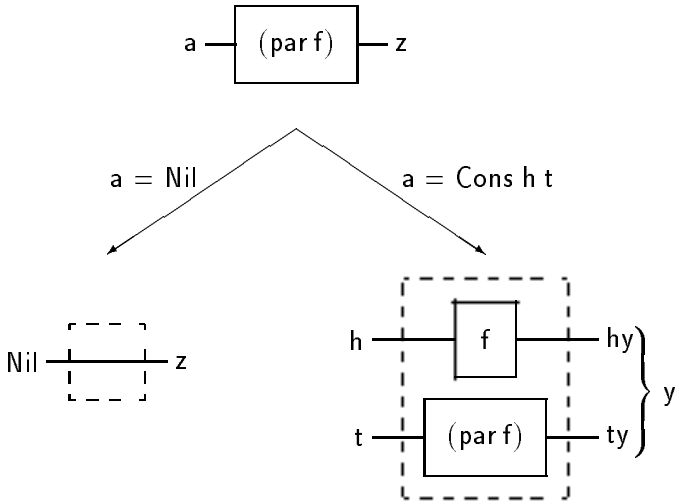

Abbildung 5.10: Rekursionsschema von par

Für par ergibt sich folgende Spezifikation:

par f Nil = Nil
ser f (Cons h t) = Cons (f h)(ser f t)

Das Gleichungssystem beschreibt eine primitive Rekursion über dem Datentyp *list*. Die Implementierung in PML lautet:

```
fun par f a =
```

```
PRIMREC_list a (fn h => fn t => fn r => CONS (f h) r);
```

Es lassen sich auch Schaltungsstrukturen beschreiben, bei denen mehrere rekursive, signalbündelnde Signaltypen vorkommen. Bei der Beschreibung eines 2^n : 1-Multiplexers werden die Steuersignale durch eine Liste und die Eingangssignale durch einen Binärbaum strukturiert (Abbildung 5.11).

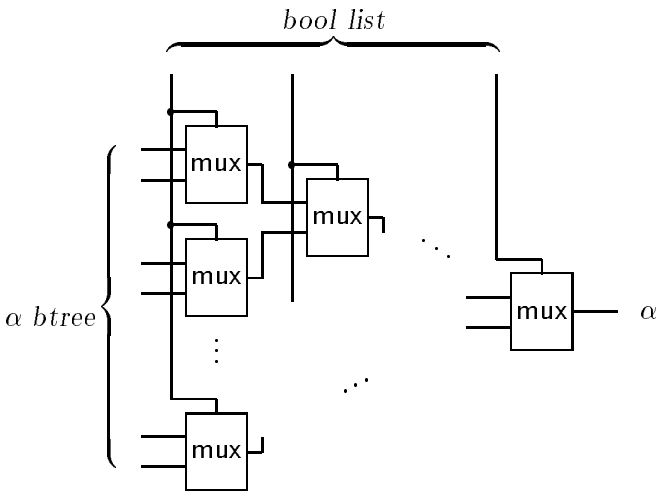

Abbildung 5.11: 2^n:1–multiplexer

Der Datentyp *btree* wird wie folgt definiert:

```
primitive_datatype " btree = Bleaf of 'a | Bnode of btree # btree ";
```

Das Steuersignal repräsentiert eine Adresse, die einen Pfad des Binärbaums beschreibt. Ein `F` in der Adresse steht für ,,links" und `T` für ,,rechts". Beispiel: Die Eingangssignale haben die folgenden Werte:

$s = [\mathsf{F}, \mathsf{T}]$
$a = \mathsf{Bnode}(\mathsf{Bnode}(\mathsf{Bleaf}\, b)(\mathsf{Bleaf}\, c))(\mathsf{Bnode}(\mathsf{Bleaf}\, d)(\mathsf{Bleaf}\, e))$

Dann ergibt sich am Ausgang:

$y = c$

Abbildung 5.12 veranschaulicht diesen Zusammenhang.

In diesem Beispiel ist die Tiefe des Binärbaumes konstant und die Länge der Liste stimmt mit der Baumtiefe überein. Für solche Fälle ist die Schaltung auch eigentlich konzipiert. Da in PML nur totale Funktionen zugelassen sind, müssen jedoch bei einer PML–Implementierung auch alle anderen möglichen Fälle berücksichtigt werden. Dazu wird die bisherige Funktionsbeschreibung ergänzt:

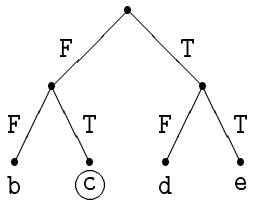

Abbildung 5.12: Auswahl eines Elements aus dem Binärbaum

- Es werden die Werte der Liste nur soweit berücksichtigt, bis man im Binärbaum auf ein Blatt stößt. Wäre in obigem Beispiel $s = [\mathsf{F}, \mathsf{T}, \mathsf{F}, \mathsf{T}]$ dann soll ebenfalls $y = c$ sein.
- Endet die Pfadbeschreibung nicht in einem Blatt, sondern in einem Knoten des Binärbaums, dann soll das am weitesten links stehende Blatt in diesem Teilbaum ausgewählt werden. Wäre in obigem Beispiel $s = [\mathsf{F}]$ dann soll $y = b$ sein.

Abbildung 5.13 zeigt das Rekursionsschema von bmux. In Abhängigkeit von den Strukturen der Eingangssignalen werden drei Fälle unterschieden. Besteht der Baum lediglich aus einem Blatt ($y = \mathsf{Bleaf}\, a$), dann wird der Inhalt des Blattes mit dem Ausgang verbunden. Setzt sich der Baum aus zwei Teilbäumen b und c zusammen und ist die Liste leer, dann wird das am weitesten links stehende Element aus b ausgewählt. Setzt sich der Baum aus zwei Teilbäumen b und c zusammen und besteht die Liste aus einem Kopfelement h und einer Restliste t, dann werden mit t in den Teilbäumen zwei Elemente ausgewählt, zwischen denen dann mit h ausgewählt wird.

Das folgende Gleichungssystem beschreibt die Funktion bmux:

$$
\begin{array}{lcl}
\mathsf{bmux}(x, \mathsf{Bleaf}\, a) & = & a \\
\mathsf{bmux}(\mathsf{Nil}, \mathsf{Bnode}\, b\, c) & = & \mathsf{bmux}(\mathsf{Nil}, b) \\
\mathsf{bmux}(\mathsf{Cons}\, h\, t, \mathsf{Bnode}\, b\, c) & = & \mathsf{mux}(h, \mathsf{bmux}(t, b), \mathsf{bmux}(t, c))
\end{array}
$$

Bei der Umsetzung dieser Funktion in eine PML–Implementierung muß diese verschachtelte primitive Rekursion über *list* und *btree* entflechtet werden. Dazu wird dieses Gleichungssystem in mehreren Schritten durch Äquivalenzumformungen umgeformt:

$$
\begin{array}{lcl}
\mathsf{bmux}(x, \mathsf{Bleaf}\, a) & = & a \\
\mathsf{bmux}(\mathsf{Nil}, \mathsf{Bnode}\, b\, c) & = & \mathsf{bmux}(\mathsf{Nil}, b) \\
\mathsf{bmux}(\mathsf{Cons}\, h\, t, \mathsf{Bnode}\, b\, c) & = & \mathsf{mux}(h, \mathsf{bmux}(t, b), \mathsf{bmux}(t, c)) \\
 & & \Downarrow \\
\mathsf{bmux}(\mathsf{Nil}, \mathsf{Bleaf}\, a) & = & a \\
\mathsf{bmux}(\mathsf{Nil}, \mathsf{Bnode}\, b\, c) & = & \mathsf{bmux}(\mathsf{Nil}, b) \\
\mathsf{bmux}(\mathsf{Cons}\, h\, t, \mathsf{Bleaf}\, a) & = & a \\
\mathsf{bmux}(\mathsf{Cons}\, h\, t, \mathsf{Bnode}\, b\, c) & = & \mathsf{mux}(h, \mathsf{bmux}(t, b), \mathsf{bmux}(t, c))
\end{array}
$$

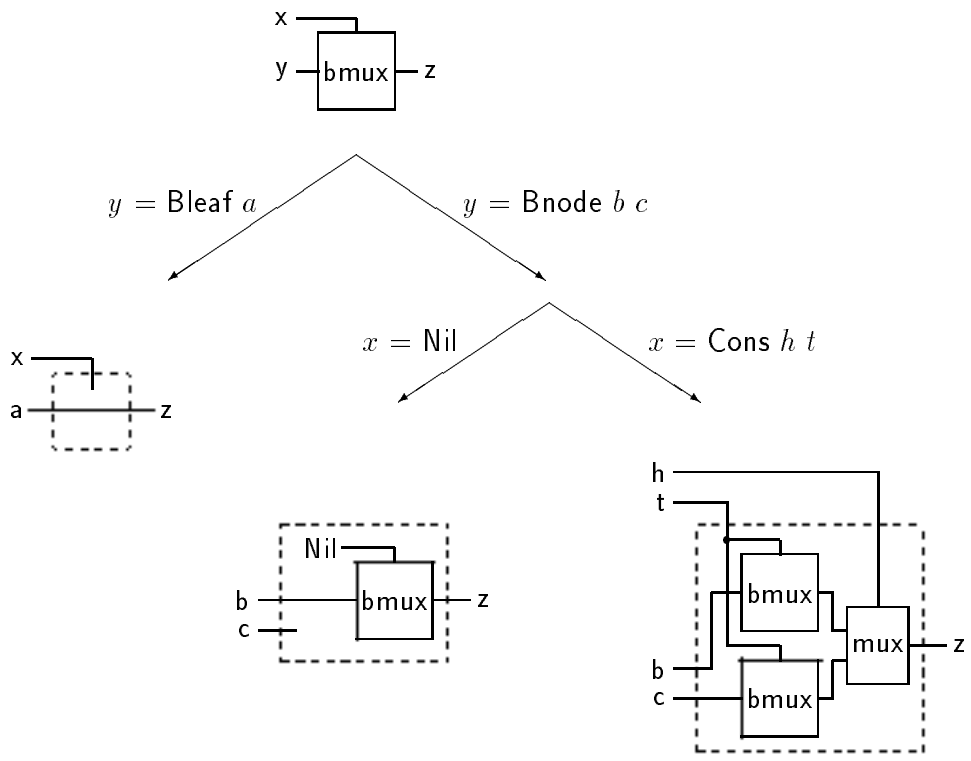

Abbildung 5.13: Rekursionsschema von bmux

$$\Downarrow$$

$$\begin{aligned}
\mathsf{bmux}(\mathsf{Nil}, y) &= \mathsf{PRIMREC_btree}\ y\ (\lambda a.\, a)\ (\lambda a\, b\, v\, w.\, v) \\
\mathsf{bmux}(\mathsf{Cons}\ h\ t, y) &= \mathsf{PRIMREC_btree}\ y\ (\lambda a.\, a) \\
&\qquad (\lambda a\, b\, v\, w.\, \mathsf{mux}(h, \mathsf{bmux}(t, v), \mathsf{bmux}(t, w)))
\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}
\mathsf{bmux}(x, y) = {} & \mathsf{PRIMREC_list}\ x \\
& (\lambda s.\, \mathsf{PRIMREC_btree}\ s\ (\lambda a.\, a)\ (\lambda a\, b\, v\, w.\, v)) \\
& (\lambda h\, t\, r\, s. \\
& \quad \mathsf{PRIMREC_btree}\ s\ (\lambda a.\, a)\ (\lambda b\, c\, v\, w.\, \mathsf{mux}(h, r\ b, r\ c))\ ) \\
& y
\end{aligned}$$

Man kommt zu folgender PML–Implementierung:

```
fun bmux (x,(y:'a btree)) =
  PRIMREC_list x
    (fn s =>
      PRIMREC_btree s (fn a => a)
        (fn a => fn b => fn v => fn w => v) )
    (fn h => fn t => fn r => fn s =>
```

```
  PRIMREC_btree s (fn a => a)
    (fn b => fn c => fn v => fn w => mux(h,r b,r c)) )
y;
```

Anmerkungen

Im Beispiel von bmux wurde die Funktion zunächst durch ein Gleichungssystem im ML–Stil beschrieben, und dann wurde diese Spezifikation von Hand durch Äquivalenzumformungen in eine PML–Implementierung umgewandelt. Es stellt sich die Frage, ob es möglich ist, diese Umwandlung zu automatisieren.

Dazu ist zunächst anzumerken, daß bmux eine primitiv rekursive Funktion ist. Funktionsbeschreibungen im ML–Stil sind mächtiger. Mit ihnen können allgemeine μ–rekursive Funktionen beschrieben werden. Es wurde bereits erläutert, weshalb zur Beschreibung μ–rekursiver Funktionen eine andere Methodik gewählt. Vor diesem Hintergrund erscheint es nicht sinnvoll, die PML–Schreibweisen in μ–rekursive PML–Funktionen zu konvertieren.

Es stellt sich aber die Frage, ob es nicht möglich ist, jene Gleichungssysteme, die primitiv rekursive Funktionen beschreiben, automatisch in PML–Notation zu konvertieren. Einfach ist diese Umwandlung dann, wenn über einem einzelnen Datentyp primitive Rekursion betrieben wird. Beispiele dafür sind die Funktionen ser und par, die durch einfache primitive Rekursion über *num* bzw. *list* beschrieben werden. Das Gleichungssystem kann in diesen Fällen durch ein paar einfache, technische Umformungen an die Form der entsprechendenden PRIMREC–Funktionen angepaßt werden, und man kommt so direkt zur PML–Implementierung.[1]

Schwieriger gestaltet sich die Umsetzung verschachtelter primitiv rekursiver Funktionen wie im Beispiel von bmux. Das Argument von bmux hat den Typ $((bool)list, (\alpha)btree)prod$. Die PML–Implementierung erfolgt durch eine geschachtelte primitive Rekursion über den Typen *bool*, *list*, *btree* und *prod*. Bisher müssen diese Verschachtelungen von primitiver Rekursion von Hand aufgelöst werden. Ein möglicher Ansatz zur Bewältigung dieser Problematik könnte folgendermaßen aussehen: Aus der Semantik der Einzeldatentypen wird die Semantik des zusammengesetzten Datentyps abgeleitet. Die Semantik des zusammengesetzten Datentyps wird dabei mit dem gleichen Schema beschrieben wie die Semantik einzelner Datentypen. Die Semantik der zusammengesetzten Datentypen rechtfertigt dann die Einführung einer PRIMREC–Grundfunktion über dem zusammengesetzten Datentyp.

[1] In HOL existiert bereits ein Mechanismus, mit dem Funktionen durch Gleichungsysteme, die eine primitive Rekursion über genau einem Datentyp beschreiben, eingeführt werden können.

Kapitel 6

Beispielschaltungen

Die Systematik der funktionalen Schaltungsbeschreibung mit PML wird in diesem Kapitel an zwei größeren Schaltungen illustriert. Das erste Beispiel ist eine ,,Benchmarkschaltung" mit dem Namen Min–Max. Im zweiten Beispiel wird ein Mikroprozessor und dazu ein Assembler entworfen. Zunächst wir eine kleine Bilbliothek von Grundbausteinen definiert, auf denen dann die beiden Beispiele aufbauen. Die vollständigen PML–Programme befinden sich in Anhang D.

6.1 Die Bausteinbibliothek

Die Bausteinbibliothek besteht aus zwei Teilen. Im ersten Teil werden Grundschaltungen definiert, die direkt durch PML–Funktionen spezifiziert werden. Alle anderen Schaltungen bauen auf den Grundbausteinen auf. Die PML–Grundfunktionen sollen nur zur Spezifikation der Grundbausteine verwendet werden und sollen in den abgeleiteten Bausteinen nicht mehr verwendet werden.

Die Menge der Grundbausteine ist aus Gründen der Übersichtlichkeit möglichst klein gehalten. Der zweite Teil der Bausteinbibliothek erweitert die Bausteinsammlung um Bausteine, die aus den Grundbausteinen abgeleitet werden.

6.1.1 Grundbausteine

Alle Einzelsignale der Bausteinbibliothek haben den Typ *bool*. Zur Bündelung der Signale konkreter Schaltungen werden Paare (*prod*) verwendet. Für abstrakte Schaltungen werden zusätzlich Listen (*list*) und Binärbäume (*btree*) zur Signalbündelung verwendet. Die Datentypen *bool*, *prod* und *list* sind in PML bereits vordefiniert. Der Datentyp *btree* wird wie folgt definiert:

```
primitive_datatype "btree = Bleaf of 'a | Bnode of btree # btree";
```

Außerdem wird noch der Datentyp unit definiert:

```
primitive_datatype "unit = Unit";
```

Der Datentyp *unit* beschreibt eine einelementige Menge. Das einzige Element dieser Menge heißt Unit. Diesem Datentyp soll die Rolle eines Dummies zukommen: Wenn in einer polymorphen Basteinfunktion eine Signalleitung mit einem variablen Typ nicht benötigt wird, dann soll diese Typvariable durch *unit* instanziiert werden. Signale vom Typ *unit* transportieren keine Information und können somit in der realen Schaltung weggelassen werden.

Für die Bezeichnung der Bausteine soll durchgängig die folgende Konvention eingehalten werden:

- Funktionen, die Schaltnetze darstellen, beginnen weder mit x, noch mit y, noch mit z.
- Übergangsschaltnetze beginnen mit dem Anfangsbuchstaben x.
- Funktionen, die (gemeinsame) Aus– und Übergangsschaltnetze darstellen, beginnen mit dem Anfangsbuchstaben y.
- Schaltwerke, die zeitabhängige Eingangssignale auf zeitabhängige Ausgangssignale abbilden, beginnen mit dem Anfangsbuchstaben z.

Den Kern der Grundbausteinbibliothek bildet der polymorphe 2:1–Multiplexer mux. Alle konkreten Bausteine, die aus Bausteinen der Grundbausteinbibliothek zusammengesetzt werden, können in einfacher Weise durch Expansion der Funktionsdefinitionen auf 1–Bit 2:1–Multiplexer zurückgeführt werden. Der 1–Bit 2:1–Multiplexer ist eine Ausprägung von mux.

Die abstrakten Bausteine map, map_btree, row, bmux, bdx, apply_f_on_addressed_bleaf und combine_btrees beschreiben reguläre Schaltungsstrukturen. Die übrigen Grundbausteine dienen ausschließlich der Verdrahtung. Jede Ausprägung dieser Bausteine kann allein durch elektrische Verbindungen und ohne einen „echten" Baustein realisiert werden. Formal drückt sich dies dadurch aus, daß diese Funktionen bei konkreten Schaltungen durch Äquivalenzumformungen eliminiert werden können. Die Namen der Verdrahtungsbausteine orientieren sich an den in ML für diese Funktionen üblichen Bezeichnungen.

6.1.2 Abgeleitete Bausteine

Die Bibliothek der abgeleiteten Bausteine teilt sich in mehrere Gruppen auf:

- Varianten der Reihung row
- Boolesche Operatoren
- Arithmetische Funktionen
- D–Flipflop und RAM (als Aus– und Übergangsschaltnetze)

6.2 Die Min–Max–Schaltung

Die Min–Max–Schaltung ist durch eine informelle, natürlichsprachliche Spezifikation gegeben:

> Min–Max hat ein Eingangssignal *input*, das eine Zahlenfolge mit Werten im Bereich von -256 bis +255 darstellt. Min–Max hat drei boolesche Kontrollsignale *clear*, *reset* und *enable*. Die Min–Max Schaltung erzeugt im gleichen Takt wie *input* ein Ausgangssignal *output*, das wie folgt definiert ist:
>
> - Wenn *clear* T ist, dann ist *output* Null — unabhängig von den anderen Kontrollsignalen.
> - Sind *clear* und *enable* F, dann nimmt *output* den Wert an, den *input* als letztes hatte, bevor *enable* F wurde.
> - Ist *clear* F und sind *enable* und *reset* T, dann nimmt *output* die Werte von *input* an.
> - Wenn *reset* F wird, dann nimmt *output* zu jedem Zeitpunkt t den Mittelwert aus Maximum und Minimum der Zahlenwerte von *input* an, die seitdem anlagen. Es gilt dann
>
> $$output = \frac{max(input)+min(input)}{2}$$

Der in der Aufgabenstellung geforderte Zahlenbereich führt zu 9–Bit breiten Bitvektoren. Die Implementierung der Schaltung erfolgt in zwei Schritten. Zunächst wird eine abstrakte Schaltung entworfen, bei der die Bitbreite variabel ist. Die Zahlen werden durch Bitlisten vom Typ $(bool)list$ in Zweierkomplementdarstellung kodiert. In einem zweiten Schritt wird dann aus der abstrakten n–Bit–Schaltung die 9–Bit Variante der Schaltung abgeleitet.

Diese Vorgehensweise hat zwei Vorteile: Zum einen können aus der abstrakten n–Bit–Schaltung ohne großen Aufwand auch andere Schaltungen mit anderen Werten für n abgeleitet werden und zum anderen können Induktionsbeweise über der abstrakten Schaltung geführt werden.

