Implementierung von first–level NoName auf einer Runtime–Stack Maschine

Diplomarbeit an der Universität Ulm

Fakultät für Informatik

vorgelegt von:

Frank Houdek

1. Gutachter: Prof. Dr. Heiko Vogler, TU Dresden
2. Gutachter: Prof. Dr. Helmuth Partsch, Uni Ulm

1995

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Ansatz der funktionalen Programmierung bietet eine Reihe interessanter Vorteile im Vergleich zum Ansatz der imperativen Programmierung. Viele komplexe Problemstellungen lassen sich mit funktionalen Programmiersprachen kurz und prägnant formulieren. Dies liegt daran, daß man mit funktionalen Programmiersprachen beschreibt, *was* die Aufgabe ist. Dem gegenüber ist man bei imperativen Programmiersprachen gezwungen anzugeben, *wie* die Aufgabe gelöst werden soll.

Funktionale Programmiersprachen orientieren sich stark an der Mathematik, so daß sie in der Regel auch eine klare, einfache und wohldefinierte Semantik besitzen. Diese Eigenschaft bildet auch die Grundlage für Korrektheitsbeweise und Programmtransformationen.

Bekannte funktionale Programmiersprachen sind beispielsweise Miranda[1] [Tur85], Gofer [Jon91], Haskell [HW88] und ML [Mil84]. Ein weiterer Vertreter dieser Sprachfamilie ist die funktionale Programmiersprache NoName, die 1994 an der Universität Ulm unter Leitung von Prof. Dr. Heiko Vogler entwickelt wurde ([EGH$^+$94]).

Um nun Programme, die eigentlich nur Zeichenreihen sind, ausführen zu können, benötigt man eine Implementation der zugrundeliegenden Programmiersprache. Eine Implementation besteht häufig aus einem Übersetzer und einer Maschine. Der Übersetzer erzeugt aus dem gegebenen Programm ein Maschinenprogramm, welches von der Maschine abgearbeitet werden kann.

In dieser Diplomarbeit stellen wir eine Implementation für eine Teilsprache von NoName, die wir NoName* nennen, vor. Dazu beschreiben wir sowohl einen Übersetzer als auch eine Maschine, die wir NoName–Maschine nennen.

1.1 Die Sprache NoName*

Die Sprache NoName hebt sich durch einige Eigenschaften von der Vielzahl der bereits exisitierenden funktionalen Programmiersprachen ab. Diese Eigenschaften, die sich in dieser Zusammenstellung bisher bei keiner anderen Programmiersprache finden, wollen wir im folgenden aufführen und kurz erläutern.

[1]Das Miranda–System ist ein Warenzeichen der Firma Software Research Ltd., Großbritannien.

- Die Sprache NoName verfolgt ein strenges Typkonzept und hat eine Syntax, die sich stark an der mathematischen Notation von Funktionen orientiert. Jede Funktion besitzt eine festgelegte Stelligkeit, die bei jeder Funktionsanwendung beachtet werden muß. Insbesondere ist es nicht möglich, eine Funktion mit weniger Parametern aufzurufen, als in der Funktionsdeklaration angegeben.
- Für jede Funktion kann explizit angegeben werden, nach welcher Strategie die Auswertung der Funktion erfolgen soll. Dabei stehen die Auswertungsstrategien Call–by–Value und Call–by–Name zur Auswahl.
- Die Erstellung von tree transducern wird durch syntaktische Konstrukte unterstützt. Unter einem tree transducer versteht man ein spezielles Schema, mit dem Baum zu Baum–Übersetzungen beschrieben werden können. In NoName wird die Beschreibung von top–down tree transducern [Rou70, Tha70], macro tree transducern [Eng80, CF82, EV85], attributed tree transducern [Fül81], modular tree transducern [EV91] und macro attributed tree transducern [KV94] durch eigene syntaktische Konstrukte unterstützt.

Eine genaue Beschreibung der Sprache NoName findet sich in [EGH+94]. Dieser Arbeit liegt die Version 1.1 von NoName zugrunde.

Vorstellung der Sprache NoName* an einem Beispiel

Die Einschränkung von NoName* gegenüber NoName besteht nun darin, daß in NoName*–Programmen keine Funktionen höherer Ordnung verwendet werden dürfen.

Im folgenden werden wir die Sprache NoName* anhand des Beispielprogramms in Abbildung 1.1 ein wenig genauer vorstellen. Am Anfang eines jeden NoName*–Programms P steht eine Anfrage, also ein Ausdruck *anfr*, der mit Hilfe der definierten Funktionen ausgewertet werden soll. In unserem Beispielprogramm ist `f_down(gen_baum(3,2,TRUE))` die Anfrage.

Die Tatsache, daß eine Anfrage Bestandteil eines Programms ist, erscheint auf den ersten Blick ein wenig verwirrend, da üblicherweise ein Programm für eine Vielzahl von Anfragen konzipiert ist. Diese Vorgehensweise hat jedoch den Vorteil, daß das Ergebnis der Übersetzung, welches auch Maschinenbefehle für *anfr* enthält, direkt ausgeführt werden kann und keine weiteren Eingaben erfordert.

Im Anschluß an die Anfrage folgt das eigentliche Programm, in dem sämtliche benötigten Datentypen und Funktionen spezifiziert werden. Die unterschiedlichen Spezifikationen werden durch eigene Schlüsselwörter gekennzeichnet.

Zwischen den Schlüsselwörtern `ALG_DEF` und `END_ALG_DEF` wird im obigen Beispiel der algebraische Datentyp `Baum` mit den Konstruktoren `Knoten` und `Blatt` definiert. Die Funktion `gen_baum` ist mit Hilfe einer Fallunterscheidung via IF definiert. Dabei können wir auch sehen, wie die explizite Angabe einer Auswertungsstrategie — hier Call–by–Name — vorgenommen wird.

Die Funktion `get_zahl` ist mittels Pattern Matching definiert. Erfolgt die Definition einer Funktion mittels Pattern Matching, so ist darauf zu achten, daß die verschiedenen Fälle überschneidungsfrei und erschöpfend sein müssen. Dies bedeutet, daß für jeden möglichen

```
GOAL f_down(gen_baum(3,2,TRUE))
PROGRAM EinlBsp;
  ALG_DEF
    Baum = Knoten(Baum,Baum,BOOLEAN)
         | Blatt(INTEGER);
  END_ALG_DEF
  FUNC_SPEC
    CALL_BY_NAME gen_baum;
    gen_baum : CARDINAL x INTEGER x BOOLEAN -> Baum.
    gen_baum(count,n,b)
      = Blatt(n), IF count = 0,
      = Knoten(gen_baum(count - 1,n,b),Blatt(n),b), OTHERWISE;
  END_FUNC_SPEC
  FUNC_SPEC
    get_zahl : Baum -> INTEGER.
    get_zahl (Knoten(t1,t2,b)) = 0
    get_zahl (Blatt(n)) = n;
  END_FUNC_SPEC
  BEGIN_MAT
    f_down : SYN WALK Baum -> INTEGER;
    f_up : INH WALK Baum x INTEGER -> INTEGER.
    ROOT : Wurzel;
    f_down(<z:Wurzel>) = f_down(<z>.1);
    f_down(<z:Knoten>) = f_down(<z>.1) + f_down(<z>.2);
    f_down(<z:Blatt>) = f_up(<z>,get_zahl(id(<z>)));
    f_up(<z:Wurzel>.1,n) = 0;
    f_up(<z:Knoten>.1,n) = n + f_up(<z>,n);
    f_up(<z:Knoten>.2,n) = f_up(<z>,n);
  END_MAT
END_PROGRAM EinlBsp.
```

Abb. 1.1: Beispiel–Programm `EinlBsp`.