Abbildung 6.1 zeigt die gewählte Implementierung. Diese Implementierung baut auf Bausteinen der Bausteinbibliothek auf. Lediglich die Teilschaltungen zminn, zmaxn und averagen sind neue, zusammengesetzte Schaltungen. Die Schaltung averagen berechnet den Mittelwert zweier Eingangssignale. Die Schaltwerke zminn und zmaxn ähneln einander: zminn berechnet das Minimum der Eingangssignale und zmaxn das Maximum der Eingangssignale. Beide Schaltungen können durch einen Steuereingang zurückgesetzt werden. Die beiden Teilschaltungen unterscheiden sich in ihrem Aufbau nur an genau einer Stelle: dort wo bei zminn ein Schaltnetz das Minimum aus bisherigem Minimum und aktuellem Eingangssignal berechnet, muß bei der zmaxn–Schaltung das Maximum berechnet werden. Statt zu beiden Schaltungen direkt die Übergangsschaltnetze xminn und xmaxn zu konstruieren, wird zunächst ein allgemeineres, abstraktes Übergangsschaltnetz xchoice

mit dem Parameter f definiert, bei der an der Stelle der Schaltung, an dem später entweder ein Maximum–Schaltnetz (maxn) oder ein Minimum–Schaltnetz (minn) zu stehen kommt, die Variable f steht. Man erhält dann xminn und xmaxn als Ausprägungen von xchoice: xminn ist (xchoice minn) und xmaxn ist (xchoice maxn).

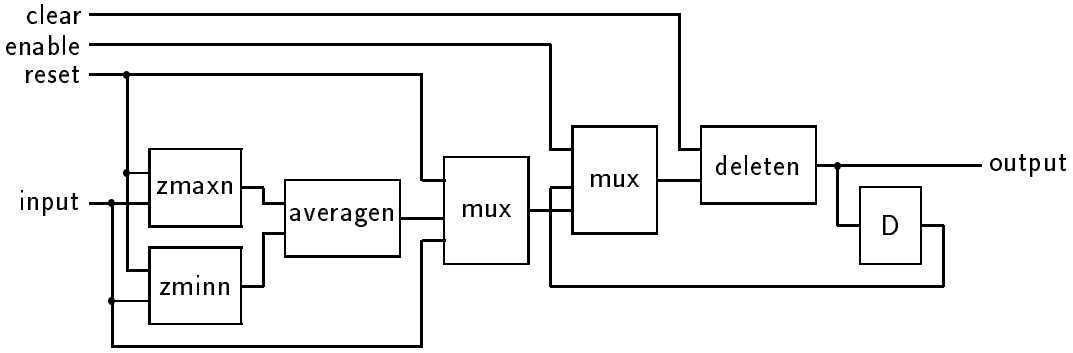

Abbildung 6.1: Schaltungsstruktur von Min–Max

6.3 Der Mikroprozessor und der Assembler

6.3.1 Der Mikroprozessor

Die Systembusschnittstelle des Mikroprozessors gliedert sich in Adreßbussignal *ab*, Datenbussignal *db* und read/write–Signal *rw* (siehe Abbildung 6.2). Auf den Adreßbus darf lediglich der Mikroprozessor schreibend zugreifen. Der Datenbus ist bidirektional. Mit rw wird festgelegt, wer auf den Datenbus schreibt: der Mikroprozessor (rw = F) oder die Systembusteilnehmer (rw = T). Die Breite von Adress– und Datensignal ist nicht festgelegt, es wird jedoch gefordert, daß die Breite der beiden Bitvektoren gleich ist.

Der Mikroprozessor setzt sich aus zwei Teilen zusammen: dem Rechenwerk (operation unit) und dem Steuerwerk (control unit). Das Rechenwerk meldet dem Steuerwerk den Befehl, der sich aktuell im Befehlsregister *ir* (instruction register) befindet, und den Zustand der arithmetischen–logischen Verknüpfungseinheit (ALU). Der Zustand der ALU setzt sich aus zwei Komponenten zusammen: *zero* gibt an, ob die letzte Operation den Funktionswert 0 hatte und *carry* gibt an, ob es bei der letzten Operation zu einem Überlauf kam.

Das Steuerwerk kann das Rechenwerk über die Signale *pcinc* (program counter increment), *absel* (address bus select), *aluop* (ALU operation), *idbw* (internal data bus write) und *idbr* (internal data bus read) steuern (siehe auch Abbildung 6.3). Durch setzen des *pcinc*–Signals kann der Befehlszähler *pc* (program counter) um eins erhöht werden. Mit *absel* wird gesteuert, ob der Befehlszähler *pc* oder das Adreßregister *ar* (address register) auf den Adreßbus geschaltet wird. Das Signal *aluop* bestimmt, welche Operation in der

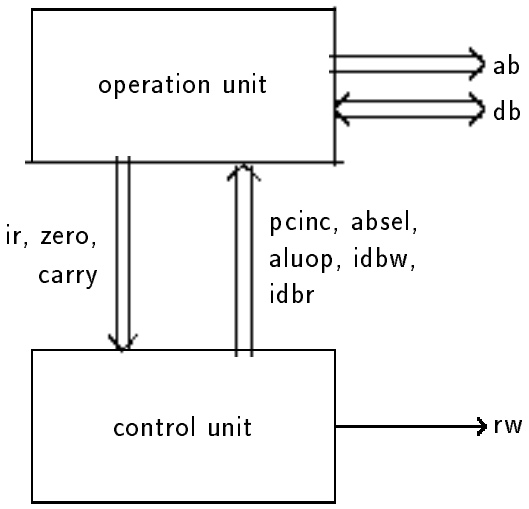

Abbildung 6.2: Grobstruktur des Mikroprozessors

ALU ausgeführt wird und ob überhaupt eine Operation ausgeführt wird. Die ALU kennt zwei Operationen: die Addition und die Negation. Zur Übertragung der Daten zwischen den Registern hat das Rechenwerk einen internen Datenbus (IDB). Der IDB wird immer so benutzt, daß genau ein Teilnehmer schreibend und auch genau ein Teilnehmer lesend auf ihn zugreift. Die Signale *idbw* (IDB–write) und *idbr* (IDB–read) legen fest, welche Teilnehmer schreibend bzw. lesend auf den IDB zugreifen. Der IDB hat sieben Teilnehmer: den Befehlszähler *pc*, das Adreßregister *ar*, den (externen) Datenbus, das Befehlsregister *db*, den Akkumulator *accu*, das Hilfsregister *auxr* und die ALU.

Abbildung 6.4 zeigt den Aufbau des Steuerwerks. Im Mittelpunkt des Steuerwerks steht ein 3–Bit–Zähler mit der aktuellen Nummer des Taktes des Befehlszyklus, der gerade bearbeitet wird. Die Befehlszyklen unterteilen sich in mehrere Maschinenzyklen. Jeder Maschinenzyklus wird in genau einem Takt abgearbeitet. Wieviele Maschinenzyklen bzw. Takte für die Abarbeitung eines bestimmten Befehls benötigt werden, wird im ROM–Baustein instruction_cycles_rom gespeichert. Die Schaltung equn vergleicht Zähler und Länge des Befehlszyklus. Erreicht der Zähler den letzten Takt des Befehlszyklus, dann wird er automatisch zurückgesetzt, und es kann dann der nächste Befehl begonnen werden. Der Zähler hat genau einen Eingang. Über diesen Eingang wird der Zähler zurückgesetzt. Der ROM–Baustein instruction_rom berechnet die Steuersignale des Steuerwerks. Als Adresse dienen der aktuelle Zustand des Zählers, der Inhalt des Befehlsregister *ir* und der Zustand der ALU (*zero*,*carry*).

Neben den Steuersignalen für das Rechenwerk berechnet das Steuerwerk noch das *rw*–Signal für den Systembus. Dieses Signal wird aus dem *idbw*–Signal abgeleitet. Der externe Datenbus ist Teilnehmer am internen Datenbus und hat dort die Adresse 0. Hat das *idbw* signal den Wert 0, dann wird der Wert des externen Datenbusses auf den internen Datenbus geschaltet. Das ist genau dann der Fall, wenn der Mikroprozessor lesend auf

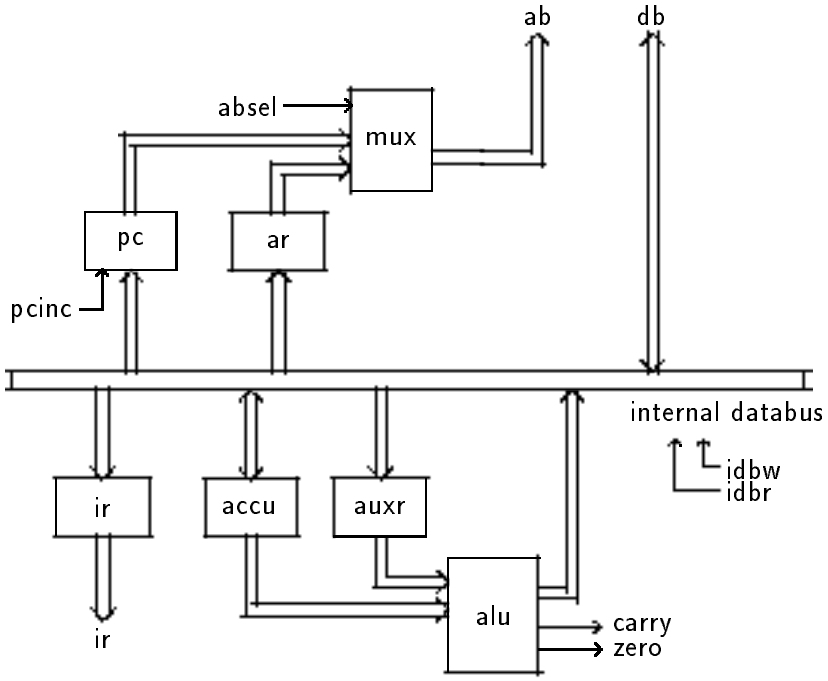

Abbildung 6.3: Rechenwerk des Mikroprozessors

den Systembus zugreift.

Der Mikroprozessor kennt zwölf verschiedene Befehle (Tabelle 6.1). Der Befehlssatz gliedert sich in vier Gruppen: Lade– und Speicherbefehle, Sprungbefehle, arithmetische Operationen und eine leere Anweisung. Jeder der Befehle besteht aus einem Befehlswort und aus einem Argument. Mit Ld wird der Inhalt der adressierten Speicherzelle in den Akkumulator geladen, mit Ldc wird eine Konstante in den Akkumulator geladen und mit LDI wird eine indirekt adressierte Speicherzelle in den Akkumulator geladen. Mit St wird der Inhalt des Akkumulator an der gegeben Adresse gespeichert, mit Sti wird der Akkumulator an der Adresse gespeichert auf der der Parameter zeigt. Den Sprungbefehl gibt es mit direkter Adressierung (Jmp) und indirekter Adressierung (Jmpi). Die Befehle Jmpz und Jmpc führen, in Abhängigkeit von *zero* bzw. *carry*, verzweigte Sprünge mit direkter Angabe der Sprungadresse durch. Die Befehle Add und Inv führen die Addition bzw. Negation durch. Bei der Addition gibt der Paramter die Adresse des Wertes im Speicher an, der auf den Akkumulator aufaddiert werden soll. Bei der Negation hat der Parameter keine Bedeutung. Der Befehl Nop hat ebenfalls einen Parameter ohne Bedeutung. Die Ausführung von Nop hat keine Auswirkungen.

Die Abarbeitung der Befehle teilt sich in zwei Phasen auf. Tabelle 6.2 beschreibt die erste Phase. Die erste Phase dauert genau drei Takte und ist für alle Befehle gleich. In Takt 0 wird die Adresse des Befehlszählers auf den Adreßbus gelegt, um das nächste Befehlswort einzulesen. Der Befehlszähler wird am Ende von Takt 0 inkrementiert. Die Information darüber, daß am Ende eines Taktes der Befehlszähler inkrementiert werden

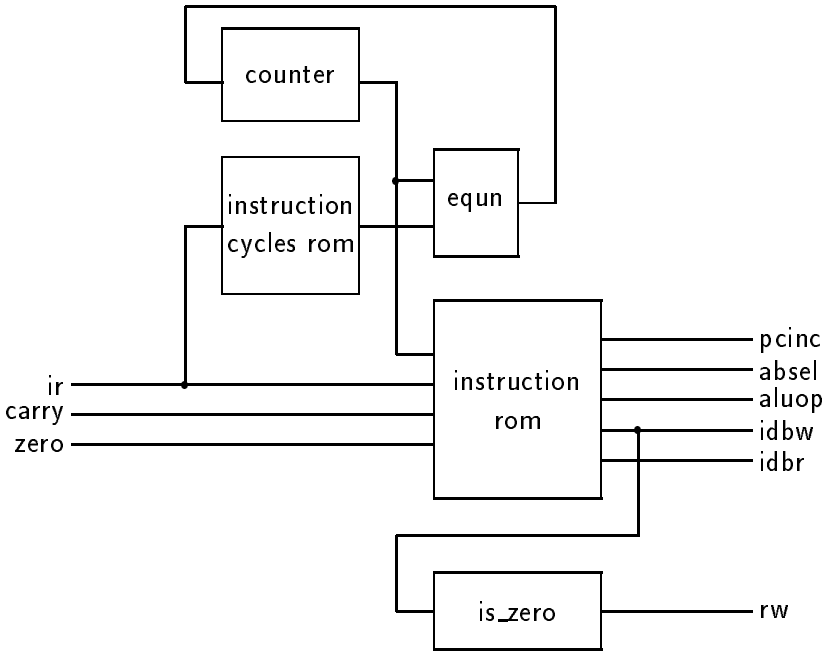

Abbildung 6.4: Steuerwerk des Mikroprozessors

muß, wird aus *absel* gewonnen. Immer wenn mit *absel* der Befehlszähler auf den Adreßbuß geschaltet wird, soll der Befehlszähler am Ende des Taktes inkrementiert werden. In Takt 1 liegt auf dem (externen) Datenbus das Befehlswort an. Das Befehlswort wird über den internen Datenbus in das Befehlsregister transportiert. In Takt 2 wird der Operand des Befehls angefordert, und der Befehlszähler wird anschließend inkrementiert. In welches Register der Operand, der dann in Takt 3 auf dem externen Datenbus anliegt, abgespeichert wird, hängt von der Art des Befehls ab, der sich seit Takt 1 im Befehlsregister befindet.

Tabelle 6.3 zeigt, wie die weitere Abarbeitung der Befehle verläuft, abhängig davon, welcher Befehl in Takten 1 in das Befehlsregister *ir* geladen wurde. Handelt es sich beispielsweise um einen Ld–Befehl, dann wird in Takt 3 der Operand in das Adreßregister transportiert, in Takt 4 wird diese Adresse auf den Adreßbus geschaltet und in Takt 5 wird das adressierte Speicherwort in den Akkumulator transportiert. Die Ausführung der meisten Befehle ist unabhängig von *zero*– und *carry*–Flag. Lediglich für die bedingten Sprünge Jmpz und Jmpc gibt es hier eine Fallunterscheidung: sind die entsprechenden Flags gesetzt, dann wir der Sprung ausgeführt ansonsten nicht.

Um den Mikroprozessor implementieren zu können, muß aus dieser eher informellen Beschreibung des Befehlszyklen die Kodierung für die ROM–Bausteine des Steuerwerks abgeleitet werden. Man könnte diesen Vorgang nun von Hand machen und dann die ROM–Bausteine mit den entstandenen Zahlentabellen personalisieren. Der Entwurf soll jedoch auf einer höheren Stufe geschehen. Die oben beschriebene Tabelle soll in Form von

Ld *address*	load	accu := mem[*address*]
Ldc *constant*	load constant	accu := *constant*
Ldi *address*	load indirect	accu := mem[mem[*address*]]
St *address*	store	mem[*address*] := accu
Sti *address*	store indirect	mem[mem[*address*]] := accu
Jmp *address*	jump	goto *address*
Jmpi *address*	jump indirect	goto mem[*address*]
Jmpz *address*	jump if zero	if zero then goto *address*
Jmpc *address*	jump if carry	if carry then goto *address*
Add *address*	add	accu := accu + mem[*address*]
Inv *dummy*	invert	accu := - accu
Nop *dummy*	no operation	—

Tabelle 6.1: Befehlssatz des Mikroprozessors

PML–Funktionen abstrakt beschrieben werden, und dann soll in einem zweiten Schritt daraus innerhalb von PML eine Kodierung abgeleitet werden. Diese Vorgehensweise hat zwei Vorteile: Zum einen wird dadurch der Entwurf sicherer. Der ROM–Baustein hat 512 Speicheradressen mit a 10 Bit. Der zweite Vorteil bei dieser abstrakten Formalisierung besteht darin, daß ein Teil des Entwurfes, die Beschreibung der Befehlsmenge und die Kodierung der Befehle, im Entwurf des Assemblers wiederverwendet werden kann und wird.

Es wird ein Datentyp *mp_instruction* eingeführt, der für die Menge der Befehlsnamen steht. Der Datentyp *idbpart* steht für die Menge der Teilnehmer des internen Datenbusses, der Datentyp *flag* beschreibt die Menge der Flags, der Datentyp *readwrite* beschreibt die

Clock	absel	rd/wr	aluop	idbw	idbr
0	PC	read	-	-	-
1	-	-	-	DB	IR
2	PC	read	-	-	-

Tabelle 6.2: Steuersignale in den Takten 0, 1 und 2

	Clock	ir	zero	carry	absel	rw	aluop	idbw	idbr
Ld	3	Ld	-	-	-	-	-	Db	Ar
	4	Ld	-	-	Ar	Read	-	-	-
	5	Ld	-	-	-	-	-	Db	Accu
Ldc	3	Ldc	-	-	-	-	-	Db	Accu
Ldi	3	Ldi	-	-	-	-	-	Db	Ar
	4	Ldi	-	-	Ar	Read	-	-	-
	5	Ldi	-	-	-	-	-	Db	Ar
	6	Ldi	-	-	Ar	Read	-	-	-
	7	Ldi	-	-	-	-	-	Db	Accu
St	3	St	-	-	-	-	-	Db	Ar
	4	St	-	-	-	-	-	Accu	Db
	5	St	-	-	Ar	Write	-	-	-
Sti	3	Sti	-	-	-	-	-	Db	Ar
	4	Sti	-	-	Ar	Read	-	-	-
	5	Sti	-	-	-	-	-	Db	Ar
	6	Sti	-	-	-	-	-	Accu	Db
	7	Sti	-	-	Ar	Write	-	-	-
Jmp	3	Jmp	-	-	-	-	-	DB	PC
Jmpi	3	Jmp	-	-	-	-	-	Db	Ar
	4	Jmp	-	-	Ar	Read	-	-	-
	5	Jmp	-	-	-	-	-	Db	Pc
Jmpz	3	Jmp	F	-	-	-	-	-	-
		Jmp	T	-	-	-	-	Db	Pc
Jmpc	3	Jmp	-	F	-	-	-	-	-
		Jmp	-	T	-	-	-	Db	Pc
Add	3	Jmp	-	-	-	-	-	Db	Auxr
	4	Jmp	-	-	-	-	Add	ALU	ACCU
Inv	3	Inv	-	-	-	-	Inv	Alu	Accu
Nop	3	-	-	-	-	-	-	-	-

Tabelle 6.3: Befehlszyklen ab Takt 3 in Abhängigkeit des geladenen Befehls

möglichen Signalwerte des *rw*–Signals und der Datentyp *alu_operation* die Menge der möglichen Ansteuerungen der ALU. Mit dem Datentyp *mp_instruction* sollen einzelne Maschinenzyklen (Takte) beschrieben werden. Es werden zwei verschiedene Formen von Maschinenzyklen: Der gewöhnliche Maschinenzyklus Simply und der bedingte Sprung Conditional_jump, unterschieden. Zum gewöhnlichen Maschinenzyklus wird angegeben, welches Adreßsignal auf den Adreßbus geschaltet wird, ob der Mikroprozessor lesend oder schreiben auf den Datenbus zugreift, welche Operation die ALU ausführt und welche Teilnehmer lesend bzw. schreibend auf den internen Datenbus zugreifen. Beim bedingten Sprung wird nur angegeben, von welchem Flag der bedingte Sprung abhängt.

Die abgeleiteten Konstanten

get_pc, get_ar, put_ar, db_to_ir, db_to_accu, db_to_ar, db_to_auxr, db_to_pc, sum_to_accu, inv_to_accu, wait, jump_if_carry, jump_if_zero

haben alle den Typ *cycle_description*. Durch diese Konstanten werden die verschiedenen in der Tabelle vorkommenden Maschinenzyklen beschrieben. Die Funktion

get_mp_instruction_description

weist jedem Befehl eine Liste von Maschinenzyklen zu. Dabei werden enthalten die Listen nur die Maschinenzyklen ab Takt 3.

Im nächsten Schritt werden Kodierungen für die Adressen der IDB–Teilnehmer, für die Maschinenbefehle, für das *rw*–Signal, für die ALU–Ansteuerung etc. festgelegt. Aus diesen Kodierungen wird dann der Inhalt der ROM–Bausteine des Steuerwerks abgeleitet.

6.3.2 Der Assembler

Der Assembler ist eine Funktion, die einem Assemblerprogramm einen RAM–Speicher–Anfangszustand zuordnet. Der Assembler hat nichts mehr mit Hardware–Beschreibungen zu tun. Die hier Assemblerfunktionen sollen nicht mehr als Schaltung, sondern ganz allgemein als programmiersprachliche Beschreibung einer Abbildung verstanden werden, und die Strukturen der Funktionsdefinitionen sollen nicht mehr als Schaltungsstrukturen, sondern ganz allgemein als Kontrollstrukturen interpretiert werden.

Die Assemblersprache kennt vier verschiedene Konstrukte: Instr, Value, Address und Label. Als ein Assemblerprogramm soll eine Liste aus solchen Konstrukten verstanden werden. Maschinenbefehle werden mit der Anweisung

Instr *mp_instruction operand_type num*

angegeben. Der erste Parameter ist der Name des Befehls (Ld, Ldi, etc.). Der zweite Parameter gibt an, ob der Operand direkt durch einen Zahenlwert oder durch eine Marke angegeben wird. Er kann die Werte Num bzw. Lab annehmen. Der dritte Parameter ist der Operand des Maschinenbefehls — eine natürliche Zahl, die in Abhängigkeit des Wertes des zweiten Parameters direkt als Zahlwert oder als Marke interpretiert werden soll.

Mit der Anweisung

Value *num*

wird an der aktuellen Adreßposition der Wert, der als Parameter angegeben wird, im Speicher abgelegt.

Folgen von Instr– und Value–Anweisungen werden, in kodierter Form, hintereinander im Speicher abgelegt. Instr–Anweisungen belegen immer genau zwei Speicherzellen: eines für das Befehlswort, das andere für den Operanden. Value–Anweisungen belegen immer genau eine Speicherzellen. Die Anfangsadresse für eine nachfolgende Folge von Isntr– und Value–Anweisungen wird mit dem folgenden Konstrukt festgelegt:

Address *num*

Innerhalb eines Maschinenprogramms dürfen auch Marken stehen, die in den Instr–Anweisungen referenziert werden dürfen. Die Syntax für Marken lautet:

Label *num*

Die Übersetzung des Programms geschieht in zwei Durchläufen. Im ersten Durchlauf wir zu jeder Marke und zu jeder Instr– und Value–Anweisung die Adresse berechnet. Im zweiten Durchlauf werden die Instr– und Value–Anweisungen kodiert und an der berechneten Stelle im Speicher abgelegt.

Kapitel 7

Verifikation

Aufgabe der Verifikation ist es, für eine gegebene Schaltung nachzuweisen, daß sie eine gegebene Spezifikation erfüllt. Die Schaltungsimplementierung sei, gemäß dem vorgestellten Formalisierungsschema, durch eine primitiv rekursive PML–Funktion $\mathsf{f}_{\alpha\to\beta}$ gegeben. Die Spezifikation ist eine allgemeine Formel, die eine beliebige Relation zwischen Ein– und Ausgangssignalen herstellt. Die Spezifikation sei durch ein Prädikat $\mathsf{S}_{\alpha*\beta\to\mathsf{bool}}$ vorgegeben. Das Beweisziel der Verifikation lautet damit:

$$\vdash\ \forall x\, y.\, (\mathsf{f}\, x = y) \Rightarrow \mathsf{S}(x, y)$$

Äquivalent dazu ist:

$$\vdash\ \forall x.\, \mathsf{S}(x, \mathsf{f}\, x)$$

Es stellt sich jetzt die Frage, wie ein solcher Beweis geführt werden kann und inwieweit es möglich ist, diesen Beweis zu automatisieren. Offensichtlich wird die Antwort auf diese Frage von der Form von S abhängen. Während man von der Schaltungsimplementierung bereits weiß, daß sie durch eine totale, berechenbare Funktion f repräsentiert wird, die Eingangssignale auf Ausgangssignale abbildet, wurden für die Spezifikation S bisher noch keine Einschränkungen gemacht. Für beliebige S läßt sich der Beweis nicht automatisieren. Denn wäre dieser Beweis automatisierbar, dann könnte man die Allgemeingültigkeit einer beliebigen geschlossenen Formel P automatisch beweisen, indem man zu einem $\mathsf{S}(x, y) = \mathsf{P}$ und einem beliebigen f das obige Beweisziel beweist. Das widerspricht jedoch der Tatsache, daß die Allgemeingültigkeit der Prädikatenlogik höherer Ordnung nicht entscheidbar ist.

Ein interessante Variante der Verifikation besteht darin, daß die Spezifikation ebenfalls durch eine primitiv rekursive PML–Funktion $s_{\alpha\to\beta}$ beschrieben wird. In diesem Fall lautet das Beweisziel:

$$\vdash\ \forall x\, y.\, (\mathsf{f}\, x = y) \Rightarrow (\mathsf{s}\, x = y)$$

Äquivalent dazu sind

$$\vdash\ \forall x.\, \mathsf{f}\, x = \mathsf{s}\, x$$

und

$\vdash$ f = s

Die Aufgabe besteht dann also darin, die Gleichheit zweier Funktionen zu beweisen. Dies ist ein Punkt, der über die Verifikation hinaus noch eine andere Anwendung hat: Die Gleichheit von Funktionen auf logischer Ebene bildet die Grundlage für korrekte Umformungen in PML. Das führt zu korrekten Transformationen über Hardwarebeschreibungen, wie sie für die Synthese und Optimierung benötigt werden.

7.1 Zielorientiertes Beweisen

Um die Gültigkeit einer Formel zu beweisen sie mit Hilfe des HOL–Kalküls abgeleitet werden. Der HOL–Kalkül besteht aus fünf Axiomen und acht Regeln. Axiome sind Theoreme; weitere Theoreme können abgeleitet werden, indem mit Regeln aus bestehenden Theoremen neue Theoreme abgeleitet werden.

Es stellt sich die Frage, wie man zu einem konkret vorgegebenen Beweisziel den Beweisweg finden kann. Eine Möglichkeit besteht darin, bei den Axiomen anfangend, sukzessiv weitere Theoreme abzuleiten und dadurch die Menge der bewiesenen Theoreme solange zu vergrößern, bis man auch das zu beweisende Theorem erreicht hat. Diese Technik wird als *Vorwärtsbeweis* bezeichnet: Ausgangspunkt sind die Axiome und aus ihnen werden weitere Theoreme und insbesondere auch das zu beweisende Theorem abgeleitet.

Eine andere Beweistechnik wird als *Rückwärtsbeweis* oder auch als *zielorientiertes Beweisen* bezeichnet. Beim zielorientierten Beweisen ist der Ausgangspunkt das zu beweisende Theorem. Ziel des Rückwärtsbeweises ist es, das Beweisziel auf Axiome *zurückzuführen*. Die Beweiszielmenge ist am Anfang eine eine Menge, welche nur eine Formel, das zu beweisende Theorem enthält. In jedem Schritt des Beweises wird eine Formel t aus der Menge ausgewählt und durch eine Menge von Formeln $t_1, t_2, \ldots t_n$ (Teilziele) ersetzt. Gleichzeitig muß für diesen Schritt eine Regel angegeben werden, mit der dann aus den Theoremen $\vdash t_1, \vdash t_2, \ldots \vdash t_n$ später das Theorem $\vdash t$ abgeleitet werden kann. Befinden sich schließlich in der Beweiszielmenge nur noch Axiome, dann ist der Beweis erbracht, denn dann können mit den Regeln die einzelnen Teilziele und schließlich das Gesamtbeweisziel (vorwärts) abgeleitet werden.

Bei der Vorwärtsbeweistechnik ergibt sich das Problem, daß man schwer abschätzen kann, ob die Theoreme, die man in den Zwischenschritten abgeleitet hat, in die richtige Richtung führen. Bei der Technik des zielorientierten Beweisens besteht dagegen das Problem darin, auf Teilziele zu stoßen, die Axiome sind. Im schlimmsten Fall entstehen widersprüchliche Teilziele, die nicht abgeleitet werden können. Daß weder die eine noch die andere Technik zu einem Algorithmus führen kann, der die Gültigkeit beliebiger Formeln ableiten kann, folgt bereits aus der Nichtentscheidbarkeit der Prädikatenlogik höherer Ordnung.

7.2 Evaluierung

Während der Beweisführung wird sich mehrfach die Notwendigkeit ergeben, mit der Schaltungsfunktion f für bestimmte Eingangssignale die Ausgangssignale zu berechnen. Im einfachsten Fall sieht ein Beweis folgendermaßen aus: f beschreibt ein Schaltnetz mit zeitunabhängigen, booleschen Ein– und Ausgangssignalen, und auch die Spezifikation ist durch eine solche Funktion s gegeben. Der gesamte Beweis besteht dann aus:

1. Fallunterscheidung über der endlichen Menge der möglichen Eingangssignale
2. Evaluierung der Funktionen und
3. Vergleich der Funktionswerte.

Die Evaluierung kann auf der Ebene der Verifikation genau so ablaufen, wie dies in einem Interpreter für eine funktionale Programmiersprache geschieht. Die Grundmechanismen sind:

- β–Reduktion
- Expansion von Funktions– und Konstantendefinitionen
- Auswertung der Grundfunktionen

Die β–Reduktion ist eine Regel des HOL–Kalküls. Sie ist als eine ,,Conversion" implementiert, die einem β–Redex $(\lambda x.p[x])\,y$ das Theorem $\vdash\ (\lambda x.p[x])\,y = p[y]$ zuordnet. Bei der Definition von Funktions– und Konstantendefinitionen entstehen Definitionsgleichungen als Theoreme. Funktionsdefinitionen

$$f\ x_1 \ldots x_n = \Phi[x_1 \ldots x_n]$$

können nach dem Funktor aufgelöst werden

$$f = \lambda x_1 \ldots x_n.\, \Phi[x_1 \ldots x_n]$$

und haben dann die gleiche Form wie Konstantendefinitionen. Per Substitution (eine weitere Grundregel des HOL–Kalküls) können, bei vorgegebenen Konstantendefinitionsgleichungen, Konstanten in Theoremen gemäß ihrer Definitionsgleichung expandiert werden. Die CASE–Grundfunktionen können in entsprechende PRIMREC–Grundfunktionen umgewandelt werden, und für die PRIMREC–Grundfunktionen gibt es ein Gleichungssystem mit dem PRIMREC–Terme in Abhängigkeit des Terms, über dem primitive Rekursion betrieben wird, vereinfacht werden können. Diese Umformungen können per Termersetzung (eine abgeleitete, zusammengesetzte Regel) durchgeführt werden.

Die Evaluierung μ–rekursiver Funktionen ist i.a. nicht berechenbar. Eine Evaluierungsfunktion, die einen μ–rekursiven Term immer dann berechnen kann, wenn er terminiert, der Funktionswert also (Defined y) ist, ist dagegen berechenbar. Man kommt zu einer solchen Evaluierungsfunktion, wenn man für die Grundfunktion WHILE die folgende Beziehung benutzt:

$$
\begin{array}{l}
(\mathsf{WHILE}\ g\ f\ x) = \\
\quad \mathsf{PRIMREC\_bool}\ (g\ x) \\
\quad\quad (\mathsf{PRIMREC\_partial}\ (f\ x)\ (\lambda y.\,(\mathsf{WHILE}\ g\ f\ y))\ \mathsf{Undefined})
\end{array}
$$

Die Reihenfolge, in der die Schritte β–Reduktion, Expansion von Funktions– und Konstantendefinitionen und die Auswertung der Grundfunktionen geschieht, ist nicht festgelegt. Das Ergebnis ist davon unabhängig. Nicht unabhängig von der Abarbeitungsreihenfolge ist dagegen der Aufwand und die Effizienz des Evaluierungsalgorithmus. Hier bestehen Spielräume für die Optimierung.

Eine Möglichkeit besteht darin, zunächst alle Funktions– und Konstantendefinitionen zu expandieren, dann alle β–Redizes per β–Reduktion zu eliminieren und schließlich die verbliebenen Grundfunktionen auszuwerten. Nach der Expansion der Funktions– und Konstantendefinitionen besteht der Term nur noch aus Grundfunktionen.[1] Bei der β–Reduktion stößt man auf einen Nachteil dieser Methode: In β–Redizes $((\lambda x.\,p[x])\,y)$, bei denen die Variable in dem Teilterm $p[x]$ nicht frei vorkommt, wären Funktions– und Konstantendefinitionen in y gar nicht notwendig gewesen, da y bei der β–Reduktion verschwindet. Auf einen weiteren Nachteil stößt man bei der anschließenden Auswertung der Grundfunktionen: Kommt in einem β–Redex $((\lambda x.\,p[x])\,y)$ die Variable x in $p[x]$ mehrfach vor, dann muß der gleiche Term y mehrfach an verschiedenen Stellen ausgewertet werden.

Es gibt zahlreiche Algorithmen für verschiedene Formen der Evaluierung. Im wesentlichen unterscheidet man zwischen *Strict–Evaluierung* und *Lazy–Evaluierung* (siehe [Jone87]). Während bei der Strict–Evaluierung zunächst alle Parameter einer Funktion berechnet werden, bevor die Funktion selbst ausgewertet wird, werden bei einer Lazy–Evaluierung die Parameter erst so spät wie möglich berechnet. Durch eine Lazy–Evaluierung kann vermieden werden, daß ein Term ausgewertet wird, der später gar nicht mehr benötigt wird. Diesen Vorteil der Lazy–Evaluierung bezahlt man mit einem größeren Aufwand des Evaluierungs–Algorithmus. Innerhalb dieser groben Einteilung in Strict– und Lazy–Evaluierung gibt es noch zahlreiche weitere Freiheitsgrade, die sich auf die Evaluierungsgeschwindigkeit auswirken. Insbesondere bei der Lazy–Evaluierung gibt es deshalb mehrere verschiedene Algorithmen.

Hand in Hand mit der Evaluierung von λ–Termen geht deren Optimierung. Bereits bevor eine Funktion für bestimmte Werte evaluiert wird, können in der Funktion bereits Umformungen (β–Reduktion, Funktions– und Konstantenexpansion und Auswertung von Grundfunktionen) durchgeführt werden. Das zahlt sich dann aus, wenn eine Funktion für mehrere verschiedene Werte ausgewertet werden soll.

Üblicherweise können λ–Terme in einem Interpreter nur dann evaluiert werden, wenn sie keine freien Variablen haben. Diese Einschränkung gilt bei der Evaluierung in HOL nicht. Hier können Terme auch symbolisch ausgewertet werden. Die symbolische Evaluierung basiert dabei auf den gleichen Mechanismen wie die nichtsymbolische Evaluierung. Beispielsweise kann der Term $\mathsf{mux}(\mathsf{T}, \mathsf{inv}\ a, b)$ evaluiert werden. Das Ergebnis ist b. All-

[1] Anmerkung: Der Vorgang der Expansion von Funktions– und Konstantendefinition ist endlich, da jede Funktionsdefinition nur auf bereits definierte Funktionen zurückgreift und der neu zu definierende Funktor nicht auf der rechten Seite der eigenen Funktionsdefinition vorkommt.

gemeiner ausgedrückt, ist das Ergebnis einer symbolischen Evaluierung ein zu dem evaluierenden Term äquivalenter, einfacherer Term. Im Gegensatz zu einer nichtsymbolischen Evaluierung ist das Ergebnis einer symbolischen Evaluierung syntaktisch nicht eindeutig. Ein zweiter Unterschied ist der, daß das Ergebnis i.a. nicht nur Konstruktoren, sondern auch Funktionen enthält. Beispielsweise könnte das Ergebnis von $\mathsf{mux}(x, \mathsf{inv}\,\mathsf{T}, y)$ $x \wedge y$ sein, genausogut aber auch $(\mathsf{CASE\_bool}\ x\ \mathsf{F}\ y)$ oder $(\mathsf{CASE\_bool}\ y\ \mathsf{F}\ x)$ etc..

Es gibt noch einen zweiten Unterschied zwischen der Evaluierung in einem normalen Interpreter und den Umformungen in PML: In PML–Interpretern wird üblicherweise das Ergebnis einer Evaluierung nicht dargestellt, wenn das Ergebnis eine Funktion ist. Der Grund ist der, daß sich Funktionen i.a. nicht eindeutig durch einen bestimmten λ–Term darstellen lassen. Bei Umformungen in HOL hat man diese Einschränkung nicht. Bei der Evaluierung eines Terms, dessen Ergebnis eine Funktion ist, werden die gleichen Grundmechanismen verwendet.

7.3 Induktion

Bisher wurden lediglich Definitionsgleichungen der Funktions– und Konstantendefinitionen sowie Theoreme der Grundfunktionen ausgenutzt. Bei der Beweisführung ist es, wenn man Fallunterscheidungen oder Induktionsbeweise führen möchte, jedoch auch erforderlich zu wissen, wie die Datentypen ,,aufgebaut" sind.

Die Semantik eines Datentyps ϑ wird durch das Theorem $\mathcal{D}(\vartheta)$ definiert. $\mathcal{D}(\vartheta)$ ist aus formaler Sicht vollkommen ausreichend, um den Datentyp zu charakterisieren und über ihn zu argumentieren. Dieses Theorem ist jedoch innerhalb einer zielorientierten Beweistechnik schwer anwendbar. Es soll deshalb aus diesem Theorem ein Induktionstheorem $\mathcal{I}(\vartheta)$ und daraus ein Induktionsmechanismus abgeleitet werden. Der Algorithmus, mit dem aus $\mathcal{D}(\vartheta)$ $\mathcal{I}(\vartheta)$ abgeleitet wird, ist bereits im *type definition package* von HOL implementiert. $\mathcal{I}(\vartheta)$ hat die folgende Form:

$$\vdash\ \forall P.\ \left(\bigwedge_{i=1}^{c} \forall\, x_1 x_2 \ldots x_{|\gamma^i|}.\ \left(\bigwedge_{j=1}^{r_i} P\ x_{\rho_{i,j}}\right) \Rightarrow P(\gamma^i\ x_1 x_2 \ldots x_{|\gamma^i|})\right) \Rightarrow (\forall x.P\,x)$$

Der Mechanismus sieht wie folgt aus: Ausgangspunkt ist eine zu beweisende Formel $\forall x.\,\Phi[x_\vartheta]$. Zunächst wird das Induktionstheorem $\mathcal{I}(\vartheta)$ mit $P = \lambda x.\Phi[x_\vartheta]$ spezialisiert. Das Ergebnis ist:

$$\vdash\ \left(\bigwedge_{i=1}^{c} \forall\, x_1 x_2 \ldots x_{|\gamma^i|}.\ \left(\bigwedge_{j=1}^{r_i} \Phi[x_{\rho_{i,j}}]\right) \Rightarrow \Phi[\gamma^i\ x_1 x_2 \ldots x_{|\gamma^i|}]\right) \Rightarrow (\forall x.\Phi[x])$$

Man erkennt, daß man das Beweisziel $\forall x.\Phi[x]$ beweisen kann, indem man zunächst

$$\vdash\ \left(\bigwedge_{i=1}^{c} \forall\, x_1 x_2 \ldots x_{|\gamma^i|}.\ \left(\bigwedge_{j=1}^{r_i} \Phi[x_{\rho_{i,j}}]\right) \Rightarrow \Phi[\gamma^i\ x_1 x_2 \ldots x_{|\gamma^i|}]\right)$$

beweist, um dann damit und mit dem spezialisierten Induktionstheorem per Modus Ponens das Beweisziel abzuleiten. Diese Beweisziele bestehen jeweils aus einer Induktionshypothese

$$\bigwedge_{j=1}^{r_i} \Phi[x_{\rho_{i,j}}]$$

und einer Induktionskonklusion

$$\Phi[\gamma^i\, x_1 x_2 \ldots x_{|\gamma^i|}]$$

Die Fallunterscheidung über der Struktur des Datentyps stellt einen Spezialfall der Induktion dar, der nicht gesondert implementiert werden muß. Bei der Fallunterscheidung wird lediglich auf die Induktionshypothese verzichtet. Sind die Teilziele ohne Induktionshypothese beweisbar, dann auch mit ihnen.

Der gesamte Induktionsmechanismus wird als eine Funktion implementiert, die zu einem vorgegebenen Beweisziel und einem Induktionstheorem die neuen Teilziele und eine Funktion, die aus den Teilzielen, das Gesamtziel ableitet, berechnet.