Funktionsaufruf dieser Funktion genau eine passende Funktionsgleichung bestimmt werden kann. Die Funktion `get_zahl` erfüllt diese Bedingung, da sich für alle möglichen Eingaben — nämlich Bäume des Typs `Baum` — genau eine passende Funktionsgleichung findet.

Zwischen den Schlüsselwörtern `BEGIN_MAT` und `END_MAT` erfolgt die Spezifikation eines macro attributed tree transducers (kurz: mat). In NoName*–Programmen können, anders als in NoName, nur mat beschrieben werden. Dies ist aber keine Einschränkung, da die Klasse der mat die größte Klasse von tree transducern ist und die anderen tree transducer Klassen, die in NoName unterstützt werden, enthält.

Allgemein betrachtet beschreibt ein tree transducer, so wie er beispielsweise in [KV94, Fül81, Eng80] beschrieben ist, ein Schema, mit dem eine Baum zu Baum–Übersetzung definiert werden kann. Im wesentlichen handelt es sich dabei um rekursive Funktionen, die sich über einen gegebenen Eingabebaum bewegen. Das Ergebnis der Berechnung sowie alle vorkommenden Parameter sind Bäume desselben Typs.

In NoName* ist dieses Konzept erweitert, so daß als Funktionsergebnistypen nicht nur Bäume, sondern beliebige Datentypen zugelassen sind. Außerdem dürfen die Kontextparameter, die mats neben dem Rekursionsargument — dem ersten Argument — besitzen dürfen, von beliebigem Typ sein.

Bei macro attributed tree transducern gibt es zwei Arten von Funktionen: synthetische und inherite Funktionen. Synthetische Funktionen bewegen sich im Eingabebaum in Richtung der Blätter und inherite Funktionen in Richtung der Wurzel. Die Kennzeichnug von synthetischen und inheriten Funktionen erfolgt durch die Schlüsselwörter `SYN WALK` bzw. `INH WALK` in der Funktionsdeklaration.

Im obigen Beispiel ist `f_down` eine synthetische Funktion, die keine Kontextparameter besitzt. Die inherite Funktion `f_up` hat einen Kontextparameter des Typs `INTEGER`.

Arbeitsweise eines macro attributed tree transducers

Wir wollen nun mit Hilfe der tree transducer Funktionen `f_down` und `f_up` aus Abbildung 1.1 kurz skizzieren, wie ein mat arbeitet. Dazu betrachten wir den Funktionsaufruf

```
f_down(Knoten(Blatt(1),Blatt(4),TRUE)).
```

Über den eigentlichen Eingabebaum wird vor der Berechnung das Symbol `Wurzel` gehängt. Wir erhalten somit einen erweiterten Eingabebaum, den wir in Abbildung 1.2 graphisch dargestellt haben, wobei es sich bei den kleinen Ziffern in der Abbildung um Pfadangaben handelt. Eine Pfadangabe ist eine durch Punkte getrennte Folge von Zahlen, mit deren

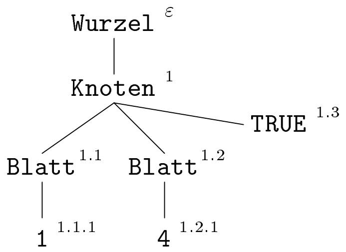


Abb. 1.2: Erweiterter Eingabebaum.

Hilfe ein Knoten in einem Baum *tree* eindeutig identifiziert wird. So verweist die Pfadangabe ε auf die Wurzel von *tree*, die Pfadangabe 1 auf den ersten Nachfolger der Wurzel, usw.

Jede Berechnung eines mat startet an dem Knoten mit der Pfadangabe $\mathtt{z} = \varepsilon$.

Betrachten wir nun die Bedeutung einer einzelnen Funktionsgleichung eines mat. Die Gleichung

```
f_down(<z:Wurzel>) = f_down(<z>.1)
```

besagt nun: Wenn der Knoten im Eingabebaum, der mit der Pfadangabe `z` erreicht wird, mit `Wurzel` beschriftet ist, dann rufe die Funktion `f_down` auf, wobei als Rekursionsargument die um `1` verlängerte Pfadangabe `z` verwendet werden soll.

Die Funktionsgleichung einer inheriten Funktion ist ein wenig anders zu lesen. So bedeutet beispielsweise

```
f_up(<z:Knoten>.2,n) = f_up(<z>,n)
```

folgendes: Nimm die aktuelle Pfadangabe und spalte dabei die letzte Zahl k ab. Betrachte nun den Knoten im Eingabebaum, der mit der verkürzten Pfadangabe `z` erreicht wird. Ist die Beschriftung dieses Knotens `Knoten` und war die abgespaltene Zahl k gleich 2, so rufe die Funktion `f_up` mit der verkürzten Pfadangabe `z` als Rekursionsargument und Kontextparameter `n` auf.

Ausgehend vom ursprünglichen Funktionsaufruf und der anfänglichen Pfadangabe ε werden nun die sich ergebenden Funktionsaufrufe nacheinander abgearbeitet. Für eine genauere Beschreibung der Arbeitsweise eines mat sei der Leser auf [KV94] verwiesen.

Nach dieser kurzen Einführung in die Programmiersprache NoName* kommen wir nun dazu, die Rahmenbedingungen für die zu entwickelnde Implementierung festzulegen.

1.2 Aufgabenstellung

Im Rahmen dieser Diplomarbeit wird für die Programmiersprache NoName* eine Implementierung vorgestellt. Diese Implementierung besteht aus der Angabe der Übersetzungsfunktionen und der Beschreibung der Zielmaschine.

Als Zielmaschine verwenden wir eine abstrakte Runtime–Stack Maschine. Eine Runtime–Stack Maschine, auch Laufzeitkeller–Maschine genannt, besitzt als wesentliche Komponente einen Stack, auf dem alle anfallenden Berechnungen durchgeführt werden. Ein Stack unterscheidet sich von einem Keller in soweit, daß nicht nur das oberste Element betrachtet werden kann, sondern daß zu jedem Zeitpunkt auf beliebige Elemente des Stacks zugegriffen werden kann.

Die Übersetzungsfunktionen erzeugen aus einem korrekten NoName*–Programm ein Maschinenprogramm, welches von der abstrakten Runtime–Stack Maschine abgearbeitet werden kann.