Für die vordefinierten Datentypen *bool*, *prod*, *list*, *num* und *partial* sehen die Induktionstheoreme folgendermaßen aus:

$$\vdash\ \forall P.\ ((P\ \mathsf{T}) \wedge (P\ \mathsf{F}))\ \Rightarrow (\forall x.\, P\ x)$$

$$\vdash\ \forall P.\ (\forall a\ b.\, P(\mathsf{Comma}\ a\ b))\ \Rightarrow (\forall x.\, P\ x)$$

$$\vdash\ \forall P.\ ((P\ \mathsf{Nil}) \wedge (\forall h\ t.\, (P\ t) \Rightarrow P(\mathsf{Cons}\ h\ t)))\ \Rightarrow (\forall x.\, P\ x)$$

$$\vdash\ \forall P.\ ((P\ \mathsf{Zero}) \wedge (\forall n.\, (P\ n) \Rightarrow P(\mathsf{Suc}\ n)))\ \Rightarrow (\forall x.\, P\ x)$$

$$\vdash\ \forall P.\ ((\forall a.\, P\ (\mathsf{Defined}\ a)) \wedge (P\ \mathsf{Undefined}))\ \Rightarrow (\forall x.\, P\ x)$$

Man erkennt, daß für jeden Konstruktor ein Teilziel entsteht. Für alle Argumente von Konstruktoren, die den Typ des Datentyps selbst haben, entsteht in diesem Teilziel eine Induktionsvoraussetzung. Nichtrekursive Datentypen wie *bool*, *prod* und *partial* haben keine Konstruktoren mit solchen Argumenten, also gibt es bei ihnen auch keine Teilziele mit Induktionsvoraussetzung. Bei diesen Typen entartet die Induktion zur Fallunterscheidung. Bei den rekursiven Datentypen muß es immer auch Konstruktoren geben, bei denen kein Argument den Typ des Datentyps hat. Die Teilziele zu diesen Konstruktoren bilden die Induktionsanfänge. Die Teilziele zu Konstruktoren mit Argumenten vom eigenen Datentyp bilden die Induktionsschritte.

7.4 Beispiel

Es soll jetzt ein Beispiel für ein Beweis gegeben werden, der mittels Induktion geführt wird. Gegeben seien die folgenden PML–Definitionen:

```
fun and2 (a,b) = PRIMREC_bool a b F;    (* a /\ b *)
fun or2 (a,b) = PRIMREC_bool a T b;     (* a \/ b *)
```

```
fun imp (a,b) = PRIMREC_bool a b T;    (* a => b *)

fun cas f (a,x) =
  PRIMREC_list x a (fn h => fn t => fn r => f(h,r));

fun andn x = cas and2 (T,x);
fun orn x = cas or2 (F,x);
fun nandn x = cas imp (F,x);

fun par f x =
  PRIMREC_list x Nil (fn h => fn t => fn r => Cons (f h) r);

fun twolevel f g = g (par f);
```

Die Funktionen and2, or2 und imp stehen für die zweistellige Konjunktion, die zweistellige Disjunktion bzw. die Implikation. Der abstrakte Baustein cas beschreibt die Kaskadierung einer Schaltungsfunktion f. Mit ihm werden die n–stellige Konjunktion andn, die n–stellige Disjunktion orn und die n–stellige Nand–Verknüpfung nandn definiert. Der abstrakte Baustein par beschreibt die Parallelschaltung einer Schaltungsfunktion f. Der abstrakte Baustein twolevel beschreibt eine zweistufige Schaltung bestehend aus einer n–fach parallelgeschalteten Schaltung f in Serie mit g (Abbildung 7.1).

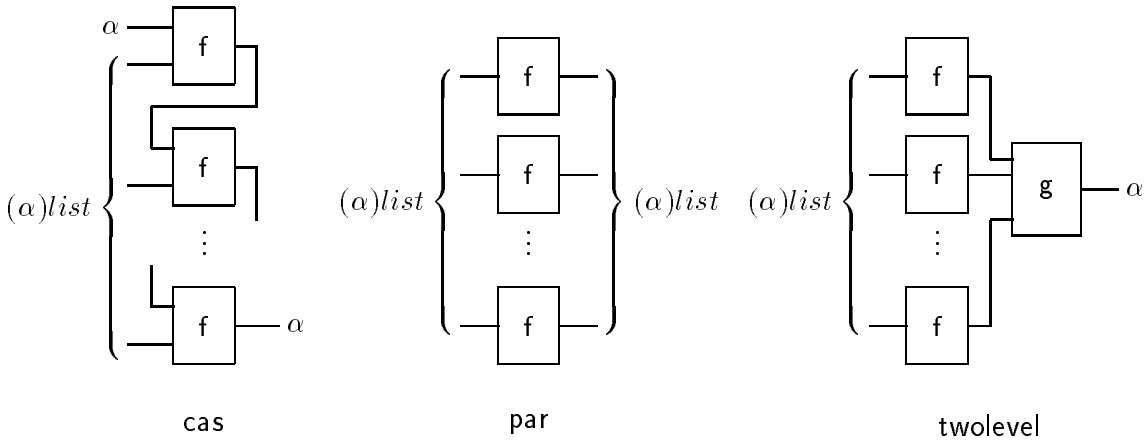

Abbildung 7.1: Die abstrakten Schaltungen cas, par und twolevel

Die Formel, deren Gültigkeit jetzt bewiesen werden soll, besagt, daß ein zweistufiges Schaltnetz bestehend aus einer And– und einer Or–Stufe äquivalent ist zu einem zweistufigen Schaltnetz bestehend aus zwei Nand–Stufen (Abbildung 7.2). Das Beweisziel:[2]

▷ twolevel andn orn = twolevel nandn nandn

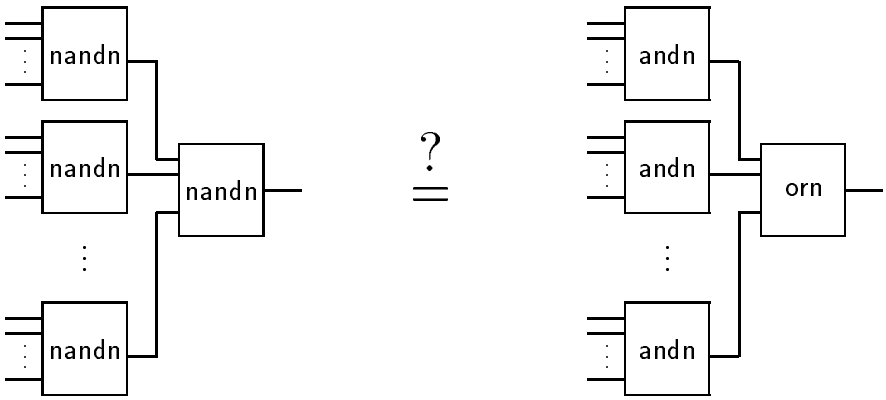

Abbildung 7.2: Beweisziel

Der Beweis soll durch zielorientierte Beweisführung erbracht werden. Zunächst wird eine aus dem Kalkül von HOL abgeleitete Regel (EXT) angewandt, die besagt, daß wenn zwei Funktionen an jeder Stelle äquivalent sind, daß dann die Funktionen äquivalent sind. Das neue Beweisziel lautet:

$$\triangleright\ \forall x.\ \mathsf{twolevel\ andn\ orn}\ x = \mathsf{twolevel\ nandn\ nandn}\ x$$

Die Variable x hat den Typ $((bool)list)list$. Der Beweis wird durch Induktion über dem Datentyp *list* geführt. Das Induktionstheorem für Listen lautet:

$$\vdash\ \forall P.\ ((P\ \mathsf{Nil}) \wedge (\forall h\, t.\, (P\ t) \Rightarrow P(\mathsf{Cons}\ h\ t))) \Rightarrow (\forall x.\, P\ x)$$

In diesem Theorem wird jetzt P so spezialisiert, daß die rechte Seite der Implikation dem Beweisziel entspricht. Dazu wird zunächst P durch

$$\lambda x.\ \mathsf{twolevel\ andn\ orn}\ x = \mathsf{twolevel\ nandn\ nandn}\ x$$

spezialisiert. Man erhält:

$$\begin{aligned}\vdash\ & (\ ((\lambda x.\ \mathsf{twolevel\ andn\ orn}\ x = \mathsf{twolevel\ nandn\ nandn}\ x)\ \mathsf{Nil}) \wedge \\ & \quad (\forall h\, t.\, (\lambda x.\ \mathsf{twolevel\ andn\ orn}\ x = \mathsf{twolevel\ nandn\ nandn}\ x)\ t \\ & \qquad \Rightarrow (\lambda x.\ \mathsf{twolevel\ andn\ orn}\ x = \mathsf{twolevel\ nandn\ nandn}\ x)\ (\mathsf{Cons}\ h\ t))\ ) \\ & \quad \Rightarrow (\forall x.\, (\lambda x.\ \mathsf{twolevel\ andn\ orn}\ x = \mathsf{twolevel\ nandn\ nandn}\ x)\ x)\end{aligned}$$

Per β–Reduktion wandelt man das spezialisierte Induktionstheorem um in:

$$\begin{aligned}\vdash\ & (\ (\mathsf{twolevel\ andn\ orn\ Nil} = \mathsf{twolevel\ nandn\ nandn\ Nil}) \wedge \\ & \quad (\forall h\, t.\, (\mathsf{twolevel\ andn\ orn}\ t = \mathsf{twolevel\ nandn\ nandn}\ t) \\ & \qquad \Rightarrow (\mathsf{twolevel\ andn\ orn\ (Cons}\ h\ t) = \mathsf{twolevel\ nandn\ nandn\ (Cons}\ h\ t))\ )\ ) \\ & \quad \Rightarrow (\forall x.\ \mathsf{twolevel\ andn\ orn}\ x = \mathsf{twolevel\ nandn\ nandn}\ x)\end{aligned}$$

[2] Mit $\triangleright$ soll ausgedrückt werden, daß es sich bei der nachfolgenden Formel um ein (noch unbewiesenes) Beweisziel handelt.

Auf der rechten Seite der Implikation steht jetzt das Beweisziel. Man kann den Beweis also dadurch führen, daß man die linke Seite der Implikation beweist. Die linke Seite besteht aus einer Konjunktion bestehend aus einem Induktionsanfang und einem Induktionsschritt. Die beiden neuen Beweisziele lauten:

- ▷ twolevel andn orn Nil = twolevel nandn nandn Nil
- ▷ $\forall h\, t.$ (twolevel andn orn t = twolevel nandn nandn t)
  $\Rightarrow$ (twolevel andn orn (Cons h t) = twolevel nandn nandn (Cons h t))

Das erste Teilziel, der Induktionsanfang, kann direkt durch Evaluierung bewiesen werden. Auf beiden Seiten der Gleichung stehen geschlossene Terme. Das Ergebnis der Evaluierung ist damit jeweils ein eindeutiger Term, der nur aus Konstruktoren besteht. Die Evaluierung ergibt sowohl bei (twolevel andn orn Nil) als auch für (twolevel nandn nandn Nil) F.

Auf das zweite Teilziel, den Induktionsschritt, werden zunächst ein paar einfache Umformungen (β–Reduktion, Expansion von Funktions– und Konstantendefinitionen, Auswertung von Grundfunktionen) angewandt:

- ▷ $\forall h\, t.$ (twolevel andn orn t = twolevel nandn nandn t)
  $\Rightarrow$ (or2 (andn h) (twolevel andn orn t) =
  imp (nandn h) (twolevel nandn nandn t))

Jetzt kann die Induktionshypothese ausgenutzt werden, indem (twolevelnandnnandn) durch (twolevel andn orn) ersetzt wird. In den weiteren Beweisschritten wird dann auf diese Induktionshypothese verzichtet.

- ▷ $\forall h\, t.$ (or2 (andn h) (twolevel andn orn t) = imp (nandn h) (twolevel andn orn t)

Im nächsten Schritt wird der Term (twolevel andn orn t) durch eine Variable b ersetzt. Dies bedeutet eine Verallgemeinerung. Das ursprüngliche Beweisziel kann aus dem neuen Beweisziel durch Spezialisierung abgeleitet werden.

- ▷ $\forall b\, h.$ or2 (andn h) b = imp (nandn h) b

Es folgt eine Induktion über b_{bool}. Analog zur vorangegangenen Induktion über *list* muß dazu das Induktionstheorem (hier: $\mathcal{I}(bool)$) angepaßt werden. Da der Datentyp *bool* nichtrekursiv ist, entartet diese Induktion zu einer Fallunterscheidung. Die beiden Fälle sind b = T und b = F.

- ▷ $\forall h.$ or2 (andn h) T = imp (nandn h) T
- ▷ $\forall h.$ or2 (andn h) F = imp (nandn h) F

Die beiden Terme (or2 (andn h) T) und (imp (nandn h) T) lassen sich jeweils zu T vereinfachen (β–Reduktion, Expansion von Funktions– und Konstantendefinitionen, Auswertung von Grundfunktionen). Damit ist das erste Teilziel bewiesen. Im zweiten Teilziel kann (or2 (andn h) F) zu (andn h) vereinfacht werden. Als Beweisziel verbleibt:

▷ $\forall h.\, \mathsf{andn}\ h = \mathsf{imp}\ (\mathsf{nandn}\ h)\ \mathsf{F}$

Als nächstes folgt eine Induktion über $h_{(bool)list}$. Sie verläuft analog zur ersten Induktion mit Hilfe des Theorems $\mathcal{I}(list)$, das wieder in geeigneter Weise angepaßt werden muß. Man erhält als Teilziele:

▷ $\mathsf{andn}\ \mathsf{Nil} = \mathsf{imp}\ (\mathsf{nandn}\ \mathsf{Nil})\ \mathsf{F}$

▷ $\forall p\, q\, (\mathsf{andn}\ q = \mathsf{imp}\ (\mathsf{nandn}\ q)\ \mathsf{F})$
$\Rightarrow (\mathsf{andn}\ (\mathsf{Cons}\ p\ q) = \mathsf{imp}\ (\mathsf{nandn}\ (\mathsf{Cons}\ p\ q))\ \mathsf{F})$

Das erste Teilziel, der Induktionsanfang, läßt sich direkt durch Evaluierung beweisen. Das zweite Teilziel wird umgeformt:

▷ $\forall p\, q\, (\mathsf{andn}\ q = \mathsf{imp}\ (\mathsf{nandn}\ q)\ \mathsf{F})$
$\Rightarrow (\mathsf{and2}\ p\ (\mathsf{andn}\ q) = \mathsf{imp}\ (\mathsf{imp}\ p\ (\mathsf{nandn}\ q))\ \mathsf{F})$

Als nächstes wird die Induktionshypothese auf die Induktionskonklusion angewandt:

▷ $\forall p\, q.\, \mathsf{and2}\ p\ (\mathsf{imp}\ (\mathsf{nandn}\ q)\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ p\ (\mathsf{nandn}\ q))\ \mathsf{F}$

Der Teilterm $(\mathsf{nandn}\ q)$ wird durch die boolesche Variable c ersetzt. Dadurch entsteht ein allgemeineres Beweisziel, das das ursprüngliche Beweisziel impliziert.

▷ $\forall c\, p.\, \mathsf{and2}\ p\ (\mathsf{imp}\ c\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ p\ c)\ \mathsf{F}$

Induktion über c_{bool} (boolesche Fallunterscheidung) führt zu:

▷ $\forall p.\, \mathsf{and2}\ p\ (\mathsf{imp}\ \mathsf{F}\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ p\ \mathsf{F})\ \mathsf{F}$

▷ $\forall p.\, \mathsf{and2}\ p\ (\mathsf{imp}\ \mathsf{T}\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ p\ \mathsf{T})\ \mathsf{F}$

Über beide Teilziele folgt eine Induktion jeweils über p_{bool} (boolesche Fallunterscheidung). Es entstehen vier Teilziele:

▷ $\mathsf{and2}\ \mathsf{T}\ (\mathsf{imp}\ \mathsf{F}\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ \mathsf{T}\ \mathsf{F})\ \mathsf{F}$

▷ $\mathsf{and2}\ \mathsf{F}\ (\mathsf{imp}\ \mathsf{F}\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ \mathsf{F}\ \mathsf{F})\ \mathsf{F}$

▷ $\mathsf{and2}\ \mathsf{T}\ (\mathsf{imp}\ \mathsf{T}\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ \mathsf{T}\ \mathsf{T})\ \mathsf{F}$

▷ $\mathsf{and2}\ \mathsf{F}\ (\mathsf{imp}\ \mathsf{T}\ \mathsf{F}) = \mathsf{imp}\ (\mathsf{imp}\ \mathsf{F}\ \mathsf{T})\ \mathsf{F}$

Alle vier Teilziele sind frei von Variablen. Die Evaluierung ergibt, daß im ersten Teilziel beide Seiten T, im zweiten Teilziel beide Seiten F, im dritten Teilziel beide Seiten F und im vierten Teilziel beide Seiten F sind. Der Beweis ist abgeschlossen.

7.5 Starke Induktion

Die gewöhnliche Induktion ist nur eine mögliche Technik zur Beweisfindung. In HOL ist lediglich diese Form der Induktion implementiert. Es soll jetzt ein anderer, mächtigerer Induktionsmechanismus vorgestellt werden, bei dem die Induktionsvoraussetzungen stärkere Aussagen machen als bei der gewöhnlichen Induktion.

Der Unterschied zwischen gewöhnlicher Induktion und starker Induktion soll zunächst am Beispiel der natürlichen Zahlen *num* erläutert werden. Das gewöhnliche Induktionstheorem für die natürlichen Zahlen lautet:

$$\vdash\ \forall P.\ ((P\ \mathsf{Zero}) \wedge (\forall n.\ (P\ n) \Rightarrow P(\mathsf{Suc}\ n))) \Rightarrow (\forall x.\ P\ x)$$

Genausogut könnte die Induktionsvoraussetzung $\forall m.\ m < n \Rightarrow (P\ m)$ statt $(P\ n)$ lauten.[3] Das Theorem für starke Induktion lautet:

$$\vdash\ \forall P.\ ((P\ \mathsf{Zero}) \wedge (\forall n.\ (\forall m.\ m < n \Rightarrow (P\ m)) \Rightarrow (P n))) \Rightarrow (\forall x.\ P\ x)$$

Es stellt sich jetzt die Frage, wie man diese Idee für beliebige Datentypen verallgemeinern kann. Spezifisch auf den Datentyp *num* abgestimmt ist die $<$–Relation. Für andere Datentypen wie Listen und Bäume müssen entsprechende Ordnungsrelationen gefunden werden und das starke Induktionstheorem muß dann jeweils explizit bewiesen werden.

Das Schema zur Erzeugung allgemeiner starker Induktionstheoreme, das jetzt vorgestellt werden soll, baut auf einem beliebigen, variablen Type α auf. α darf auch überabzählbar sein, darf also beispielsweise den Typ $num \rightarrow num$ haben. Zu α wird eine Maßfunktion $h_{\alpha \rightarrow num}$ definiert, die jedem Element x_α aus α eine Größe $(h\ x)_{num}$ zuordnet. Die Maßfunktion h teilt α in Äquivalenzklassen von Elementen gleicher Größe (Abbildung 7.3).

Der Beweis, daß alle x_α eine Eigenschaft P erfüllen, soll wie folgt erbracht werden: Es wird gezeigt, daß $(P\ x)$ für alle x_α mit $h\ x = 0$ gilt und es wird gezeigt, daß wenn $(P\ x)$ für alle x_α mit $h\ x \leq n$ gilt, daß dann auch $(P\ x)$ für alle x_α mit $h\ x = \mathsf{Suc}\ n$ gilt (Abbildung 7.4).

Das Theorem, das die allgemeine starke Induktion formal beschreibt lautet:

$$\begin{array}{l}\vdash\ \forall h.\ \forall P.\\ \quad (\ (\forall x.\ (h\ x = \mathsf{Zero}) \Rightarrow P\ x)\ \wedge\\ \quad\ (\forall x.\ (\forall y.\ (h\ y\ <\ h\ x) \Rightarrow (P\ y)) \Rightarrow (P\ x))\ )\\ \quad\quad \Rightarrow (\forall x.\ P\ x)\end{array}$$

Um dieses Theorem innerhalb eines Induktionsmechanismus anwenden zu können, muß es zunächst mit einer geeigneten Maßfunktion h spezialisiert werden. Nach der Spezialisierung hat das Theorem eine Form, die der Form der gewöhnlichen Induktionstheoreme ähnelt. Die Durchführung des starken Induktionsmechanismus läuft dann genau so ab wie die gewöhnliche Induktion: Spezialisierung von P, Beweis der Teilziele, Modus Ponens.

[3] Die kleiner– bzw. kleiner–gleich–Relation wird aus Gründen der Anschaulichkeit mit den Infixoperatoren $<$ bzw. $\leq$ dargestellt. Es handelt sich dabei um Funktionen vom Type $num * num \rightarrow bool$, die durch PML–Funktionsdefinitionen definiert werden können.

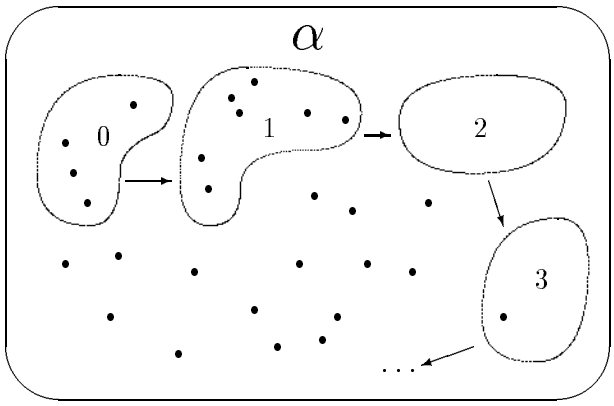

Abbildung 7.3: Aufteilung von α in Äuvilenzklassen mit Elementen gleicher Größe

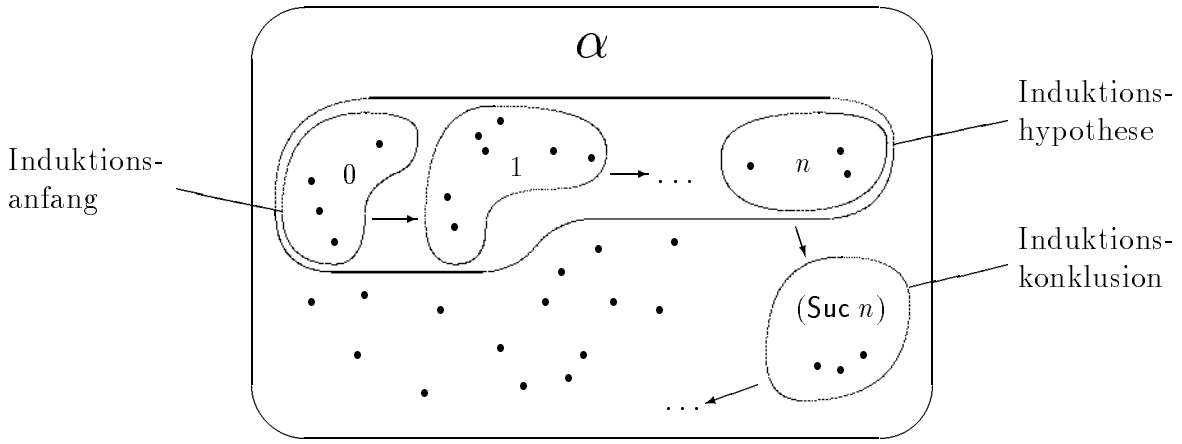

Abbildung 7.4: Starke Induktion

Durch Spezialisierung der Maßfunktion $h_{\alpha \to num}$ wird insbesondere auch der Typ α festgelegt, über dem die starke Induktion geführt wird. Im Gegensatz zur gewöhnlichen Induktion gibt es bei der starken Induktion nicht nur ein Induktionstheorem pro Datentyp, zumal die starke Induktion nicht nur über einzelnen Datentypen sondern über beliebigen, zusammengesetzten Typen geführt werden kann.

Beispiel: Es soll eine starke Induktion über dem Datentyp $(num)list$ geführt werden. Dazu muß eine geeignete Maßfunktion $h_{(num)list \to num}$ gefunden werden. Welche Funktion geeignet ist, hängt von dem Beweisziel ab, auf das die Induktion angewandt werden soll. Eine Möglichkeit besteht darin, die Länge der Liste length als Maßfunktion zu nehmen:

$$\begin{aligned} &\mathsf{length\ Nil} && = \mathsf{Zero} \\ &\mathsf{length\ (Cons}\ h\ t) && = \mathsf{Suc\ (length}\ t) \end{aligned}$$

Die Implementierung in PML lautet:

```
fun length x =
  PRIMREC_list x Zero (fn h => fn t => fn r => Suc r);
```

Durch diese Funktion wird die Menge der Listen natürlichen Zahlen in Partitionen gleich langer Listen aufgeteilt. Spezialisiert man das allgemeine starke Induktionstheorem mit length, dann kommt man zu:

$$\begin{array}{l}\vdash\ \forall P.\\ \quad (\ (\forall x.\,(\mathsf{length}\,x = \mathsf{Zero}) \Rightarrow P\,x)\ \wedge\\ \quad\ (\forall x.\,(\forall y.\,(\mathsf{length}\,y\ <\ \mathsf{length}\,x) \Rightarrow (P\,y)) \Rightarrow (P\,x))\ )\\ \quad\ \Rightarrow (\forall x.\,P\,x)\end{array}$$

Dieses Theorem läßt sich noch ein wenig vereinfachen:

$$\begin{array}{l}\vdash\ \forall P.\\ \quad (\ (P\ \mathsf{Nil})\ \wedge\\ \quad\ (\forall x.\,(\forall y.\,(\mathsf{length}\,y\ <\ \mathsf{length}\,x) \Rightarrow (P\,y)) \Rightarrow (P\,x))\ )\\ \quad\ \Rightarrow (\forall x.\,P\,x)\end{array}$$

7.6 Umwandlung in relationale Schaltungsbeschreibungen

Funktionale Schaltungsbeschreibungen können immer auch in relationale Schaltungsbeschreibungen umgewandelt werden. Der Beweiser MEPHISTO [KuSK93], mit dem die Verifikation bestimmter Schaltungsbeschreibungen vollständig automatisiert werden kann, arbeitet mit relationalen Schaltungsbeschreibungen. Schaltungsbeschreibungen, die für MEPHISTO verwendet werden sollen, müssen in einer ganz bestimmten Form vorliegen, die als *Hardwareformeln* bezeichnet werden.

In PML wurden Schaltwerkstrukturen auf Schaltnetzstrukturen mit Aus- und Übergangsschaltnetze zurückgeführt, da sich Schaltwerkstrukturen i.a. nicht funktional ausdrücken lassen. Durch Hardwareformeln können dagegen beliebige Schaltungsstrukturen ausgedrückt werden, und der Umweg über die Schaltnetzstruktrukturen ist nicht notwendig. Trotzdem ist der für die PML–Schaltungsstrukturen gewählte Weg über die Schaltnetze der Aus– und Übergangsfunktionen auch mit Hardwareformeln möglich. Um PML–Schaltungsstrukturen in Hardwareformeln zu konvertieren, müssen PML–Schaltnetzstrukturen in entsprechende Hardwareformeln zu konvertieren.

Abbildung 7.5 zeigt eine solche Konvertierung. Man erkennt, daß die Unterschiede durch eine einfache, rein syntaktische Umformung überwunden werden können. Die Funktionen der PML–Notation werden in die allgemeineren Relationen der Hardwareformeln umgesetzt, an die Stelle der Gleichheit zweier Terme rückt die Äquivalenz zweier Relationen, die Ein– und Ausgangsleitungen der Gesamtschaltung werden allquantifiziert und die inneren Leitungen, die in PML–Notation durch die lokalen Variablen der β–Redizes

```
fun fulladder (cin,(a,b)) =
  let val w1 = xor(a,b) in
  let val w2 = and(b,a) in
  let val sum = xor(cin,w1) in
  let val w3 = and(cin,w1) in
  let val cout = or(w3,w2) in
    (sum,cout)
  end end end end end;
```

$\longrightarrow$

$$
\begin{aligned}
&\forall t\, cin\, a\, b\, sum\, cout.\\
&\quad \mathsf{fulladder}(cin\, t, a\, t, b\, t, sum\, t, cout\, t) \Leftrightarrow\\
&\qquad \exists w1\, w2\, w3.\\
&\qquad\quad \mathsf{xor}(a\, t, b\, t, w1\, t) \wedge\\
&\qquad\quad \mathsf{and}(b\, t, a\, t, w2\, t) \wedge\\
&\qquad\quad \mathsf{xor}(cin\, t, w1\, t, sum\, t) \wedge\\
&\qquad\quad \mathsf{and}(cin\, t, w1\, t, w3\, t) \wedge\\
&\qquad\quad \mathsf{or}(w3\, t, w2\, t, cout\, t)
\end{aligned}
$$

Abbildung 7.5: Umwandlung PML–Schaltnetzstrukturbeschreibung $\rightarrow$ Hardwareformel

repräsentiert wurden, werden existenzquantifiziert. Die Signale von Schaltnetzen in Hardwareformeln werden — im Gegensatz zu dem gewählten PML–Ansatz — als von der Zeit t abhängige Funktionen dargestellt.

Bei PML–Schaltungsstrukturen werden die Aus– und Übergangsschaltnetze der Gesamtschaltung durch makeseq auf eine Schaltwerksfunktion abgebildet. Auf der Seite der Hardwareformeln ist es nicht notwendig, eine derartige Funktion explizit zu definieren. Aus– und Übergangsschaltnetzfunktionen können direkt miteinander verknüpft werden, und die Speicherkomponente wird durch eine sequentielle Leitung mit dem vorgegebenen Anfangszustand modelliert.

Kapitel 8

Zusammenfassung und Ausblick

Es wurde erläutert, wie auf einer kleinen funktionalen Sprache aufbauend, Schaltungen funktional beschrieben werden können. Es wurde gezeigt, wie dieser Ansatz zu Schaltungsbeschreibungen führt, die im logischen Sinne formal exakt sind und damit die Grundlage für eine Verifikation bilden können. Diese Schaltungsbeschreibungen stellen gleichzeitig ausführbare, funktionale Programme dar.

Neben der formalen Beschreibung real existierender Schaltungen und Schaltungsstrukturen wurden abstrakte Formen der Schaltungsbeschreibungen diskutiert und es wurde erläutert, wie Schaltungsstrukturbeschreibungen aussehen können, die über reine Netzlisten hinausgehen. Insbesondere wurde eine Methodik zur Formalisierung regulärer Schaltungsstrukturen mittels primitiver Rekursion eingeführt.

PML, als Basis für diesen Ansatz, ist zwar bereits eine höhere Programmiersprache, im Gegensatz zu anderen funktionalen Programmiersprachen wie SML besitzt sie jedoch nur sehr wenige Konstrukte und ist damit weniger anwenderfreundlich. Hier sind Erweiterungen denkbar. Beispielsweise könnte PML um weitere Pretty–Printing–Schreibweisen erweitert werden, und es könnten insbesondere Infixschreibweisen eingeführt werden. Bei den Erweiterungen ist darauf zu achten, daß die formale Verankerung nicht zu komplex werden sollte. Es empfiehlt sich deshalb, neue Konstrukte nicht durch die explizite Definition ihrer Semantik zu definieren, sondern sie auf den bereits bestehenden Konstrukten (als Schreibweisen) aufzubauen.

In bezug auf die Verifikation wurden bisher lediglich einige Techniken (Evaluierung, Fallunterscheidung, vollständige Induktion, starke Induktion) vorgestellt, und es wurde angedeutet, wie Beweise teilautomatisiert werden können. Auf der Ebene abstrakter Schaltungen ist eine vollständige Automatisierung nicht möglich. Ein interessantes Teilgebiet ist die Verifikation der Äquivalenz zweier Funktionen mittels Induktion. Da dabei als Induktionshypothesen nur Gleichungen auftreten, könnte ein modifizierter Knuth–Bendix–Algorithmus zur Umformung des Beweiszieles mit dem Termersetzungssystem, das sich aus Induktionshypothesen und Funktionsdefinitionsgleichungen zusammensetzt, verwendet werden.

Für die Verifikation konkreter Schaltungen wurde die Anbindung an MEPHISTO erläutert. Dazu müssen die funktionalen Schaltungsbeschreibungen in relationale Schaltungs-

beschreibungen konvertiert werden. Dieser Vorgang wurde bisher manuell durchgeführt — eine Automatisierung ist möglich, wurde aber noch nicht implementiert.

Bisher wurden PML–Programme nur in konventioneller Weise (nichtsymbolisch) evaluiert. Es wurde bereits angesprochen, daß auch eine symbolische Evaluierung innerhalb von HOL denkbar ist. Die Evaluierungsalgorithmen können in Form von Conversions implementiert werden. Interessant an dieser Form der Evaluierung ist auch der Aspekt, daß die Ausführung des Programms korrekt ist: Das Ergebnis wird nicht berechnet sondern bewiesen.

Ein anderes Gebiet, auf das ebenfalls noch nicht näher eingegangen wurde, ist die korrekte Umformung von Hardwarebeschreibungen. Es können in HOL Algorithmen in Form von Conversions geschrieben werden, die gegebene Schaltungen in äquivalente, aber „bessere" Schaltungen umformen. Die Korrektheit der Umformungen wird durch den HOL–Beweiser garantiert. Es können Algorithmen geschrieben werden, die Schaltungsbeschreibungen von einer höheren Beschreibungsebene auf äquivalente Schaltungen abbilden, die nur aus einfachen Grundschaltungen einer vorgegebenen Bibliothek bestehen. Interessant ist dies insbesondere für die High–Level–Synthese, da gerade hier komplizierte, korrektheitssensible Transformationen durchgeführt werden müssen. Es können Kostenfunktionen eingeführt werden, mit denen innerhalb des Syntheseprogramms die Schaltungen bewertet werden können. Die Kostenfunktionen können für die Grundbausteine der Bausteinbibliothek als Konstanten festgelegt werden und durch geeignete Funktionen können aus den PML–Strukturbeschreibungen dann die Kosten für zusammengesetzte Bausteine abgeleitet werden.

Anhang A

Syntax von PML

<Buchstabe> ::=
"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
"l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" |
"w" | "x" | "y" | "z" |
"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
"L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" |
"W" | "X" | "Y" | "Z" |
"_"

<Ziffer> :== "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<Bezeichner> ::= <Buchstabe> {<Buchstabe> | <Ziffer>}

<Zahl> ::= <Ziffer> {<Ziffer>}

<Typvariablenname> ::= "'"<Bezeichner>

<Typenkonstantenname> ::= <Bezeichner>

<Variablenname> ::= <Bezeichner>

<Konstruktorname> ::= <Bezeichner>

<AbgeleiteterKonstantenname> ::= <Bezeichner>

<Grundfunktionsname> ::=
    "WHILE" |
    "PRIMREC_"<Typenkonstantenname> |
    "CASE_"<Typenkonstantenname>

<Konstantenname> ::=
    <Konstruktorname> |
    <Grundfunktionsname> |
    <AbgeleiteterKonstantenname>

<Typname> ::=
    <Typvariablenname> |
    <Typenkonstantenname>

<Konstruktordeklaration> ::=
    <Konstruktorname> ["of" <Typname> {"#" <Typname>}]

<Konstruktordeklarationen> ::=
    <Konstruktordeklaration> {"|" <Konstruktordeklaration>}

<Datentypdeklaration> ::=
    "primitive_datatype "" <Typenkonstantenname> "=" <Konstruktordeklarationen> ""

<Typ> ::=
    <Typvariablenname> |
    ["(" <Typ> {"," <Typ>} ")"] <Typenkonstantenname> |
    <Typ> "*" <Typ> |
    <Typ> "->" <Typ>

<Konstante> ::= <Konstantenname>[":"<Typ>]

<Variable> ::= <Variablenname>[":"<Typ>]

<Variablenpaarung> ::=
    <Variable> |
    "(" <Variablenpaarung> {","<Variablenpaarung>} ")"

⟨*Term*⟩ ::=

⟨*Konstante*⟩ |
⟨*Variable*⟩ |
⟨*Term*⟩ ⟨*Term*⟩ |
"`fn`" ⟨*Variablenpaarung*⟩ "`=>`" ⟨*Term*⟩ |
"`(`" ⟨*Term*⟩ [","⟨*Term*⟩] "`)`" |
"`let val`" ⟨*Variablenpaarung*⟩ = ⟨*Term*⟩ "`in`" ⟨*Term*⟩ "`end`" |
"`[`" ⟨*Term*⟩ [","⟨*Term*⟩] "`]`" |
⟨*Zahl*⟩

⟨*Konstantendefinition*⟩ ::=

"`val`" ⟨*AbgeleiteterKonstantenname*⟩ "`=`" ⟨*Term*⟩ "`;`"

⟨*Argumentliste*⟩ ::= ⟨*Variablenpaarung*⟩ {⟨*Variablenpaarung*⟩}

⟨*Funktionsdefinition*⟩ ::=

"`fun`" ⟨*AbgeleiteterKonstantenname*⟩ ⟨*Argumentliste*⟩ "`=`" ⟨*Term*⟩ "`;`"

⟨*PML-Programm*⟩ ::=

{ ⟨*Datentypdeklaration*⟩ | ⟨*Konstantendefinition*⟩ | ⟨*Funktionsdefinition*⟩ }

Anhang B

PML–Konverter

Der Modul *Pml* bildet eine Erweiterung für HOL90.5 . Mit diesem Modul ist es möglich PML–Programme in Theoreme zu konvertieren, die ihre Semantik beschreiben. Nach dem Einladen des Moduls stehen die folgenden Funktionen zur Verfügung:

init

Bevor mit dem Modul gearbeitet werden kann, muß diese Funktion einmal aufgerufen werden. Durch die Ausführung der Funktion werden drei neue Theorien mit den Namen *adaptions*, *standard_types* und *while* erzeugt. Diese drei Theorien bilden die Grundlage für die Semantik von PML. Sie enthalten syntaktische Anpassungen (adaptions), die Semantik der vordefinierten Typen (standard_types) und die Semantik der WHILE–Funktion (while). Wo diese Theorien auf der Festplatte abgespeichert werden, hängt von *theory_path* ab.

extend_theory_by_pml_string

Diese Funktion erhält als Parameter eine Zeichenkette, die ein PML–Programm enthalten soll. Durch die Ausführung der Funktion wird die aktuelle Theorie um die Theoreme erweitert, die die Semantik des Programms beschreiben.

extend_theory_by_pml_file

Wie extend_theory_by_pml_string, nur daß das Programm in einer Datei abgelegt sein muß. Als Parameter wird der Dateiname angegeben.

dump

Mit dieser Funktion wird der aktuelle Stand angezeigt. Es werden alle PML–Konstanten, PML–Funktionen und PML–Datentypen zusammen mit den Theorien, in denen die Theoreme abgelegt wurden, aufgelistet. Durch den Parameter kann ausgewählt werden, ob die Datentypdeklarationen ebenfalls angezeigt werden sollen oder nicht.

defined_functions

Gibt in einer Liste die Namen der bisher definierten PML–Funktionen zurück.

theory_of_function

Bestimmt zu einem Namen einer PML–Funktion die zugehörige Theorie.

function_definition

Bestimmt zu einem Namen einer PML–Funktion das zugehörige Definitionstheorem.

defined_types

Gibt in einer Liste die Namen der bisher definierten PML–Typen zurück.

theory_of_type

Bestimmt zum Namen eines PML–Datentyps die zugehörige Theorie.