Die Einschränkung von NoName* gegenüber NoName besteht ja darin, daß ein NoName*–Programm keine Funktionen höherer Ordnung enthalten darf. Funktionen höherer Ordnung werden auch als *high–level–functions* bezeichnet. Funktionen, die nicht höherer Ordnung sind, werden folglich als *first–level–functions* bezeichnet. In der Programmiersprache *first–level NoName* liegen also alle NoName–Programme, die keine Funktionen höherer Ordnung enthalten. Ein solches Programm bezeichnen wir auch NoName*–Programm. Mit dieser Überlegung erhalten wir also den Titel dieser Arbeit, nämlich

„Implementierung von first–level NoName auf einer Runtime–Stack Maschine.“

1.3 Implementierung der Sprache NoName*

Bei der Übersetzung erzeugen wir für ein gegebenes NoName*–Programm eine Reihe von Maschinenbefehlen, ein Maschinenprogramm. Dieses Maschinenprogramm kann dann von einer Runtime–Stack Maschine ausgeführt werden.

Prinzipielle Vorgehensweise bei der Übersetzung

Beim Übersetzungsvorgang gehen wir von einem korrekten NoName*–Programm P aus. Zuerst ermitteln wir aus P Informationen, die wir später bei der Anwendung der Übersetzungsfunktionen benötigen. Zu diesen Informationen zählen die definierten Konstruktoren und die angegebenen Auswertungsstrategien.

Mit Hilfe von unterschiedlichen Übersetzungsfunktionen erzeugen wir dann Maschinenbefehle für die Anfrage von P. Danach erzeugen wir einen `stop`–Befehl, damit später die Ausführung des erzeugten Maschinenprogramms nach Abarbeitung der Maschinenbefehle der Anfrage beendet wird. Zum Schluß übersetzen wir der Reihe nach die einzelnen Funktionsspezifikationen. Der sich aus dieser Vorgehensweise ergebende prinzipielle Aufbau des Maschinenprogramms für das NoName*–Programm `EinlBsp` haben wir in Abbildung 1.3 graphisch dargestellt.

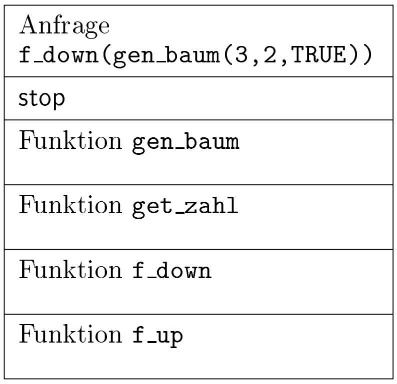


Abb. 1.3: Aufbau des Maschinenprogramms für `EinlBsp`.

Die Definition einer Funktion kann, wie wir am Beispiel von `gen_baum` und `get_zahl` gesehen haben, über eine Fallunterscheidung erfolgen. Nachdem also eine Funktion `f` aufgerufen worden ist, muß zuerst festgestellt werden, welcher der angegebenen Fälle angewendet werden kann. Ist `f` durch eine Fallunterscheidung mittels IF definiert, so müssen bei der Übersetzung Befehle erzeugt werden, die die verschiedenen IF–Fälle in einer bestimmten Reihenfolge abtesten. Ist ein Test positiv, so haben wir den anzuwendenden Fall bestimmt. Trifft kein IF–Fall zu, so findet der OTHERWISE–Fall Anwendung. In Abbildung 1.4 ist der daraus folgende Aufbau des Maschinenprogramms der Funktion `gen_baum` dargestellt.

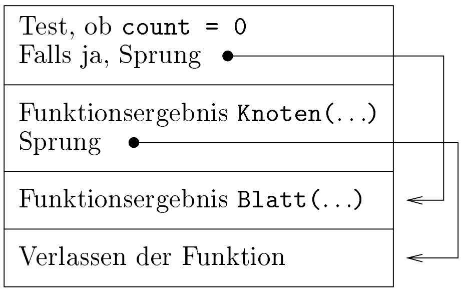


Abb. 1.4: Aufbau des Maschinenprogramms der Funktion `gen_baum`.

Ist `f` mittels Pattern Matching definiert, so gibt es zwei Möglichkeiten, den aktuell anzuwendenden Fall zu bestimmen:

- Man vergleicht die aktuellen Eingabeparameter mit den Mustern auf der linken Seite jeder Funktionsgleichung. Diese Vorgehensweise ist aber sehr ineffizient, da bei n Parametern und m Funktionsgleichungen im schlimmsten Fall $n \cdot m$ Überprüfungen durchgeführt werden müssen.
- Man ermittelt aufgrund der Muster in den linken Seiten der Funktionsgleichungen einen Entscheidungsbaum (vgl. [Pey87]), der durch Betrachten der Eingabeparameter den anzuwendenden Fall ermittelt. Dabei wird jeder Eingabeparameter höchstens einmal betrachtet. Diese Vorgehensweise erfordert höchstens n Überprüfungen.

Wir haben uns hier für die zweite Möglichkeit entschieden. Da jedes Pattern Matching in einem NoName*–Programm überschneidungsfrei und erschöpfend ist, läßt sich solch ein Entscheidungsbaum immer angeben.

Im folgenden wollen wir ein Beispiel für einen solchen Entscheidungsbaum geben. Betrachten wir dazu zuerst die Funktionsgleichungen der Ackermann–Funktion, die in Abbildung 1.5 in NoName*–Syntax wiedergegeben sind.

```
FUNC_SPEC
   acker : CARDINAL x CARDINAL -> CARDINAL.
   acker(0,y) = y+1
   acker(x+1,0) = acker(x,1)
   acker(x+1,y+1) = acker(x,acker(x+1,y));
END_FUNC_SPEC
```

Abb. 1.5: Definition der Ackermann–Funktion.

Den Entscheidungsbaum, der für diese Funktion erzeugt wird, haben wir in Abbildung 1.6 dargestellt. Wie man leicht sehen kann, werden höchstens zwei Überprüfungen durch-

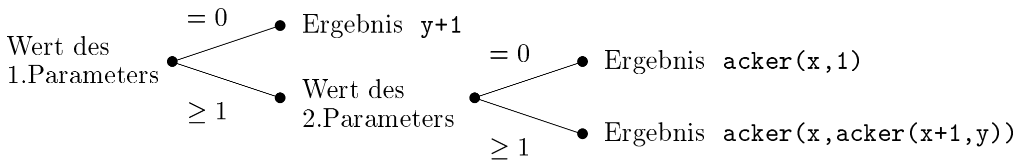


Abb. 1.6: Entscheidungsbaum für die Ackermann–Funktion.

geführt. In machen Fällen ist es sogar möglich, den anzuwendenden Fall nach nur einem Test zu ermitteln. Diese Vorgehensweise hat noch einen weiteren Vorteil. Falls die betrachtete Funktion Call–by–Name ausgewertet werden soll und aufgrund des Entscheidungsbaums nicht alle Parameterwerte ausgewertet werden müssen, kann das Terminierungsverhalten der Funktion verbessert werden.

Aus einem Entscheidungsbaum läßt sich dann eine Befehlssequenz erzeugen, die den anzuwendenden Fall zur Laufzeit ermittelt. Dies wollen wir an dieser Stelle jedoch nicht näher erläutern.