type_constructors

Bestimmt zum Namen eines PML–Datentyps die Liste der Namen der Konstruktoren.

```
signature PML =

   sig

      val  init                          : unit -> unit

      val  extend_theory_by_pml_string   : string -> unit
      val  extend_theory_by_pml_file     : string -> unit

      val  dump                          : bool -> unit

      val  defined_functions             : unit -> string list
      val  theory_of_function            : string -> string
      val  function_definition           : string -> thm

      val  defined_types                 : unit -> string list
      val  theory_of_type                : string -> string
      val  type_constructors             : string -> string list

   end;




structure Pml:PML =

   struct

      fun new_theory_kill_old name =
         (new_theory name)
            handle HOL_ERR _ =>
              (
                System.system
                   ("rm " ^ (hd (! theory_path)) ^ name ^ ".holsig");
                System.system
                   ("rm " ^ (hd (! theory_path)) ^ name ^ ".thms");
                new_theory name
              )

      fun mk_theory_adaptions () =

       let

         val _ = new_theory_kill_old "adaptions"
```

```
val COMMA_DEF =
   new_definition ("COMMA", (--‘Comma (a:’a) (b:’b) = (a,b)‘--))

val NIL_DEF  =
   new_definition ("NIL", (--‘Nil = (NIL:’a list)‘--))

val CONS_DEF =
   new_definition
      ("CONS", (--‘Cons = (CONS:’a -> ’a list -> ’a list)‘--))

val ZERO_DEF =
   new_definition ("ZERO", (--‘Zero = 0‘--))

val SUC_DEF  =
   new_definition ("SUC", (--‘Suc = SUC‘--))

val COMMA_DEF’ =  ( GEN_ALL o SYM o SPEC_ALL) COMMA_DEF

val lemma =
   prove(
      --‘ !f:’a-> ’b.!g. (f = g) = (!b. f b = g b)‘--
      ,
      REPEAT GEN_TAC THEN EQ_TAC THEN STRIP_TAC
      THENL
        [ ASM_REWRITE_TAC[],
          ACCEPT_TAC
             (EXT (ASSUME (--‘!b. (f:’a->’b) b = g b‘--))) ]
    )

val pair_lemma =
   (GEN_ALL o SYM o SPEC_ALL) (theorem "pair" "PAIR")

val prod_DEF’ =
   prove(
      --‘!(f:’a -> (’b -> ’c)). ?!fn. !x0 x1. fn (x0,x1) = f x0 x1‘--
      ,
      GEN_TAC THEN
      (REWRITE_TAC [definition "bool" "EXISTS_UNIQUE_DEF"]) THEN
      BETA_TAC THEN BETA_TAC THEN STRIP_TAC THENL
       [
         (EXISTS_TAC
            (--‘(\(x0,x1). (f:’a -> (’b -> ’c)) x0 x1)‘--)) THEN
         (CONV_TAC (DEPTH_CONV PAIRED_BETA_CONV)) THEN
         GEN_TAC THEN GEN_TAC THEN
         (ASM_REWRITE_TAC [])
```

```
          ,
          GEN_TAC THEN
          GEN_TAC THEN
          STRIP_TAC THEN
          (SUBST1_TAC
             (SPECL [--‘x:(’a # ’b) ->’c‘--, --‘y:(’a # ’b) ->’c‘--]
                (INST_TYPE
                   [ {residue = (==‘:’c‘==),
                      redex = (==‘:’b‘==)},
                     {residue = (==‘:(’a # ’b)‘==),
                      redex = (==‘:’a‘==)} ]
                   lemma)
             ))  THEN
          GEN_TAC THEN
          (ONCE_REWRITE_TAC [pair_lemma]) THEN
          (ASM_REWRITE_TAC[])
        ]
    )

val bool_DEF =
    prove (
       (--‘!(e0:’a) e1. ?!fn. (fn T = e0) /\ (fn F = e1)‘--),
       GEN_TAC THEN GEN_TAC THEN
       (REWRITE_TAC [definition "bool" "EXISTS_UNIQUE_DEF"]) THEN
       BETA_TAC THEN STRIP_TAC THENL
        [
          (EXISTS_TAC (--‘(\x. COND (x:bool) (e0:’a) e1)‘--)) THEN
          BETA_TAC THEN
          (REWRITE_TAC []) THEN
          GEN_TAC THEN GEN_TAC THEN
          BETA_TAC THEN
          (ASM_REWRITE_TAC [])
          ,
          GEN_TAC THEN
          GEN_TAC THEN
          BETA_TAC THEN
          STRIP_TAC THEN
          (SUBST1_TAC
             (SPECL [ --‘x:bool->’a‘--, --‘y:bool-> ’a‘--]
                (INST_TYPE
                   [ {residue = (==‘:bool‘==), redex = (==‘:’a‘==)},
                     {residue = (==‘:’a‘==), redex = (==‘:’b‘==)} ]
                   lemma)
             )) THEN
          GEN_TAC THEN
          (BOOL_CASES_TAC (--‘b:bool‘--)) THEN
```

```
            (ASM_REWRITE_TAC[])
          ] )

   val prod_DEF = ONCE_REWRITE_RULE [COMMA_DEF'] prod_DEF'

   val list_DEF =
      REWRITE_RULE
         [SYM CONS_DEF,SYM NIL_DEF]
         (theorem "list" "list_Axiom")

   val num_DEF =
      REWRITE_RULE
         [SYM ZERO_DEF,SYM SUC_DEF]
         (theorem "prim_rec" "num_Axiom")

in

       ( save_thm ("bool_DEF",bool_DEF);
         save_thm ("prod_DEF",prod_DEF);
         save_thm ("list_DEF",list_DEF);
         save_thm ("num_DEF",num_DEF);
         close_theory();
         export_theory() )

end




(**** Tables containing current PML--Definitions ****)


exception pml_definition_doesnt_exist

val pml_functions = ref ([] : (string * string) list)

fun add_function x =
   (pml_functions := (x,current_theory()) :: (! pml_functions))

fun theory_of_function x =
   ( (snd o hd o (filter (fn (a,b) => a = x)))
     (! pml_functions) )
   handle _ => raise pml_definition_doesnt_exist

fun defined_functions () = map fst (! pml_functions)
```

```
fun function_definition s = definition (theory_of_function s) s

val pml_types =
   ref ([] : (string * string * ((string * (string list)) list)) list)

fun add_type (tyname,def) =
   (pml_types := (tyname,current_theory(),def) :: (! pml_types))

fun remove_type tyname =
   (pml_types := (filter (fn (x,_,_) => x <> tyname) (! pml_types)))

fun defined_types() = map (fn (x,_,_) => x) (! pml_types)

fun lookup_type tyname =
  ( (hd o (filter (fn (a,_,_) => a = tyname)) ) (! pml_types) )
    handle _ => raise pml_definition_doesnt_exist




(**** Deriving the PRIMREC- and CASE-function definitions ****)


fun type_definition tyname = (fn (_,_,x) => x) (lookup_type tyname)

fun theory_of_type tyname = (fn (_,x,_) => x) (lookup_type tyname)

fun constructor_definition typename constname =
   ( (snd o hd o (filter (fn (x,_) =>
      (x = constname)))) (type_definition typename) )
   handle _ => raise pml_definition_doesnt_exist

fun kill_doubles [] = [] |
    kill_doubles (h::t) =
       if (mem h t) then
          (kill_doubles t)
       else
          (h::(kill_doubles t))

fun type_variables typename =
   ( kill_doubles o
     (filter (fn z => (hd (explode z)) = "'")) o
     flatten o
     (map snd) )
```

```
    (type_definition typename)

fun type_constructors typename = map fst (type_definition typename)

local
   fun mapnum' len f [] = [] |
       mapnum' len f (h::t) =
         (f (h,len - (length t)))  :: (mapnum' len f t)
in
   fun mapnum f l = mapnum' (length l) f l
end

fun constructor_position typename constname =
  (
   (snd o hd o (filter (fn (x,pos) => (pos<>0))))
   (mapnum (fn (x,pos) => if (x = constname) then (x,pos) else (x,0))
      (type_constructors typename))
  )
   handle _ => raise pml_definition_doesnt_exist

fun curried_type [] ty = ty |
    curried_type (h::t) ty =
      let val ty' = curried_type t ty in
         mk_type {Tyop = "fun", Args = [h,ty']}
      end

fun type_expression typename =
   mk_type
      { Tyop = typename,
        Args = map mk_vartype (type_variables typename) }

fun num_of_recursive_pars typename constname =
   length
      (filter
         (fn x => (x = typename))
         (constructor_definition typename constname))

fun positions_of_recursive_pars typename constname =
   ( (map snd) o  (filter (fn (x,_) => (x = typename))) )
   (mapnum
     (fn x => x)
     (constructor_definition typename constname))

fun multi n x = if (n = 0) then [] else (x::(multi (n-1) x))

val dummy_type = (==`:bool`==)
```

```
fun handler_arg_type_list recursive ty typename constname =
   let
      val tyl1 =
         map
            (fn s =>
               if (hd(explode s) = "'") then
                  mk_vartype s
               else type_expression s )
            (constructor_definition typename constname)
      val tyl2 = multi (num_of_recursive_pars typename constname) ty
   in
      if recursive then (tyl1 @ tyl2) else tyl1
   end

fun constructor_type typename constname =
   curried_type
      (handler_arg_type_list false dummy_type typename constname)
      (type_expression typename)

fun handler_type recursive ty typename constname =
   curried_type
      (handler_arg_type_list recursive ty typename constname)
      ty

local

   fun add_vars' n var_pre t =
      ( case ((dest_type o type_of) t) of
        {Tyop = _, Args = [ty1,ty2]} =>
           add_vars'
              (n + 1)
              var_pre
              (mk_comb
                 {Rator = t,
                  Rand =
                     mk_var{
                        Name = var_pre ^ (int_to_string n),
                        Ty = ty1} } )  |
          _           => t )
      handle _ => t

in

   val add_vars = add_vars' 1
```

```
end

fun constructor_term typename constname =
   mk_const{
      Name = constname,
      Ty = constructor_type typename constname }

fun ratorn n t = if (n = 0) then t else (rator (ratorn (n-1) t))

fun handler_type_list_rec typename =
   ( [type_expression typename] @
     (map
        (handler_type true (==':'y'==) typename)
        (type_constructors typename) ) )

fun const_term typename constname =
   add_vars "x"
      (mk_const{
         Name = constname,
         Ty = constructor_type typename constname})

fun PRIMREC_var_term typename cterm =
   ((add_vars "f") o
    (fn x => mk_comb {Rator = x, Rand = cterm}) o
    mk_var)
      {Name = "PRIMREC_" ^ typename,
       Ty =
          curried_type
             (handler_type_list_rec typename)
             (mk_vartype "'y")}

fun rhs_term' typename constname =
   (
     (ratorn (num_of_recursive_pars typename constname)) o
     (add_vars "x")
   )
     (mk_var
        {
         Name =
            "f" ^
            (int_to_string (constructor_position typename constname)),
         Ty =
            handler_type true (==':'y'==) typename constname
        })

fun rhs_term  typename constname =
```

```
   list_mk_comb (
      (rhs_term' typename constname) ,
      (map
         (fn x =>
            PRIMREC_var_term
               typename
               (mk_var{
                  Name = "x" ^ (int_to_string x),
                  Ty = type_expression typename})
         )
         (positions_of_recursive_pars typename constname)
      )
   )

fun PRIMREC_defining_term typename =
   list_mk_conj
      (map
         (fn constname =>
            mk_eq{
               lhs =
                  PRIMREC_var_term
                     typename
                     (const_term typename constname),
               rhs = rhs_term typename constname} )
      (type_constructors typename) )

fun PRIMREC_const typename =
   mk_const{
       Name = "PRIMREC_" ^ typename,
       Ty =
          curried_type
             (handler_type_list_rec typename)
             (mk_vartype "'y")}

fun PRIMREC_const_term typename =
   ((fn {Rator = x, Rand = _} => x) o dest_comb o (add_vars "f"))
      (PRIMREC_const typename)

fun handler_type_list_nonrec typename =
   (map
     (handler_type false (==`:'y`==) typename)
     (type_constructors typename) )

fun CASE_var_term typename cterm =
   ((add_vars "f") o
    (fn x => mk_comb{Rator = x, Rand = cterm}) o
```

```
    mk_var )
     {Name = "CASE_" ^ typename,
      Ty =
         curried_type
            ([type_expression typename] @
               (handler_type_list_nonrec typename))
            (mk_vartype "'y") }

fun handler_term typename constname =
   list_mk_abs(
      (mapnum
         (fn (x,pos) =>
            mk_var{Name = "x" ^ (int_to_string pos), Ty = x})
         (handler_arg_type_list true (==`:'y`==) typename constname )),
     ((add_vars "x") o mk_var)
      {Name =
          "f" ^
          (int_to_string (constructor_position typename constname)),
        Ty = handler_type false (==`:'y`==) typename constname }
     )

fun CASE_defining_term typename =
   let
      val xvar = mk_var {Name = "x", Ty = type_expression typename};
   in
      mk_eq{
         lhs = CASE_var_term typename xvar,
         rhs =
            list_mk_comb
               (PRIMREC_const typename,
                (xvar ::
                 (map
                    (handler_term typename)
                    (type_constructors typename)))
                )
         }
   end

fun CASE_const typename =
   mk_const{
      Name = "CASE_" ^ typename,
      Ty =
         curried_type
            ( (type_expression typename) ::
                 (handler_type_list_nonrec typename) )
            (==`:'y`==)
```

```
      }


(****  Testing the functions above  ****)


fun stringlist_to_string sep [] = "" |
    stringlist_to_string sep [x] = x |
    stringlist_to_string sep (h::(h'::t)) =
       h ^ sep ^ (stringlist_to_string sep (h'::t))

fun demo out f =
   (print "\n";
    print
      ((stringlist_to_string "\n\n")
       (map
         (fn typename =>
            (typename ^ "\t" ^ (out (f typename)) ) )
         (defined_types())) );
    print "\n\n\n")

fun demo' out f =
   (print "\n";
    print
      ((stringlist_to_string "\n\n")
       (map
         (fn typename =>
            (stringlist_to_string "\n")
            (map
               (fn constname =>
                  (typename ^ "\t" ^ constname ^ "\t" ^
                   (out (f typename constname)) ) )
               (type_constructors typename)
             ) )
         (defined_types())) );
    print "\n\n\n")

val tes = term_to_string

val tys = type_to_string

val tyls = fn x => (stringlist_to_string "   " (map tys x))
```

```
(**** dump_pml : print the currently defined pml-objects ****)


fun insert_to_class (a,b) [] = [(b,[a])] |
    insert_to_class (a,b) ((c,d)::t) =
       if (b = c) then
          ((c,(a::d))::t)
       else
          ( (c,d) :: (insert_to_class (a,b) t) )

fun classify [] = [] |
    classify (h::t) = insert_to_class h (classify t)

fun left_adjust n s =
  let
     val bs = (n - (length(explode s)))
     val blanks = if (bs < 0) then 0 else bs
  in
      s ^ (implode (multi blanks " "))
  end

fun sum_of_list [] = 0 |
    sum_of_list (h::t) = h + (sum_of_list t)

fun split_stringlist' _ [] y = y |
    split_stringlist' n (h::t) [] = split_stringlist' n t [[h]] |
    split_stringlist' n (h::t) (hy::ty) =
       let
          val chars =
             sum_of_list (map (fn x => (length (explode x)) + 2) hy)
          val chars' = length (explode h)
       in
          if ((chars + chars') <= n) then
             split_stringlist' n t ((h::hy)::ty)
          else
             split_stringlist' n t ([h]::(hy::ty))
       end

fun split_stringlist n l = split_stringlist' n l []

fun stringlist_to_formated_string m n l =
   stringlist_to_string ("\n" ^ (implode (multi m " ")))
   (map
      (fn x => stringlist_to_string ", " x)
```

```
      (split_stringlist n l))

fun type_declarations_to_formated_string () =
     "\n\n\n\nTYPE-DECLARATIONS\n\n" ^
     (stringlist_to_string "\n"
       ( map
           (fn (name,theory,def) =>
              name ^ "\t =  " ^ (stringlist_to_string " | "
                 (map
                     (fn (const,types) =>
                        const ^
                        (if (types = []) then
                           ""
                         else
                           " of " ^
                           (stringlist_to_string " # "
                              (map
                                 (fn x =>
                                    if (mem " " (explode x)) then
                                       ("("^x^")")
                                    else
                                       x)
                                 types)
                           )
                        )
                     )
                     def)
              )
           )
           (rev (! pml_types))
       )
     )

fun dump print_type_declarations =
   print (
      "\n\n\n   PML-DEFINITIONS\n" ^
      "\n\n\n\nTYPES\n\n" ^
      (stringlist_to_string "\n"
          ( ( (map (fn (a,b) =>
                 (left_adjust 20 (a ^ " : ")) ^
                 (stringlist_to_formated_string 20 50 b))) o
              classify o
              (map (fn (a,b,c) => (a,b))) )
           (! pml_types))
       )   ^
      ( if print_type_declarations then
```

```
           type_declarations_to_formated_string ()
         else
           "" ) ^
       "\n\n\n\nFUNCTIONS\n\n" ^
       (stringlist_to_string "\n"
           ( ( (map (fn (a,b) =>
                  (left_adjust 20 (a ^ " : ")) ^
                  (stringlist_to_formated_string 20 50 b))) o
              classify )
           (! pml_functions))
        )   ^
       "\n\n\n"
     )




(**** Converting PML--programs to HOL ****)


local

fun cut_list_at sep [] = [[]] |
    cut_list_at sep (hx::tx) =
       let val y = cut_list_at sep tx in
          if (hx = sep) then
             []::y
          else
             (hx :: (hd y)) :: (tl y)
       end

(* string_to_defstringlist : splits a pml-text in its definitions *)
local

   fun && (a,b) = if a then b else false
   infix &&

   datatype char_class = alphanum | special  | separator

   fun get_char_class c =
       if (mem c [" ","\n"]) then
          separator
       else
          if (mem c ["(",")","[","]",",",":","*","#",";","|","\""]) then
             special
```

```
       else
          alphanum

local
   fun cut_w [] wl = [implode (rev wl)] |
       cut_w (h::t) wl =
          case (get_char_class h) of
             separator  =>
                if (wl = []) then
                   cut_w t []
                else
                   (implode (rev wl)) :: (cut_w t [])   |
             special  =>
                if (wl = []) then
                   h::(cut_w t [])
                else
                   ((implode (rev wl)) :: (h :: (cut_w t [])))   |
             alphanum =>
                (cut_w t (h::(wl)))
in
   fun cut_words s = cut_w (explode s) []
end

local
   fun delete_comments' [] _ = [] |
       delete_comments' ("("::("*"::t)) false =
          delete_comments' t true |
       delete_comments' ("*"::(")"::t)) true =
          delete_comments' t false |
       delete_comments' (h::t) true =
          delete_comments' t true |
       delete_comments' (h::t) false =
          h :: (delete_comments' t false)
in
  fun delete_comments t = delete_comments' t false
end

fun combine_letval [] = [] |
    combine_letval ("let" :: "val" :: t) =
       "letval" :: (combine_letval t) |
    combine_letval (h::t) = h :: (combine_letval t)

fun cut_defs sl =
  (
   (map (fn x=> (hd x,tl x))) o
   (filter (fn x => (x <> []) && (x <> [""]))) o
```

```
      (cut_list_at ";")
     )
     sl

in

   fun string_to_defstringlist x =
      (cut_defs o combine_letval o delete_comments o cut_words) x

end

(* handle_datatype_definition
   adds a datatype-definition to the current theory *)

datatype 'a option = none | any of 'a

fun split_args [] _ res = res |
    split_args ("#"::t) 0 res = split_args t 0 ([]::res) |
    split_args ("("::t) n res =
       split_args t (n+1) (("(":: (hd res))::(tl res)) |
    split_args (")"::t) n res =
       split_args t (n-1) ((")":: (hd res))::(tl res)) |
    split_args (x::t) n res =
       split_args t n ((x:: (hd res))::(tl res))

fun handle_datatype_definition sl =
   let
      val sl2 = (filter (fn x => x <> "\"")) sl
      val tyname = hd sl2
      val def_string = stringlist_to_string " " sl2
      val sl3 = ((cut_list_at "|") o tl o tl) sl2
      val sl4 =
         map
            (fn x => (hd x, if (tl x = []) then [] else (tl (tl x))))
            sl3
      val sl5 = map (fn (x,y) => (x,split_args y 0 [[]])) sl4
      val sl6 =
         map
            (fn (x,y) =>
               (x,rev (map ((stringlist_to_string " ") o rev) y)))
            sl5
      val def = map (fn (x,y) => (x,filter (fn x => x <> "") y)) sl6
   in
        (
         ( (save_thm(tyname,(theorem "adaptions" (tyname ^ "_DEF"))))
           handle _ =>
```

```
              define_type {
                 name = tyname,
                 type_spec = [QUOTE def_string],
                 fixities = map (fn _ => Prefix) def
               }
         );
         add_type (tyname,def);
         new_recursive_definition{
            def = PRIMREC_defining_term tyname,
            fixity = Prefix,
            name =  "PRIMREC_" ^ tyname,
            rec_axiom = theorem (current_theory()) tyname };
         new_definition ("CASE_" ^ tyname,CASE_defining_term tyname);
         any (tyname,none)
        )
         handle
            HOL_ERR{
               message = a,
               origin_function = b,
               origin_structure = c
             }
            =>
            if (a = "\"" ^ tyname ^
                    "_ISO_DEF\" already an axiom or definition" ) then
               none
            else
               (remove_type tyname;
                any (tyname,any(a,b,c,def_string)))
   end

(* handle_function_definition :
   adds a pml-style function defintion to the current theory *)
local

   fun substitute' (x,y) [] = [] |
       substitute' (x,y) (h::t) =
          (if (h=x) then y else h) :: (substitute' (x,y) t);

   fun substitute [] l = l |
       substitute (h::t) l = substitute t (substitute' h l);

   fun lhsrhs [] = ([],[]) |
       lhsrhs ("="::t) = ([],t) |
       lhsrhs (h::t) = let val (l,r) = lhsrhs t in (h::l,r) end;

   val sublist =      [   (   "fn",      "\\"      ) ,
```

```
                          (   "=>",      "."       ),
                          (   "letval",  "(let"    ),
                          (   "end",     ")"       ),
                          (   "*",       "#"       )  ]

   fun mk _ [] = [] |
       mk m (h::t) =
          case h of
             "(" =>  h :: (mk (false::m) t) |
             "[" =>  h :: (mk (true::m) t)  |
             "]" =>  h :: (mk ((tl m) handle Tl => []) t)  |
             ")" =>  h :: (mk ((tl m) handle Tl => []) t)  |
             "," =>  (if ((hd m) handle Tl => false) then ";" else ",")
                     :: (mk m t)  |
             _   =>  h :: (mk m t)

   val mk_hol_lists = mk [];

in

   fun handle_function_definition sl =
     (let
         val (lhs,rhs) = lhsrhs sl
         val rhs2 = mk_hol_lists (substitute sublist rhs)
         val lhs2 = substitute sublist lhs
         val s = stringlist_to_string " " (lhs2 @ ["="] @ rhs2)
         val name = hd lhs
      in
         let val t = (-- [QUOTE s] --) in
            (
             new_definition(name,t);
             add_function name;
             any (name,none)
            )
         end
         handle HOL_ERR{
            message = a,
            origin_function = b,
            origin_structure = c}
         =>
           if ( (a,b,c) =
                ("attempt to redefine constant","check_lhs","Const_def"))
           then
              none
           else
              any (name, any (a,b,c,s))
```

```
      end)

end (* of function handle_function_definition *)


local

   fun handle_definition (s,sl) =
      case s of
         "primitive_datatype" =>  handle_datatype_definition sl |
         _                    =>  handle_function_definition sl

in

   fun handle_definitions defs =
     let val output_HOL_ERR_old = ! output_HOL_ERR in
        (
         output_HOL_ERR :=
           (fn {message = _:string,
                origin_function = _:
                string,origin_structure = _:string}
            => ());
         let
            val results =
                ( flatten o
                  (map (fn none => [] | (any x) => [x])) o
                  (map (fn x => (handle_definition x)
                                  handle _ =>
                    any("** CAN'T FIND POSITION OF ERROR **",
                       any("Can't convert definition to HOL-Syntax",
                           "","","",""))
                  ) ) )
                defs
            val new_definitions =
               map fst (filter (fn (_,err) => (err = none)) results)
            val errors =
               (
                flatten o
                (map (fn (_,none) => [] | (name,any x) => [(name,x)]))
               )
                results
            val first_error = case errors of
               [] => none |
               (h :: _) => (any h)
         in
            (output_HOL_ERR := output_HOL_ERR_old;
```

```
              (new_definitions, first_error) )
          end
         )
     end

end

fun filename_to_string fname =
   let val f = open_in fname in
   let val s = input(f,100000) in
      (close_in f; s)
   end end

in

fun extend_theory_by_pml_string s =
    let val (new_definitions,first_error) =
       handle_definitions (string_to_defstringlist s)
    in
       print
          (
            "\n\nNEW DEFINITIONS :  \n" ^
            (stringlist_to_string ", " new_definitions) ^
            "\n\n\n\n" ^
            (case first_error of
              none => "CONVERSION COMPLETED." |
              (any (name, (a,b,c,d))) =>
                  "ERROR IN '" ^ name ^ "' .\n\n" ^
                  "HOL-DEFINITION :\n" ^ d ^ "\n\n" ^
                  "HOL-ERROR :\n" ^
                  (stringlist_to_string "\n" [a,b,c])
             ) ^
            "\n\n\n"
          )
    end

fun extend_theory_by_pml_file filename =
  (
    print
       ("\n\n\n " ^ filename ^ " ---> " ^ (current_theory()) ^ "\n\n");
    (extend_theory_by_pml_string (filename_to_string filename))
  )

end
```

```
fun mk_theory_while () =
   (

   new_theory_kill_old "while";

   extend_theory_by_pml_string
      "primitive_datatype \" partial = Defined of 'a | Undefined\";";

   new_definition
      ( "iota",
        (--‘iota (f:'a -> bool) =
            COND (?!x. f x) (Defined (@x.f x)) Undefined ‘--) );

   new_definition
      ( "terminates",
        (--‘terminates (f,n) =
            ( (f n) /\ (!m. (n > m) ==> (~(f m))) ) ‘--) );

   new_definition
     ( "mu",
       (--‘mu f = (iota (\m.terminates(f,m)))‘--)  );

   new_definition
      ( "power",
        (--‘power f n (x:'a) =
           (PRIMREC_num
               n
               (Defined x)
               (\a b. CASE_partial b f Undefined))‘--)
      );

   extend_theory_by_pml_string
        (
      " fun WHILE g f (x : 'a) =                                        " ^
      "    let val steps =                                              " ^
      "       (mu (fn n => (CASE_partial (power f n (x:'a)) g F)) )     " ^
      "    in                                                           " ^
      "        CASE_partial steps (fn n => power f n x) Undefined       " ^
      "    end;                                                         "
        );

   extend_theory_by_pml_string
      "fun PARTIALIZE (f:'a -> 'b) x = Defined (f x);";

   extend_theory_by_pml_string
      "fun PAPPLY (f:'a -> 'b partial) x = CASE_partial x f Undefined;";
```

```
  close_theory()

  )

  fun mk_theory_standard_types() =
  (
  new_theory_kill_old "standard_types";
  extend_theory_by_pml_string
     (
     "primitive_datatype \"  bool   =  T | F                    \";" ^
     "primitive_datatype \"  prod   =  Comma of 'a # 'b         \";" ^
     "primitive_datatype \"  list   =  Nil | Cons of 'a # list  \";" ^
     "primitive_datatype \"  num    =  Zero | Suc of num        \";"
      );
  close_theory()
  )

  fun init() =
     (
      mk_theory_adaptions();
      mk_theory_standard_types();
      mk_theory_while()
     )

end; (* of structure Pml *)
```