Bearbeitung von Funktionsaufrufen

Betrachten wir nun, wie wir Funktionsaufrufe, die sich im Zuge der Abarbeitung der Anfrage oder der rechten Seite einer Funktionsgleichung ergeben, behandeln. Da die Zielmaschine eine Runtime–Stack Maschine ist, werden alle anfallenden Berechnungen auf einem Stack, also einer kellerartigen Struktur, durchgeführt.

Bei einem Funktionsaufruf legen wir nun einen sogenannten Kellerrahmen auf diesem Stack an. In Abbildung 1.7 haben wir skizziert, wie solch ein Kellerrahmen aufgebaut ist. Die

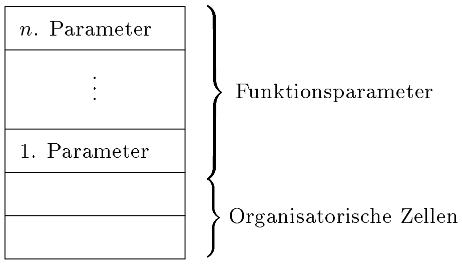


Abb. 1.7: Aufbau eines Kellerrahmens.

Darstellung wächst dabei von unten nach oben. Die unteren beiden Zellen eines jeden Kellerrahmens beinhalten Daten, die zur Verwaltung der einzelnen Funktionsaufrufe notwendig sind. In den oberen Zellen liegen der Reihe nach die einzelnen Funktionsparameter.

Nach Abarbeitung eines Funktionsaufrufs wird der Kellerrahmen wieder entfernt und statt dessen das Ergebnis der Berechnung auf den Stack gelegt. Erfolgen Funktionsaufrufe verschachtelt, z.B. im Zuge einer Rekursion, so können zu einem Zeitpunkt mehrere Kellerrahmen auf dem Stack liegen, nämlich einer für jeden Funktionsaufruf. Auf den obersten Kellerrahmen, der zum momentan bearbeiteten Funktionsaufruf gehört, zeigt ein Register der Maschine, der sogenannte Kellerrahmen–Zeiger.

Möchte man nun im Zuge der Berechnung eines Ausdrucks auf den Wert eines Parameters zugreifen, so geschieht dies, indem man auf die entsprechende Zelle des aktuellen Kellerrahmens zugreift.

Wird eine Funktion nach der Strategie Call–by–Value ausgewertet, so handelt es sich bei den Parametern immer um Werte. Soll eine Funktion nach der Strategie Call–by–Name ausgewertet werden, so verwenden wir als Parameter sogenannte Funktionselemente, die eine Adresse adr und einen Verweis v enthalten. Wird nun im Verlauf einer Berechnung auf eine solches Funktionselement zugegriffen, so wird im Maschinenprogramm an die Adresse adr gesprungen. Die dort stehenden Maschinenbefehle führen dann die Bestimmung des gewünschten Parameterwerts durch. Die dabei wiederum benötigten Paramter werden dann aufgrund des Verweises v ermittelt.

Damit beenden wir die Beschreibung der Behandlung von Funktionsaufrufen. Zum Abschluß der Vorstellung der Implementierung von NoName* wollen wir noch die Zielmaschine, die wir abstrakte NoName–Maschine nennen, kurz beschreiben.

Die abstrakte NoName–Maschine

Die abstrakte NoName–Maschine (kurz: NoMach) besitzt, wie bereits beschrieben, als wesentliche Komponente einen Stack, auf dem die anfallenden Berechnungen durchgeführt werden. Weitere Komponenten der Maschine sind eine Halde, zwei Keller und drei Register. Außerdem gehört zu jeder NoMach ein Maschinenprogramm.

Auf der Halde werden diejenigen Objekte abgespeichert, deren Größe nicht vorherbestimmt werden kann. Dies sind Strukturen, die mit Hilfe von algebraischen Datentypen aufgebaut werden können. Auf den beiden Kellern werden Verwaltungsinformationen, die während eines Maschinenlaufs anfallen, abgelegt. Eines der Register, den Kellerrahmen–Zeiger, haben wir im vorigen Abschnitt bereits kennengelernt. Der Programmzähler — ein weiteres Register — enthält immer die Adresse desjenigen Befehls, der als nächstes ausgeführt werden soll. Im Rückgabewert–Register findet sich nach dem Anhalten der Maschine ein Wert, der den Grund des Anhaltens der Maschine kodiert. Auf diese Weise ist es möglich, Fehler, die sich bei der Abarbeitung der Maschinenbefehle ergeben (z.B. Division durch Null), zu melden. In Abbildung 1.8 haben wir den Aufbau einer NoMach nochmals graphisch dargestellt. Die darin verwendeten Abkürzungen PC, AFP und RC stehen für Programmzähler, Kellerrahmen–Zeiger und Rückgabewert–Register.

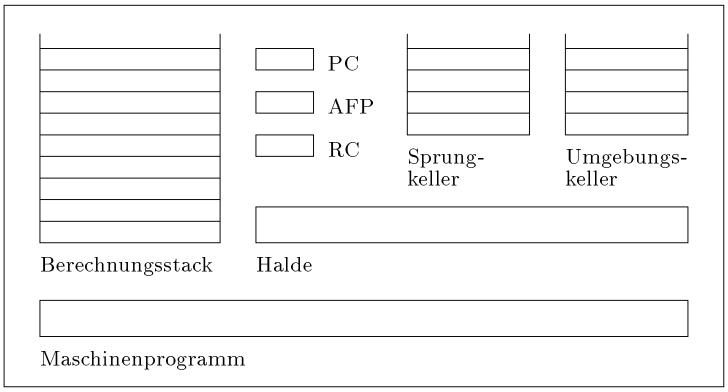


Abb. 1.8: Aufbau einer NoName–Maschine.

1.4 Aufbau der Arbeit

Die für diese Arbeit grundlegenden Begriffe führen wir in Kapitel 2 ein. Es handelt sich dabei um die Beschreibung der hier verwendeten Symbole und um die Definition bekannter Strukturen, wie Keller oder Stack.

In Kapitel 3 beschreiben wir zuerst die Sprache NoName und reduzieren diese dann auf NoName*. Für NoName*–Programme führen wir dann eine formale Notation ein, die es uns ermöglicht, von korrekten Programmen zu sprechen. Auf dieser formalisierten Darstellung von NoName*–Programmen setzen auch die später definierten Übersetzungsfunktionen auf.

Die abstrakte NoName–Maschine beschreiben wir in Kapitel 4. Neben einer informellen Beschreibung findet sich dort auch die genaue Beschreibung der Befehlssemantik jedes einzelnen Maschinenbefehls.

In Kapitel 5 beschreiben wir die Übersetzungsfunktionen, die aus der formalisierten Darstellung eines NoName*–Programms ein Maschinenprogramm erzeugen. Dabei erläutern wir auch die Konzepte, die bei der Definition der Übersetzungsfunktionen Verwendung fanden.

In Kapitel 6 stellen wir eine Umsetzung der Implementierung auf einem realen Rechner vor. Wir beschreiben dort auch die Installation und Bedienung der vorgestellten Software.

Im Ausblick (Kapitel 7) stellen wir einige Überlegungen an, wie die vorgestellte Implementierung verbessert werden kann und wie die hier vorgestellten Ansätze erweitert werden können.

Im Anhang finden sich dann die Beschreibung der kontextfreien Grammatik von NoName* und die Listings der besprochenen Software.

Danksagung

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. Heiko Vogler und Herrn Prof. Dr. Helmuth Partsch für ihre Betreuung und die Erstellung der Gutachten herzlich bedanken. Mein Dank gilt auch Frau Dipl. Math. Andrea Mößle für ihre stete Diskussionsbereitschaft und die wertvollen Ratschläge bei der Erstellung dieser Arbeit.

Ulm, September 1995 *Frank Houdek*

Kapitel 2

Grundlegende Definitionen

In diesem Abschnitt werden wir die grundlegenden Begriffe einführen, die wir im weiteren Verlauf dieses Textes immer wieder benötigen werden. Anhand einiger Beispiele werden wir diese Begriffe zudem verdeutlichen.

Die leere Menge wird mit $\emptyset$ bezeichnet, die Menge der natürlichen Zahlen mit $\mathbb{N}$, die Menge der natürlichen Zahlen einschließlich der Null mit $\mathbb{N}_0$. Für $j \in \mathbb{N}$ ist $[j]$ eine abkürzende Schreibweise für die Menge $\{1, \ldots, j\}$, und $[0]$ bezeichnet die leere Menge.

Die Symbole $\mathbb{Z}$ und $\mathbb{R}$ stehen für die Menge der ganzen Zahlen bzw. die Menge der reellen Zahlen. Als Operatorsymbol für die ganzzahlige Division, also die Division, deren Operandentypen und der Ergebnistyp ganzzahlig sind, wird `div` verwendet. Mit $\lfloor x \rfloor$ wird die größte ganze Zahl bezeichnet, die kleiner oder gleich x ist. Das kleinste Element einer Menge A von Zahlen wird mit $\min A$ notiert.

Das Symbol CH bezeichnet die Menge $\{\texttt{'}x\texttt{'} \mid x \in A\}$, wobei A eine beliebige, aber feste Menge von Zeichen ist. In der Regel ist diese Menge A gleich der Menge der ASCII–Zeichen.

Der Typ einer totalen Funktion f, die als Definitionsbereich die Menge A und als Wertebereich die Menge B hat, wird mit $f : A \rightarrow B$ angegeben. Ist die Funktion f partiell, so wird dies durch die Schreibweise $f : A \relbar\joinrel\rightarrow B$ ausgedrückt. Ist f eine Funktion, so bezeichnet $f[n/m]$ eine Funktion g, mit $g(x) = m$ für $x = n$ und $g(x) = f(x)$ für $x \neq n$.

Seien A und B endliche Mengen. Die Schreibweise $A \backslash B$ steht für die Differenz der Mengen A und B. Das Kreuzprodukt von A und B, also die Menge $\{(a, b) \mid a \in A, b \in B\}$, wird mit $A \times B$ bezeichnet. Die Notation AB steht für die Menge $\{ab \mid a \in A, b \in B\}$. Mit $|A|$ wird die Mächtigkeit einer endlichen Menge A bezeichnet.

Eine Sequenz über einer Menge A ist eine endliche Folge $a_1 \ldots a_n$ von Elementen aus A. Eine leere Sequenz entspricht dem leeren Wort, welches mit ε notiert wird.

Sei A eine endliche Menge. Ist $n \in \mathbb{N}_0$ eine natürliche Zahl, so bezeichnet A^n die Menge $\{a_1 \ldots a_n \mid a_i \in A \text{ für alle } 1 \leq i \leq n\}$, also die Menge aller Sequenzen der Länge n. A^0 ist also folglich die Menge $\{\varepsilon\}$.

Mit A^* und A^+ werden die Mengen $\bigcup\limits_{i=0}^{\infty} A^i$ bzw. $\bigcup\limits_{i=1}^{\infty} A^i$ bezeichnet.

Definition 2.1 (S–sortierte Menge, Sortenmenge, Sorte)

Seien A und S beliebige, endliche Mengen und $sort : A \rightarrow S$ eine totale Funktion. Das Paar $(A, sort)$ heißt *S–sortierte Menge*. Die Menge S heißt *Sortenmenge*, die Elemente von S heißen *Sorte*.

Mit $A^{(s)}$ wird die Menge $\{a \in A \mid sort(a) = s\}$ bezeichnet. □

Ist aus dem Kontext klar, wie die Funktion *sort* beschaffen ist, so kann diese bei der Angabe der S–sortierten Menge weggelassen werden.

Definition 2.2 (Typ eines Elementes einer S–sortierten Menge)

Sei S eine Sortenmenge, $(A, sort)$ eine S–sortierte Menge und $a \in A$ ein Element der Menge A. Dann heißt $sort(a)$ der *Typ von a*. □

Beispiel 2.3 (S–sortierte Menge)

Sei S die Sortenmenge $\{int, bool, char\}$ und $(A, sort)$ die S–sortierte Menge mit $A = \{1, 2, a, b, \texttt{TRUE}, \texttt{FALSE}\}$ und

$$\begin{array}{lll} sort\,(1) = int & sort\,(2) = int & sort\,(a) = char \\ sort\,(b) = char & sort\,(\texttt{TRUE}) = bool & sort\,(\texttt{FALSE}) = bool. \end{array}$$

Abkürzend kann man für $(A, sort)$ auch $\{1^{(int)}, 2^{(int)}, a^{(char)}, b^{(char)}, \texttt{TRUE}^{(bool)}, \texttt{FALSE}^{(bool)}\}$ schreiben.

$A^{(char)}$ steht für die Menge $\{a, b\}$ und der Typ von TRUE ist *bool*. □

Definition 2.4 (Eingabetyp, Resultattyp)

Seien S und T Sortenmengen, $(A, sort)$ eine $(S \times T)$–sortierte Menge und $a \in A$ ein Element von A mit $sort(a) = (s, t)$, wobei $s \in S$ und $t \in T$ gilt.

Dann ist s der *Eingabetyp von a*, bezeichnet mit $input(a)$, und t ist der *Resultattyp von a*, bezeichnet mit $result(a)$. □

Beispiel 2.5 (Eingabetyp, Resultattyp)

Sei S die Sortenmenge $\{int, char\}$, T die Sortenmenge $\{baum\}$ und $(A, sort)$ die $(S \times T)$–sortierte Menge mit $A = \{f, g\}$ und $sort\,(f) = (int, baum)$ sowie $sort\,(g) = (bool, baum)$.

Der Eingabetyp von f, bezeichnet mit $input(f)$, ist folglich *int*, der Resultattyp von g, bezeichnet mit $result(g)$, ist *baum*. □

Definition 2.6 (Keller)

Ein *Keller Pd* über einer Menge M ist eine Sequenz von Elementen aus M, also ein Element aus M^*. Eine *Konfiguration eines Kellers Pd* wird durch $p.1 : p.2 : \ldots : p.n$ beschrieben, wobei $n \geq 0$ und die einzelnen Kellerelemente $p.i$ aus M für $i \in [n]$ sind. Das Element $p.1$ heißt *oberstes Kellerelement*, $p.n$ wird *unterstes Kellerelement* genannt.