Anhang C

PML–Simulator

Die nachfolgenden Funktionen sind für *Standard ML of New Jersey Version 0.93* geschrieben. Durch diese Funktionen wird der SML–Interpreter an PML angepaßt und es können dann Programme mit PML–Syntax eingeladen und ausgeführt werden. Mit diesem PML–Simulator können sowohl Programme in PML–Syntax als auch Programme in der allgemeineren SML–Syntax ausgeführt werden. Das Einladen und Ausführen der PML–Programme geschieht genauso wie bei normalen SML–Programmen. Es wird nicht überprüft, ob ein Programm tatsächlich PML–Syntax hat, oder, ob es in der allgemeineren SML–Syntax geschrieben wurde.

Es sei noch auf eine Eigenheit dieses PML–Interpreters hingewiesen: Die Argumente mehrstelliger Konstruktoren werden (entgegen der PML–Syntax) getupelt dargestellt. Beispielsweise wird die PML–Datentypdeklaration

```
primitive_datatype "tripel = trip of 'a # 'b # 'c ";
```

in die nachfolgende SML–Notation übersetzt:

```
datatype tripel = trip' of 'a * 'b * 'c;
fun trip (a:'a) (b:'b) (c:'c) = trip'(a,b,c);
```

Die SML–Datentypdeklaration selbst entspricht noch nicht den Anforderungen der PML–Semantik, da der Typ des Konstruktors

$\mathsf{trip'}_{(\alpha*\beta*\gamma)\to(\alpha,\beta,\gamma)tripel}$

getupelt ist. Durch die Semantik von PML wird gefordert, daß der Konstruktor

$\mathsf{trip}_{\alpha\to\beta\to\gamma\to(\alpha,\beta,\gamma)tripel}$

gecurriet ist. Aus diesem Grunde wird der Konstruktor trip nicht direkt über die SML–Datentypdeklaration eingeführt, sondern es wird in der Datentypdeklaration zunächst der Konstruktor trip' eingeführt und und der PML–Konstruktor trip wird dann als eine von trip' abgeleitete SML–Funktion definiert.

Für die Syntax der PML–Programme hat dies keine Bedeutung. Es ist lediglich zu beachten, daß der Interpreter Konstruktoren, die zwei oder mehr Argumente haben, getupelt statt gecurriet darstellt. Statt beispielsweise (trip 4 1 3) als Ergebnis einer Evaluierung anzugeben, gibt der Interpreter trip'$(4,1,3)$ aus.

```
(**** Adapting the predefined types ****)



(* bool *)

val F = false;
val T = true;

fun PRIMREC_bool true f1 f2 = f1 |
    PRIMREC_bool false f1 f2 = f2;

val CASE_bool = PRIMREC_bool;


(* prod *)

fun Comma x1 x2 = (x1,x2);

fun PRIMREC_prod (a,b) f1 = f1(a,b);

fun CASE_prod (a,b) f1 = f1(a,b);


(* list *)

fun Cons a b = a :: b;
val Nil = [];

fun PRIMREC_list [] f1 f2 = f1 |
    PRIMREC_list (x1::x2) f1 f2 = f2 x1 x2 (PRIMREC_list x2 f1 f2);

fun CASE_list [] f1 f2 = f1 |
    CASE_list (x1::x2) f1 f2 = f2 x1 x2;


(* num *)

type num = int;

val (Zero:num) = 0;
fun Suc (x:num) = ((x + 1):num);

fun PRIMREC_num (x:num) =
   if (x < 1) then
      (fn f1 => fn f2 => f1)
```

```
   else
      (fn f1 => fn f2 =>  (f2 (x-1) (PRIMREC_num (x-1) f1 f2)) );

fun PRIMREC_num (0:num) = (fn f1 => fn f2 => f1) |
    PRIMREC_num (x:num) =
       (fn f1 => fn f2 =>  (f2 (x-1) (PRIMREC_num (x-1) f1 f2)) );

fun CASE_num (x:num) f1 f2 =
   if (x < 1) then
      f1
   else
      (f2 (x-1));




(**** primitive_datatype : the datatype-declaration mechanism of PML ****)


local

   fun string_to_file str fname =
      let val f = open_out fname in
         (output (f,str);
         close_out f)
       end;

   fun cut_list_at sep [] = [[]] |
       cut_list_at sep (hx::tx) =
          let val y = cut_list_at sep tx in
             if (hx = sep) then
               []::y
             else
                (hx :: (hd y)) :: (tl y)
          end;

   fun filter p [] = [] |
       filter p (h::t) = if (p h) then (h::(filter p t)) else (filter p t);

   local
      fun nts 0 = "" |
          nts n = (nts (n div 10))^(chr((n mod 10)+48))
   in
      fun int_to_string 0 = "0" |
          int_to_string n =
```

```
          (if (not ((abs n) = n))
             then "~"
             else "")^(nts (abs n))
end;

fun mem i (a::rst) = (i=a) orelse (mem i rst) |
    mem i [] = false;

fun fst(x,y) = x;

fun snd(x,y) = y;

fun kill_doubles [] = [] |
    kill_doubles (h::t) =
      if (mem h t) then (kill_doubles t) else (h::(kill_doubles t));

fun flatten [] = [] |
    flatten (h::t) = h @ (flatten t);

fun use_string s =
   (string_to_file s "dummy_file";
    use "dummy_file";
    System.system "rm dummy_file");

fun stringlist_to_string sep [] = "" |
    stringlist_to_string sep [x] = x |
    stringlist_to_string sep (h::(h'::t)) =
        h ^ sep ^ (stringlist_to_string sep (h'::t));

local
   fun mapnum' len f [] = [] |
       mapnum' len f (h::t) =
         (f (h,len - (length t)))  :: (mapnum' len f t)
in
   fun mapnum f l = mapnum' (length l) f l
end;

fun in_brackets s = "(" ^ s ^ ")";

datatype char_class = alphanum | special  | separator;

fun get_char_class c =
   if (mem c [" ","\n"]) then
       separator
   else
      if (mem c ["(",")","[","]",",",":","*","#",";","|","\""]) then
```

```
          special
       else
          alphanum;

local
   fun cut_w [] wl = [implode (rev wl)] |
       cut_w (h::t) wl =
          case (get_char_class h) of
             separator  =>
                if (wl = []) then
                   cut_w t []
                else
                   (implode (rev wl)) :: (cut_w t [])   |
             special  =>
                if (wl = []) then
                   h::(cut_w t [])
                else
                   ((implode (rev wl)) :: (h :: (cut_w t [])))   |
             alphanum =>
                (cut_w t (h::(wl)))
in
   fun cut_words s = cut_w (explode s) []
end;

fun split_args [] _ res = res |
    split_args ("#"::t) 0 res = split_args t 0 ([]::res) |
    split_args ("("::t) n res =
       split_args t (n+1) (("(":: (hd res))::(tl res)) |
    split_args (")"::t) n res =
       split_args t (n-1) ((")":: (hd res))::(tl res)) |
    split_args (x::t) n res =
       split_args t n ((x:: (hd res))::(tl res));

fun parse_type_definition s =
   let
      val sl = cut_words s ;
      val name = hd sl;
      val sl3 = ((cut_list_at "|") o tl o tl) sl
      val sl4 =
         map (fn x => (hd x, if (tl x = []) then [] else (tl (tl x)))) sl3;
      val sl5 = map (fn (x,y) => (x,split_args y 0 [[]])) sl4;
      val sl6 =
         map
            (fn (x,y) => (x,rev (map ((stringlist_to_string " ") o rev) y)))
            sl5;
      val sl7 = map (fn (x,y) => (x,filter (fn x => x <> "") y)) sl6;
```

```
      val def =
         map
            (fn (x,y) =>
               (x,
                map (implode o (map (fn "#" => "*" | x => x)) o explode) y)
             )
            sl7;
   in
     (name,def)
   end;

fun type_variables def =
   (kill_doubles o (filter (fn x => hd(explode x) = "'")) o flatten)
      (map snd def);

fun typed_name (name,def) =
   let val tyvars = type_variables def in
      if (tyvars = []) then
         name
      else
         "(" ^ (stringlist_to_string "," (type_variables def)) ^ ") " ^
         name
end;

fun constructors def = map fst def;

fun typed_constructors (name,def)  =
   map
      (fn (con,tys) =>
         let val tys' =
            map
               (fn x => if (x = name) then (typed_name (name,def)) else x)
               tys
         in
            case (length tys) of
              0 => con  |
              1 => con ^ " of " ^ (hd tys) |
              _ => con ^ "' of " ^
                   (in_brackets (stringlist_to_string " * " tys'))
         end )
      def;

fun datatype_string (name,def) =
   "datatype " ^ (typed_name (name,def)) ^ " = " ^
   (stringlist_to_string " | " (typed_constructors (name,def)) ^ ";" );
```

```
fun currying_string def =
   (
    (stringlist_to_string "\n") o
    (filter (fn x => x <> ""))
   )
   (map
      (fn (con,tys) =>
         if (length tys <= 1) then
             ""
          else
             let
                val vars =
                   mapnum (fn (_,pos) => "x" ^ (int_to_string pos)) tys
             in
                "fun " ^
                con ^ " " ^ (stringlist_to_string " " vars) ^
                " = " ^
                con ^ "'" ^
                (in_brackets (stringlist_to_string "," vars)) ^ ";"
             end
         )
      def);

fun primrec_string (name,def) =
   let
      val fs = mapnum (fn (_,pos) => "f" ^ (int_to_string pos)) def;
      val fstring = stringlist_to_string " " fs;
      val pr_basic = "PRIMREC_" ^ name
      val equations =
         mapnum
            (fn ((con,tys),pos) =>
               let
                  val varlist =
                     mapnum (fn (_,pos) => "x" ^ (int_to_string pos)) tys
                  val recvarlist =
                    flatten
                       (mapnum
                          (fn (ty,pos) =>
                             if (ty = name) then
                                ["x" ^ (int_to_string pos)]
                             else
                                [])
                          tys)
                 val term =
                    case (length tys) of
                       0 => con |
```

```
                        1 => in_brackets (con ^ " x1") |
                        _ => in_brackets
                                (con ^ "'" ^
                                  (in_brackets
                                     (stringlist_to_string "," varlist)))
                in
                   pr_basic ^ " " ^
                   term ^ " " ^
                   fstring ^ " " ^
                   " = f" ^ (int_to_string pos) ^ " " ^
                   (stringlist_to_string " " varlist) ^ " " ^
                   (stringlist_to_string " "
                      (map
                         (fn x =>
                            in_brackets
                               (pr_basic ^ " " ^ x ^ " " ^ fstring ^ " "))
                         recvarlist) )
                end
               )
            def
   in
      "fun " ^ (stringlist_to_string " |\n    " equations) ^ ";"
   end;

fun case_string (name,def) =
   let
      val fs = mapnum (fn (_,pos) => "f" ^ (int_to_string pos)) def
      val fstring = stringlist_to_string " " fs
      val pr_basic = "CASE_" ^ name
      val equations =
         mapnum
            (fn ((con,tys),pos) =>
               let
                  val varlist =
                     mapnum (fn (_,pos) => "x" ^ (int_to_string pos)) tys
                  val recvarlist =
                    flatten
                       (mapnum
                          (fn (ty,pos) =>
                             if (ty = name) then
                                ["x" ^ (int_to_string pos)]
                             else
                                [] )
                          tys)
                  val term =
                    case (length tys) of
```

```
                        0 => con |
                        1 => in_brackets (con ^ " x1") |
                        _ => in_brackets
                                (con ^ "'" ^
                                  (in_brackets
                                     (stringlist_to_string "," varlist)))
                in
                   pr_basic ^ " " ^
                   term ^ " " ^
                   fstring ^ " " ^
                   " = f" ^ (int_to_string pos) ^ " " ^
                   (stringlist_to_string " " varlist)
                end
               )
             def
      in
         "fun " ^ (stringlist_to_string " |\n    " equations) ^ ";"
      end;

in

   fun primitive_datatype_defstring s =
      let
         val (name,def) = parse_type_definition s
      in
         "\n\n (*   \n    primitiveML-style type definition: \n\n       " ^
         s ^ "\n *)\n\n\n" ^
         "\n\n\n(* Type-Declaration *)\n\n" ^
         (datatype_string (name,def)) ^
         "\n\n\n(* Currying the constructors *)\n\n" ^
         (currying_string def) ^
         "\n\n\n(* Primitive Recursion *)\n\n" ^
         (primrec_string (name,def)) ^
         "\n\n\n(* case *)\n\n" ^
         (case_string (name,def)) ^
         "\n\n"
      end;

   fun primitive_datatype s =
      use_string (primitive_datatype_defstring s);

end;
```

```
(**** Implementation of basics for mu-recursion ****)


primitive_datatype "partial = Defined of 'a | Undefined";

local
  fun WHILE' g f (Defined x) =
        if (g x) then (WHILE' g f (f x)) else (Defined x) |
      WHILE' g f Undefined = Undefined;
in
  fun WHILE g f x = (WHILE' g f (Defined x))
end;

fun PARTIALIZE f x = Defined (f x);

fun PAPPLY f x =
   case x of
      Undefined   => Undefined |
      (Defined y) => (f y);
```

Anhang D

PML–Programme der Beispielschaltungen

D.1 Grundbausteinbibliothek

```
primitive_datatype "btree  =  Bleaf of 'a | Bnode of btree # btree";

primitive_datatype "unit  =  Unit";




(**** polymorphic multiplexer ****)


fun mux (a,(b:'a),c) = CASE_bool a c b;




(**** generic circuits ****)


fun map (f:'a -> 'b) x =
   PRIMREC_list
      x
      Nil
      (fn ah => fn at => fn ra => Cons (f ah) ra );

fun map_btree (f:'a -> 'b) x =
   PRIMREC_btree
      x
      (fn x => Bleaf (f x))
      (fn a => fn b => fn ra => fn rb => Bnode ra rb);

fun row (f:('a * 'b) -> ('c * 'a)) (a,b) =
   PRIMREC_list
       b
       (Nil,a)
       (fn h => fn t => fn (ct,ax) =>
          let val (ch,d) = f(ax,h) in
             ( (Cons ch ct) , d )
          end
       );
```

```
fun bmux (s,(a:'a btree)) =
   PRIMREC_btree a
      (fn x => fn s => x)
      (fn x => fn y => fn rx => fn ry => fn s =>
         CASE_list s
            (rx [])
            (fn h => fn t => mux(h,(rx t),(ry t)))
      )
      s;

fun bdx default ((l:bool list),(x:'a)) =
   PRIMREC_list
      l
      (fn x => Bleaf x)
      (fn h => fn t => fn r =>
         fn x =>
            Bnode (r(mux(h,x,default))) (r(mux(h,default,x))) )
      x;

fun apply_f_on_addressed_bleaf (f:'a -> 'a) ((l:bool list),(bt:'a btree)) =
   PRIMREC_list
      l
      (fn x =>
         CASE_btree
            x
            (fn y => Bleaf (f y))
            (fn a => fn b => Bnode a b)
      )
      (fn h => fn t => fn r =>
         fn x =>
            CASE_btree
               x
               (fn y => Bleaf (f y))
               (fn a => fn b =>
                  Bnode (mux(h,a,r b)) (mux(h,r a,b))
               )
      )
      bt;

fun combine_btrees (f: ('a * 'b) -> 'c) (bt1,bt2) =
   PRIMREC_btree
      bt1
      (fn a =>
         (fn x =>
            PRIMREC_btree
               x
```

```
               (fn b => Bleaf (f(a,b)))
               (fn bt1 => fn bt2 => fn br1 => fn br2 =>  br1)
         )
      )
      (fn at1 => fn at2 => fn ar1 => fn ar2 =>
         (fn x =>
            CASE_btree
               x
               (fn b => ar1 (Bleaf b))
               (fn bt1 => fn bt2 => (Bnode (ar1 bt1) (ar2 bt2))  )
         )
      )
      bt2;




(**** Wiring ****)


fun fst ((a:'a),(b:'b)) = a;

fun snd ((a:'a),(b:'b)) = b;

fun tail x = CASE_list x Nil (fn (xh:'a) => (fn xt => xt));

fun head (x,head_of_nil) =
   CASE_list x head_of_nil (fn (xh:'a) => (fn xt => xh));

fun append ((a:'a list),b) =
   PRIMREC_list
      a
      b
      (fn ah => (fn at => (fn ra => Cons ah ra)));

fun reverse x =
   PRIMREC_list
      x
      Nil
      (fn ah => (fn (at:'a list) => (fn ra => append(ra,Cons ah Nil) )));

fun flat x =
   PRIMREC_list x Nil (fn (h:'a list) => fn t => fn r => append(h,r));

fun flat_btree x =
```

```
   PRIMREC_btree
      x
      (fn (x:'a btree) => x)
      (fn a => fn b => fn ra => fn rb => Bnode ra rb);

fun multi n (x:'a) =
   PRIMREC_num
      n
      []
      (fn m => fn r => (Cons x r));

fun multi_btree n (x:'a) =
   PRIMREC_num
      n
      (Bleaf x)
      (fn n => fn r => Bnode r r);

fun nth default n (l:'a list) =
   PRIMREC_list
      l
      (fn m => default)
      (fn h => fn t => fn r =>
         (fn m => CASE_num m h (fn k => r k))
      )
      n ;

fun lp (x: ('a * 'b) list) =
   PRIMREC_list
      x
      (Nil,Nil)
      (fn (ah,bh) => (fn t => (fn (at,bt) => ((Cons ah at),(Cons bh bt)) )));

fun pl (a: ('a list) , (b : ('b list)) ) =
   PRIMREC_list
      a
      (fn x =>
         (CASE_list x Nil (fn bh => (fn bt => Nil)))
      )
      (fn ah =>  (fn at =>  (fn abt =>
         (fn x =>
            (CASE_list x Nil (fn bh => (fn bt => Cons (ah,bh) (abt bt))))
         )
      )))
      b;

fun btree_to_list x =
```

```
   PRIMREC_btree
      x
      (fn (x : 'a) => [x])
      (fn a => fn b => fn ra => fn rb => (append(ra,rb)));

fun list_to_btree (default:'a) n l =
   map_btree
      (fn n => nth default n l)
      (fst
         (PRIMREC_btree
            (multi_btree n Unit)
            (fn x => (fn start => (Bleaf start, Suc start)))
            (fn a => fn b => fn ra => fn rb =>
               (fn start =>
                  let val (lhs,startr) = ra start in
                  let val (rhs,startrr) = rb startr in
                     (Bnode lhs rhs,startrr)
                  end end
                )
              )
           Zero
      ));
```