Ein *leerer Keller* wird durch das leere Wort, also ε, beschrieben. □

Definition 2.7 (Stack)

Ein *Stack St* über einer Menge M ist ein Tupel (sp, st), wobei $sp \in \mathbb{N}_0$ der *Stackpointer* ist und der *Stackspeicher st* eine partielle Funktion des Typs $\mathbb{N} \dashrightarrow M$ bezeichnet. Die Menge aller Stacks über einer Menge M wird mit $STACK(M)$ bezeichnet.

Ein Stack $(0, st)$ heißt *leerer Stack*. Für einen nicht leeren Stack (sp, st) wird das Element $st(sp)$ *oberstes Stackelement* genannt, das Element $st(1)$ heißt *unterstes Stackelement*. Auf einem Stack sind die Operationen *push*, *top*, *is_empty*, *pop*, *read*, *write* und *height* definiert. Diese Operationen modifizieren einen Stack $(sp, st) \in STACK(M)$ in folgender Weise.

$push : STACK(M) \times M \to STACK(M)$
mit $push((sp, st), x) = (sp + 1, st[(sp + 1)/x])$

$top : STACK(M) \to M \cup \{\perp\}$
mit $top((sp, st)) = \begin{cases} st(sp), & \text{falls } sp \geq 1 \\ \perp, & \text{sonst} \end{cases}$
wobei $\perp$ ein Symbol ist, welches nicht Element der Menge M ist.

$is\_empty : STACK(M) \to \{\texttt{TRUE}, \texttt{FALSE}\}$
mit $is\_empty((sp, st)) = \begin{cases} \texttt{TRUE}, & \text{falls } sp = 0 \\ \texttt{FALSE}, & \text{sonst} \end{cases}$

$pop : STACK(M) \to STACK(M)$
mit $pop((sp, st)) = \begin{cases} (sp - 1, st), & \text{falls } sp \geq 1 \\ (0, st), & \text{sonst} \end{cases}$

$read : STACK(M) \times \mathbb{N} \dashrightarrow M$
mit $read((sp, st), n) = st(n)$, falls $n \in [sp]$

$write : STACK(M) \times \mathbb{N} \times M \dashrightarrow STACK(M)$
mit $write((sp, st), n, m) = (sp, st[n/m])$, falls $n \in [sp]$

$height : STACK(M) \to \mathbb{N}_0$
mit $height((sp, st)) = sp$

Anstelle von $read(St, n)$ mit $St = (sp, st)$ kann auch die abkürzende Schreibweise $St[n]$ verwendet werden. □

Beispiel 2.8 (Stack und Anwendung einiger Stackoperationen)

Sei $My\_St = (0, my\_st)$ ein leerer Stack über der Menge der natürlichen Zahlen. In Abbildung 2.1 sind (a) My_St sowie die Zustände nach den Stackoperationen (b) $push(My\_St, 3)$, (c) $push(push(My\_St, 3), 7)$ und (d) $pop(push(push(My\_St, 3), 7))$ graphisch dargestellt.

(a) (b) (c) (d)

Abb. 2.1: Darstellung einiger Stackoperationen.

Der Ausdruck $height(push(My\_St, 3))$ ergibt 1, da sich nach der Operation $push(My\_St, 3)$ genau ein Element auf dem so entstandenen Stack befindet.

Der Ausdruck $top(pop(push(push(My\_St, 3), 7)))$ ergibt 3, da nach der Stackoperation (d) die Zahl 3 das oberste Stackelement ist. □

Definition 2.9 (Halde)

Eine *Halde* Hp über einer Menge M ist eine partielle Funktion des Typs $\mathbb{N} \dashrightarrow M$. Die Menge aller Halden über einer Menge M wird mit $HEAP(M)$ bezeichnet.

Auf einer Halde sind die Operationen *new*, *modify* und *deref* definiert. Diese Operationen modifizieren eine Halde Hp in der folgenden Weise.

$new : HEAP(M) \times M \to HEAP(M) \times \mathbb{N}$
mit $new(Hp, m) = (Hp', n)$
wobei $Hp' = Hp[n/m]$ und $n = \min\{k \mid k \in \mathbb{N}, Hp[k] \text{ undefiniert}\}$

$modify : HEAP(M) \times \mathbb{N} \times M \dashrightarrow HEAP(M)$
mit $modify(Hp, n, m) = Hp[n/m]$, falls $Hp(n)$ definiert

$deref : HEAP(M) \times \mathbb{N} \dashrightarrow M$
mit $deref(Hp, n) = Hp(n)$, falls $Hp(n)$ definiert

Anstelle von $deref(Hp, n)$ kann auch die abkürzende Schreibweise $Hp[n]$ verwendet werden. □

Definition 2.10 (Aufzählungsfunktion)

Sei M eine endliche Menge. Eine Funktion $enum : M \to [|M|]$ heißt *Aufzählungsfunktion für* M, wenn sie injektiv ist, d.h. wenn für alle $m_1, m_2 \in M$ die Aussage $m_1 = m_2 \Leftrightarrow enum(m_1) = enum(m_2)$ gilt. □

Kapitel 3

Die Sprache NoName

In diesem Kapitel stellen wir die funktionale Programmiersprache NoName vor. Ausgehend von dieser Darstellung führen wir dann eine Teilsprache von NoName, die wir NoName* nennen, ein. Für diese Sprache stellen wir im Rahmen dieser Arbeit eine Implementierung vor. In Abschnitt 3.3 definieren wir eine formalisierte Version von NoName*, die als Basis für die in Kapitel 5 definierten Übersetzungsfunktionen dient.

3.1 Einführung in die Sprache NoName

Die Sprache NoName wurde 1994 von einer Gruppe Studenten unter der Leitung von Prof. Dr. Heiko Vogler im Rahmen eines Praktikums an der Universität Ulm entwickelt. Die neu entwickelte Sprache sollte einige Eigenschaften besitzen, die in dieser Zusammenstellung bei keiner anderen Programmiersprache zu finden waren:

- strenges Typkonzept,
- mathematische Syntax,
- Möglichkeit der expliziten Angabe einer Auswertungsstrategie für jede Funktion,
- Unterstützung der Programmierung von tree transducern,
- benutzerdefinierte algebraische Datentypen und
- Funktionen höherer Ordnung.

Die mathematische Syntax hat zur Folge, daß insbesondere das Currying, also das Erzeugen von Funktionen höherer Ordnung durch Unterversorgung, nicht möglich ist.

Anhand einiger Beispiele werden wir im folgenden die Programmiersprache NoName informell einführen. Wir weisen jedoch darauf hin, daß dieser Abschnitt lediglich ein intuitives Verständnis vermitteln will. Der interessierte Leser sei also in jedem Fall auf die Sprachbeschreibung der Sprache NoName ([EGH+94]) verwiesen.

Datentypen

Die in NoName bekannten Basisdatentypen sind `CARDINAL` (natürliche Zahlen), `INTEGER` (ganze Zahlen), `REAL` (reelle Zahlen), `BOOLEAN` (Wahrheitswerte) und `CHARACTER` (Zeichen). Darüber hinaus ist es möglich, algebraische Datentypen zu definieren. Die Definition eines algebraischen Datentyps `Baum`, mit dessen Hilfe spezielle Binärbäume erzeugt werden können, ist in Abbildung 3.1 wiedergegeben. Die Konstruktorsymbole diese Datentyps sind `Knoten` und `Blatt`.

```
ALG_DEF
  Baum = Knoten(Baum,Baum,BOOLEAN)
       | Blatt(INTEGER);
END_ALG_DEF
```

Abb. 3.1: Definition des algebraischen Datentyps `Baum`.

Ein möglicher Ausdruck des Typs `Baum` wäre `Knoten(Blatt(3),Blatt(7),FALSE)`.

Außerdem können in NoName, wie die Abbildung 3.2 zeigt, Abkürzungen für einen oder mehrere bekannte Datentypen definiert werden, die sogenannten Datentypsynonyme.

```
TYPE_DEF
  Komplex = REAL x REAL;
  Struktur = Baum x Komplex x Komplex;
END_TYPE_DEF
```

Abb. 3.2: Definition von Datentypsynonymen.

Dieser Punkt hat aber wenig Bedeutung, da durch diese Technik keine echten Datentypen geschaffen werden (vgl. [EGH+94], Seite 8).

Funktionen

Zur besseren Unterscheidung der verschiedenen Arten von Funktionen teilen wir die Menge der Funktionen in zwei Gruppen auf: die tree transducer Funktionen und die gewöhnlichen Funktionen.

Funktionen werden innerhalb von Spezifikationen erklärt, wobei eine Spezifikation sich aus einer Folge von Deklarationen und einer Folge von Definitionen zusammensetzt. In der Deklaration werden die Typen — und damit auch die Stelligkeit — von Eingabe und Resultat festgelegt. Die Definition ist dann die Angabe, wie der Funktionswert, das Resultat, aus der Eingabe zu bestimmen ist. Eine Definition zerfällt in linke und rechte Seite der Funktionsgleichung mit dem Gleichheitssymbol als Trenner.

$$\underbrace{f(x, y, z)}_{\text{linke Seite}} = \underbrace{(x + y) * z}_{\text{rechte Seite}}$$

Jede Funktionsdefinition muß linkslinear und typkorrekt sein. Eine Funktionsdefinition ist linkslinear, wenn kein Variablenname zweimal auf der linken Seite einer Funktionsgleichung auftritt.

Als Infixoperatoren stehen die üblichen arithmetischen und vergleichenden Operatoren zur Verfügung. Jede Funktion kann jede andere, unabhängig von deren Position im Programmtext, aufrufen. Eine Ausnahme von dieser Regel bilden, wie wir später sehen werden, die inheriten tree transducer Funktionen, die nur von Funktionen desselben tree transducers aufgerufen werden dürfen.

Betrachten wir nun, wie gewöhnliche Funktionen sowie tree transducer Funktionen in NoName dargestellt werden.

Gewöhnliche Funktionen

In Abbildung 3.3 findet sich ein Beispiel für die Spezifikation einer gewöhnlichen Funktion.

```
FUNC_SPEC
   f : REAL x REAL x BOOLEAN -> BOOLEAN.
   f(u,v,y) = y and (u = v);
END_FUNC_SPEC
```

Abb. 3.3: Einfache Funktionsspezifikation.

Neben der in Abbildung 3.3 dargestellten einfachen Funktionsdefinition ist auch eine Funktionsdefinition mittels Pattern Matching und via IF möglich. In Abbildung 3.4 ist die Fibonacci–Funktion mittels Pattern Matching definiert, in Abbildung 3.5 via IF.

```
FUNC_SPEC
   fib : CARDINAL -> CARDINAL.
   fib(0) = 1
   fib(1) = 1
   fib(n+2) = fib(n) + fib(n+1);
END_FUNC_SPEC
```

Abb. 3.4: Fibonacci mit Pattern Matching.

```
FUNC_SPEC
   fib : CARDINAL -> CARDINAL.
   fib(n) = 1, IF (n = 0) or (n = 1),
          = fib(n-1) + fib(n-2),
            OTHERWISE;
END_FUNC_SPEC
```

Abb. 3.5: Fibonacci mittels IF.

Die Funktionsdefinition in Abbildung 3.3 ist ein Spezialfall einer Funktionsdefinition mittels Pattern Matching. Ein Mischen der beiden Verfahren — Funktionsdefinition mittels Pattern Matching oder via IF — ist nicht zulässig. Die Fälle, die durch die verschiedenen IF–Klauseln angegeben werden, müssen überschneidungsfrei und erschöpfend sein. Diese Bedingung kann aber nicht überprüft werden, da sich hinter dieser Bedingung ein unentscheidbares Problem verbirgt. Um aber den Programmierer anzuhalten, dennoch diese Bedingung einzuhalten, darf der Übersetzer die Reihenfolge, in der die einzelnen IF–Fälle getestet werden, beliebig festlegen. Nur durch das Einhalten der Überschneidungsfreiheit kann also der Programmierer unliebsame Effekte vermeiden. Der `OTHERWISE`–Fall ist nicht zwingend notwendig.

Pattern Matching ist nur auf induktiven Datentypen, also `CARDINAL`, `BOOLEAN` sowie algebraischen Datentypen erlaubt. In Abbildung 3.6 findet sich ein Beispiel für Pattern Matching auf algebraischen Datentypen, wobei der Datentyp `Baum` derselbe wie in Abbildung 3.1 ist. Die Funktion `blaetter` berechnet die Anzahl der Blätter eines gegebenen Binärbaums. Jedes Pattern Matching muß überschneidungsfrei und erschöpfend sein, d.h. für jeden möglichen Eingabevektor, also eine Belegung der Funktionsparameter, muß es genau eine Funktionsgleichung geben, die zu diesem Eingabevektor paßt.

```
FUNC_SPEC
  blaetter : Baum -> CARDINAL.
  blaetter(Blatt(x)) = 1
  blaetter(Knoten(t1,t2,b)) = blaetter(t1) + blaetter(t2);
END_FUNC_SPEC
```

Abb. 3.6: Pattern Matching auf algebraischem Datentyp.

Als Eingabe– und Resultattypen sind bei gewöhnlichen Funktionen auch Funktionstypen, also Angaben der Form (t_1 -> t_2), möglich, wobei t_1 und t_2 wiederum beliebige Typen sind. In diesem Fall spricht man von einer Funktion höherer Ordnung. In Abbildung 3.7 finden sich zwei Beispiele für Funktionen höherer Ordnung. Die Funktion `f_op` wendet ihr erstes Argument — eine Funktion — auf die beiden anderen Argumente an. Mit `inc_n` wird eine Inkrementierungsfunktion realisiert.

```
FUNC_SPEC
  f_op : (REAL -> REAL) x REAL x REAL -> REAL;
  inc_n : CARDINAL -> (CARDINAL -> CARDINAL).
  f_op(f,a,b) = f(a,b);
  inc_n(n) = help_inc_n
         WHERE FUNC_SPEC
                 help_inc_n : CARDINAL -> CARDINAL.
                 help_inc_n(m) = n + m;
               END_FUNC_SPEC;
END_FUNC_SPEC
```

Abb. 3.7: Funktionen höherer Ordnung.