D.2 Bibliothek aus abgeleiteten Bausteinen

```
(**** variants of row ****)


fun cas a (f: ('a * 'b) -> 'a) x =
   let val g = (fn ab => (Unit,f ab)) in
   let val (out1,out2) = (row g (a,x)) in
      out2
   end end;

fun ser n f (x:'a) =
   snd (row (fn (x,z:unit) => (Unit,f x)) (x,multi n Unit));




(****  boolean circuits  ****)


fun inv x =
   mux(x,T,F);

fun and2 (a,b) =
   mux(a,F,b);

fun or2 (a,b) =
   mux(a,b,T);

fun eor (a,b) =
   mux(a,b,(inv b));

fun equ (a,b) =
   mux(a,(inv b),b);

fun imp (a,b) =
   mux(b,(inv a),T);   (* b -> a *)


fun nandn x =
   cas F (fn (a,b) => imp(b,a)) x;

fun andn x =
```

```
   inv (nandn x);

fun orn x =
   cas F or2 x;


fun two_level (f:('a list)->'b) (g:('b list)->'c) x =
   g (map f x);

val df =
   two_level andn orn;




(****  arithmetic  ****)


fun fulladder (cin,(a,b)) =
   let val w1  = eor(a,b) in
   let val w2  = and2(a,b) in
   let val sum = eor(w1,cin) in
   let val w3 = and2(w1,cin) in
   let val cout = or2(w2,w3) in
      (sum,cout)
   end end end end end;

fun fulladdern (cin,a,b) =
   let val ab = pl(a,b) in
   let val scout = row fulladder (cin,ab) in
      scout
   end end;

fun add (a,b) =
   fst (fulladdern (F,a,b));

fun add_carry (a,b) =
   snd(fulladdern(F,a,b));


fun increment x =
   row (fn (a,b) => (eor (a,b), and2 (a,b))) (T,x);

fun inc x =
   fst (increment x);
```

```
fun inc_carry x =
   snd (increment x);


fun twocomp x =
   let val (w1,w2) = (increment (map inv x)) in w1 end;


fun equn(a,b) =
   cas
      T
      (fn (l,(an,bn)) => and2(equ(an,bn),l))
      (pl(a,b));


fun lt (a,b) = and2(inv a,b);

fun ltn (a,b) =
   cas
      F
      (fn (l,(an,bn)) => or2(lt(an,bn),and2(equ(an,bn),l)) )
      (pl(a,b));


fun ltequ (a,b) =
   imp(a,b);

fun ltequn (a,b) =
   cas
      T
      (fn (l,(an,bn)) => or2(lt(an,bn),and2(equ(an,bn),l)) )
      (pl(a,b));

fun minn (a,b) =
   mux ((ltn(b,a)),a,b);

fun maxn (a,b) =
   mux ((ltn(a,b)),a,b);

fun inv_first_bit x =
   Cons (inv (head(x,F))) (tail x);

fun ltn_twocomp (a,b) =
   ltn(inv_first_bit a,inv_first_bit b);
```

```
fun ltequn_twocomp (a,b) =
   ltequn(inv_first_bit a,inv_first_bit b);

fun minn_twocomp (a,b) =
   mux ((ltn_twocomp(b,a)),a,b);

fun maxn_twocomp (a,b) =
   mux ((ltn_twocomp(a,b)),a,b);

fun is_zero x =
   inv (cas F or2 x);

fun deleten (reset,x) =
   mux (reset,x,map (fn p => F) x);

fun xcounter (reset,state) =
   deleten (reset,inc state);

fun num_to_boollist m n =
   ser n inc (multi m F);




(**** D-Flipflop, RAM ****)


fun xdff ((d,en), (state:'a)) =
   mux(en,state,d);

fun yram ((address, write_enable, data_in), state) =
   let val data_out = bmux (address,state) in
   let val state_   =
      apply_f_on_addressed_bleaf
         (fn x => mux (write_enable,x,data_in))
         (address,state)
   in
      (data_out, state_)
   end end;
```

D.3 Die Min–Max–Schaltung

```
fun averagen (a,b) =
   let val (sum,carry) = fulladdern(F,a,b) in
   let val l = reverse(tail(reverse(append ([carry],sum)))) in
      Cons (head(l,F)) l
   end end;

fun choice ch (((x:bool list),reset), state) =
   deleten(reset,ch (x,state));

val xmaxn =
   choice maxn_twocomp;

val xminn =
   choice minn_twocomp;

fun yminmax ((input,clear,enable,reset), (min,max,old_output)) =
   let val min_ = xminn ((input,reset), min) in
   let val max_ = xmaxn ((input,reset), max) in
   let val w2 = averagen(min_,max_) in
   let val w3 = mux(reset,w2,input) in
   let val w4 = mux(enable,old_output,w3) in
   let val output = deleten(clear,w4) in
   let val old_output_ = output in
      ((output,min_,max_),(min_,max_,old_output_))
   end end end end end end end;

fun y_to_z ((ycir:'a * 'b -> 'c * 'b),ystate0) inputfn t =
   snd
      (PRIMREC_num
         t
         (ycir (inputfn 0,ystate0))
         (fn n => fn (old_out,ystate) => ycir (inputfn n,ystate) ));

fun zminmax wordlength =
   let val zero = multi wordlength F in
      y_to_z (yminmax,(zero,zero,zero))
   end;
```

D.4 Der Mikroprozessor

```
(**** Abstract description of instruction cycles ****)


primitive_datatype
  "mp_instruction = Ld|Ldc|Ldi|St|Sti|Jmp|Jmpi|Jmpz|Jmpc|Add|Inv|Nop";

primitive_datatype "idbpart = Pc|Ar|Db|Ir|Accu|Auxr|Alu|None";

primitive_datatype "flag = Carry_flag | Zero_flag";

primitive_datatype "readwrite = Read | Write";

primitive_datatype "alu_operation = Plus | Invers | Nop";

primitive_datatype
   "cycle_description = Simply of idbpart#readwrite#alu_operation#idbpart#
                        idbpart | Conditional_jump of flag";


val get_pc       =  Simply  Pc   Read Nop    None None;
val get_ar       =  Simply  Ar   Read Nop    None None;

val put_ar       =  Simply  Ar   Write Nop   Accu Db;

val db_to_ir     =  Simply  None Read Nop    Db   Ir;
val db_to_accu   =  Simply  None Read Nop    Db   Accu;
val db_to_ar     =  Simply  None Read Nop    Db   Ar;
val db_to_auxr   =  Simply  None Read Nop    Db   Auxr;
val db_to_pc     =  Simply  None Read Nop    Db   Pc;
val sum_to_accu  =  Simply  None Read Plus   Alu  Accu;
val inv_to_accu  =  Simply  None Read Invers Alu  Accu;

val wait         =  Simply  None Read Nop    None None;

val jump_if_carry  = Conditional_jump Carry_flag;
val jump_if_zero   = Conditional_jump Zero_flag;


fun get_cycle_description x =
   append
      ([get_pc, db_to_ir, get_pc],
       PRIMREC_mp_instruction
```

```
                x
      (* Ld *)    [db_to_ar, get_ar, db_to_accu]
      (* Ldc *)   [db_to_accu]
      (* Ldi *)   [db_to_ar, get_ar, db_to_ar, get_ar, db_to_accu]
      (* St *)    [db_to_ar, put_ar]
      (* Sti *)   [db_to_ar, get_ar, db_to_ar, put_ar]
      (* Jmp *)   [db_to_pc]
      (* Jmpi *)  [db_to_ar, get_ar, db_to_pc]
      (* Jmpz *)  [jump_if_zero]
      (* Jmpc *)  [jump_if_carry]
      (* Add *)   [db_to_auxr, sum_to_accu]
      (* Inv *)   [inv_to_accu]
      (* Nop *)   [wait]
    );




(**** Coding ****)


fun address_of_idbpart x =
   num_to_boollist 3 (PRIMREC_idbpart x 0 1 2 3 4 5 6 7);

fun code_of_alu_operation x = PRIMREC_alu_operation x (F,T) (T,T) (F,F);

fun code_of_readwrite x = PRIMREC_readwrite x false true;

fun code_of_cycle_ absel rw aluop wr rd =
   (
      PRIMREC_idbpart absel T F F F F F F F ,
      code_of_readwrite rw,
      code_of_alu_operation aluop,
      address_of_idbpart wr,
      address_of_idbpart rd
   );




(* CPU-Internal ROMs *)
```

```
fun code_of_cycle x =
   PRIMREC_cycle_description
      x
      (fn absel => fn rw => fn aluop => fn wr => fn rd =>
         let val a = code_of_cycle_ absel rw aluop wr rd in
            Bnode (Bnode (Bleaf a) (Bleaf a)) (Bnode (Bleaf a) (Bleaf a))
         end
      )
      (fn flag =>
         let val a = code_of_cycle_ None Read Nop None None in
         let val b = code_of_cycle_ None Read Nop Db Pc in
            PRIMREC_flag
               flag
               (Bnode (Bnode (Bleaf a) (Bleaf a)) (Bnode (Bleaf b) (Bleaf b)))
               (Bnode (Bnode (Bleaf a) (Bleaf b)) (Bnode (Bleaf a) (Bleaf b)))
         end end
      ) ;

fun code_of_instruction x =
   flat_btree
      (list_to_btree
         (code_of_cycle wait)
         3
         (map
             code_of_cycle
             (get_cycle_description x) )
       );

val mp_instruction_btree =
   list_to_btree Nop 4 [Ld,Ldc,Ldi,St,Sti,Jmp,Jmpi,Jmpz,Jmpc,Add,INV];

fun length x =
   PRIMREC_list x Zero (fn (h:'a) => fn t => fn r => (Suc r));

val instruction_code =
   flat_btree (map_btree code_of_instruction mp_instruction_btree);

val instruction_cycles_code =
   map_btree
      (fn y =>
         num_to_boollist
            3
            (length(tail(get_cycle_description y))) )
      mp_instruction_btree;

fun instruction_rom x =
```

```
   bmux (x,instruction_code);

fun instruction_cycles_rom x =
   bmux (x,instruction_cycles_code);




(**** ALU - Arithmetic And Logic Unit ****)


fun yalu ((accu,auxr,operation,en),carry) =
   let val (sum,car) = fulladdern (F,accu,auxr) in
   let val carry_ = xdff ( (car,en),carry ) in
   let val inv = twocomp accu in
   let val result = mux (operation,sum,inv) in
      (result, carry_)
   end end end end;




(**** Operation Unit ****)


fun yoperation_unit
    ( ((absel,(rw:bool),(alu_op,alu_en),idbw,idbr), db_in),
      ((ir,carry,zero),(pc,ar,accu,auxr)) )
 =
   let val pcinc = absel in
   let val ab_out = mux(absel,ar,pc) in
   let val (alu_result,carry_) = yalu ((accu,auxr,alu_op,alu_en),carry)  in
   let val idb =
      bmux
         (idbw,list_to_btree db_in 3 [pc,ar,db_in,ir,accu,auxr,alu_result])
    in
   let val ens =
      (fn x => bmux (address_of_idbpart x,bdx F (idbr,T))) in
   let val pc_    = mux(ens Pc,mux(pcinc,pc,inc pc),idb) in
   let val ar_    = xdff ( (idb,ens Ar), ar ) in
   let val db_out = idb in
   let val ir_    = xdff ( (idb,ens Ir), ir ) in
   let val new_accu = mux (ens Accu,accu,idb) in
   let val accu_  = new_accu in
```

```
   let val auxr_  = xdff ( (idb,ens Auxr), auxr ) in
   let val zero_  = is_zero new_accu in
      ( (ab_out,db_out,rw), ((ir_,carry_,zero_),(pc_,ar_,accu_,auxr_)) )
   end end end end end end end end end end end end;




(**** Control Unit ****)


fun ycontrol_unit ((ir,carry,zero),state) =
   let val ir4 = [nth F 0 ir, nth F 1 ir, nth F 2 ir, nth F 3 ir] in
   let val out =
      instruction_rom
         (append (ir4,append(state,Cons carry (Cons zero Nil))))
    in
   let val reset = equn (state,instruction_cycles_rom ir) in
   let val state_ = xcounter(reset,state) in
      (out,state_)
   end end end end;




(**** MP - The Microprocessor ****)


fun ymp  ( db_in, ( status, oper_state, contr_state) ) =
   let val (control,contr_state_) = ycontrol_unit (status,contr_state) in
   let val (out,(status_,oper_state_)) =
               yoperation_unit ((control,db_in),(status,oper_state)) in
      (out, ( status_, oper_state_, contr_state_))
   end end;




(**** MC - A Simple Microcomputer consisting of a MP and a RAM ****)


fun ymc (u:unit,(ram_state,mp_state,db)) =
   let val ((ab,mp_db,rw),mp_state_) = ymp (db,mp_state) in
   let val (ram_db,ram_state_) = yram ((ab,rw,mp_db),ram_state) in
```

```
let val db_ = ram_db in
   ( (ram_state,mp_state,db), (ram_state_,mp_state_,db_))
end end end;
```

D.5 Der Assembler

```
primitive_datatype
   "assembler_operand_type = Num | Lab";

primitive_datatype
   "assembler_line = Address of num | Value of num | Label of num |
                     Instr of mp_instruction # assembler_operand_type # num";

fun instruction_to_boollist wordlength mp_instr =
   append (
      num_to_boollist
         4
         (CASE_mp_instruction mp_instr 0 1 2 3 4 5 6 7 8 9 10 11 ),
      tail (tail (tail (tail (multi wordlength F))))
      );

fun first_pass_ l =
   PRIMREC_list
      l
      ((Nil,Nil),0)
      (fn h => fn t => fn ((lines,labels),ad) =>
         CASE_assembler_line
             h
             (fn x => ((lines,labels),x))    (* Address *)
             (fn x => ((Cons (ad,Value x) lines,labels),Suc ad))   (* Value *)
             (fn x => ((lines,Cons (x,ad) labels), ad))   (* Label *)
             (fn x => fn y => fn z =>
                ((Cons (ad,Instr x y z) lines,labels),Suc (Suc ad)))
                   (* Instr *)
      );

fun first_pass x = fst (first_pass_ (reverse x));

fun eq_num (a,b) =
   PRIMREC_num
      a
      (fn x => (CASE_num x T (fn pa => F)))
      (fn pa => fn ra =>  (fn x => CASE_num x F ra ))
      b;

fun lt_num (a,b) =
   PRIMREC_num
      a
```

```
      (fn x => (CASE_num x F (fn x => T)))
      (fn pa => fn ra => (fn x => CASE_num x F ra))
      b;

fun max_of_num_list l =
   PRIMREC_list
      l
      0
      (fn h => fn t => fn r =>  CASE_bool (lt_num(r,h)) h r);

fun lookup (default:'a) x l =
   PRIMREC_list
      l
      default
      (fn (a,b) => fn t => fn r =>
         CASE_bool (eq_num (a,x)) b r
      );

fun fun_to_list n f =
   PRIMREC_num
     n
     []
     (fn m => fn r => append (r, [f m]));

fun address_lines_to_addressed_data wordlength label_list addressed_lines =
   flat
      (map
         (fn (ad,line) =>
            CASE_assembler_line
               line
               (fn x => [])
               (fn x => [(ad,num_to_boollist wordlength x)])
               (fn x => [])
               (fn mp_instr => fn optype => fn x =>
                  [
                   (ad,instruction_to_boollist wordlength mp_instr),
                   (Suc ad,
                    num_to_boollist wordlength
                      (CASE_assembler_operand_type
                         optype x (lookup 0 x label_list) ) )
                  ]
               )
         )
         addressed_lines);

fun assemble wordlength prog =
```

```
let val (addressed_lines,label_list) = first_pass prog in
let val addressed_data = address_lines_to_addressed_data
                            wordlength label_list addressed_lines in
let val max_address = max_of_num_list (map fst addressed_data) in
let val default = instruction_to_boollist wordlength Nop in
   list_to_btree
      default
      wordlength
      (fun_to_list
         (Suc max_address)
         (fn ad => lookup default ad addressed_data))
end end end end;
```

Literaturverzeichnis

[BGGH92] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experiences with Embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R. Boute, editors, *Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.

[BGHT91] R. Boulton, M. Gordon, J. Herbert, and J. van Tassel. The HOL verification of ELLA designs. In *International Workshop on Formal Methods in VLSI Design*, January 1991.

[BoPS92] D. Borrione, L. Pierre, and A. Salem. Formal verification of VHDL descriptions in Boyer-Moore: First results. In J. Mermet, editor, *VDHL for Simulation, Synthesis and Formal Proofs of Hardware*, pages 227–243. Kluwer Academic Press, 1992.

[BrHY92] Bishop C. Brock, Warren A. Hunt, Jr., and William D. Young. Introduction to a formally defined hardware description language. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Theorem Provers in Circuit Design*, pages 3–35. North Holland, 1992.

[CaGM86] A. Camilleri, M.J.C. Gordon, and Th. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66. North-Holland, 1986.

[CaMe92] J. Camilleri and T.F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, 1992.

[Cami88] J. Camilleri. Executing behavioural definitions in higher order logic. Technical Report 140, University of Cambridge Computer Laboratory, 1988.

[Cami91] J. Camilleri. Symbolic compilation and execution of programs by proof: a case study in HOL. Technical Report 240, University of Cambridge Computer Laboratory, 1991.

[DeHa93] Alan Dent and Keith Hanna. Reasoning about array structures using a depently typed logic. In David Agnew, Luc Claesen, and Raul Camposano, edi-

tors, *Conference Proceedings of the IFIP Conference on Hardware Description Languages and their Applications*, pages 195–212. OCRI Publications, 1993.

[FlLi90] T. Flik and H. Liebig. *Mikroprozessortechnik*. Springer–Verlag, 1990.

[Gunt92] E. Gunter. Why we can't have SML-style datatype declarations in HOL. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions*, pages 561–568, Leuven, Belgium, 1992. North-Holland.

[HOL93] University of Cambridge, DSTO, SRI Internatioal. *Description of the HOL-System*, 1993.

[Hunt86] W.A. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89–132. North-Holland, 1986.

[Jone87] S.L.P. Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[JoSh91] G. Jones and Mary Sheeran. Collecting butterflies. Technical report, Oxford University Computing Laboratory, 1991.

[KfMA86] A.J. Kfoury, R.N. Moll, and M.A. Arbib. *A Programming Approach to Computability*. Springer, 1986.

[KuSK93] R. Kumar, K. Schneider, and Th. Kropf. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Journal of Formal Methods in System Design*, 2(2):165–223, 1993.

[Melh88] T. F. Melham. Automating recursive type definitions in higher order logic. Technical Report 146, University of Cambridge Computer Laboratory, September 1988.

[OLLA92] John O'Leary, Mark Linderman, Miriam Leeser, and Mark Aagaard. HML: A hardware description language based on SML. Technical Report EE–CEG–92–7, School of Electrical Engineering, Cornell University, Ithaca, NY 14853, 1992.

[Scho85] R. Scholz. *Einführung in die Mikrocomputertechnik*. Teubner Studienskripten, Stuttgart, 1985.

[Shee88] M. Sheeran. Retiming and slowdown in Ruby. In G.J. Milne, editor, *The fusion of Hardware Design and Verification*, pages 289–308, Glasgow, 1988. North Holland.

[ShRa93] R. Sharp and O. Rasmussen. Rewriting with constraints in T-Ruby. In *CHARME93*, number 683 in Lecture Notes in Computer Science, pages 226–241, Arles,France, May 1993. Springer Verlag.