Mit Hilfe des `WHERE` Schlüsselwortes ist es auch möglich, lokale Hilfsfunktionen zu definieren, ohne daß Funktionen höherer Ordnung Verwendung finden.

Tree transducer

NoName unterstützt als bisher einzige Sprache auch auf syntaktischer Ebene die Definition von verschiedenen tree transducern:

- top–down tree transducer [Rou70, Tha70],
- macro tree transducer [Eng80, CF82, EV85],
- attributed tree transducer [Fül81],
- macro attributed tree transducer [KV94] und
- modular tree transducer [EV91].

Diese verschiedenen tree transducer wollen wir hier jedoch nicht detailliert vorstellen. Der interessierte Leser sei diesbezüglich auf die Literatur verwiesen.

Wir wollen hier nur anhand von macro attributed tree transducern (mat) kurz die grundlegenden Konzepte der tree transducer darlegen. Ein tree transducer im eigentlichen Sinne ist ein Baum zu Baum–Übersetzer, wobei diese Übersetzung mit Hilfe von rekursiven Funktionen durchgeführt wird. Eingabe und Resultat sind Bäume über beliebigen, aber festen algebraischen Datentypen. Der Eingabebaum t, der übersetzt werden soll, bleibt während der Berechnung unverändert. Die Funktionen bewegen sich mittels Pfadangaben über diesen Baum t. In Abbildung 3.8 ist ein solcher Eingabebaum dargestellt, wobei die einzelnen Knoten mit den zugehörigen Pfadangaben beschriftet sind.

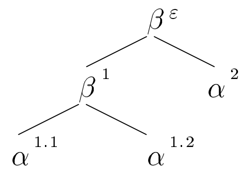


Abb. 3.8: Eingabebaum mit Pfadangaben.

Bei mats gibt es zwei verschiedene Arten von tree transducer Funktionen: synthetische und inherite Funktionen. Synthetische Funktionen bewirken im Eingabebaum eine Bewegung in Richtung der Blätter (durch Verlängern der Pfadangabe) und inherite Funktionen eine Bewegung in Richtung der Wurzel (durch Verkürzen der Pfadangabe). In Abbildung 3.9 ist ein mat mit der synthetischen Funktion f_s und der inheriten Funktion f_h wiedergegeben.

$$
\begin{array}{lll@{\qquad}lll}
R_\beta: & (1) & f_s\,(z, y_1) = f_s\,(z.1, y_1) & R_{root}: & (4) & f_s\,(z, y_1) = f_s\,(z.1, y_1) \\
 & (2) & f_h\,(z.1, y_1) = \beta\,(f_s\,(z.2, \alpha), y_1) & & (5) & f_h\,(z.1, y_1) = \alpha \\
 & (3) & f_h\,(z.2, y_1) = f_h\,(z, y_1) & R_\alpha: & (6) & f_s\,(z, y_1) = f_h\,(z, y_1)
\end{array}
$$

Abb. 3.9: Beispiel für einen mat.

Die Regel (1) für die Funktion f_s beispielsweise ist in der folgenden Weise zu lesen: Wenn der aufgrund der aktuellen Pfadangabe betrachtete Knoten im Eingabebaum mit β beschriftet ist (wegen R_β), dann rufe die synthetische Funktion f_s auf und steige in den ersten Teilbaum ab ($z.1$), wobei der zweite Funktionsparameter unverändert übernommen wird ($f_s\,(z.1, y_1)$).

Die Regel (3) für die Funktion f_h bedeutet: Versuche die aktuelle Pfadangabe mit $z.2$ zu unifizieren. Gelingt dies und ist der Knoten, der mit der verkürzten Pfadangabe z erreicht wird, mit β beschriftet, so rufe mit dieser verkürzten Pfadangabe z die inherite Funktion f_h auf, wobei auch hier der zweite Funktionsparameter unverändert übernommen wird.

Ein wenig verwunderlich mag erscheinen, daß es außer Regeln für die Knotenbeschriftungen α und β auch noch eine Regel für das sogenannte Wurzelsymbol *root* gibt, welches im Eingabebaum nicht auftaucht. Dies liegt daran, daß durch Verwendung von inheriten Funktionen versucht werden könnte, auf den Knoten oberhalb der Wurzel des Eingabebaums zuzugreifen, was freilich nicht möglich ist. Um dieses Problem zu lösen, wird über den eigentlichen Eingabebaum das Wurzelsymbol gehängt. Der so entstandene Baum wird erweiterter Eingabebaum genannt. Die Regeln, die dem Wurzelsymbol zugeordnet sind, dürfen auf der rechten Seite keine inheriten Funktionen rufen. Mit diesem Mechanismus wird eine unzulässige Bewegung im Baum verhindert. Das Wurzelsymbol *root* wird uns im weiteren Verlauf dieser Arbeit immer wieder begegnen.

In Abbildung 3.11 finden sich die ersten Schritte der Rechnung des mat aus Abbildung 3.9 über dem erweiterten Eingabebaum aus Abbildung 3.10. Die hochgestellten Nummern in Abbildung 3.11 beziehen sich auf die Regelnummern aus Abbildung 3.9.

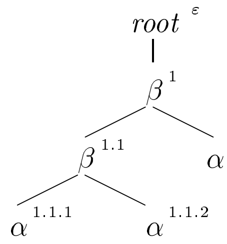


Abb. 3.10: Erweiterter Eingabebaum.

$$
\begin{array}{rl}
 & f_s\,(\varepsilon, \beta\,(\alpha, \alpha)) \\
=^4 & f_s\,(1, \beta\,(\alpha, \alpha)) \\
=^1 & f_s\,(1.1, \beta\,(\alpha, \alpha)) \\
=^1 & f_s\,(1.1.1, \beta\,(\alpha, \alpha)) \\
=^6 & f_h\,(1.1.1, \beta\,(\alpha, \alpha)) \\
=^2 & \beta\,(f_s\,(1.1.2, \alpha), \beta\,(\alpha, \alpha)) \\
= & \ldots
\end{array}
$$

Abb. 3.11: Rechnung eines mat.

Der strukturelle Aufbau von mat–Spezifikationen in NoName ist dem Aufbau von Spezifikationen bei gewöhnlichen Funktionen sehr ähnlich. So muß auch jede tree transducer Funktion deklariert werden. Zwischen den Deklarationen und Definitionen steht die Angabe eines Namens für das Wurzelsymbol. Allerdings hat hier eine Spezifikation eine größere Bindungskraft als bei gewöhnlichen Funktionen: Alle Funktionen eines mats müssen innerhalb eines Spezifikationsblocks angegeben werden. In Abbildung 3.12 ist der mat aus Abbildung 3.9 in NoName–Syntax wiedergegeben.

```
ALG_DEF
  MatBaum = Beta (Baum, Baum)
          | Alpha;
END_ALG_DEF
BEGIN_MAT
  f_s : SYN WALK MatBaum x MatBaum -> MatBaum;
  f_h : INH WALK MatBaum x MatBaum -> MatBaum.
  ROOT : MatRoot;
  f_s(<z:MatRoot>,y1) = f_s(<z.1>,Alpha);
  f_h(<z:MatRoot>.1,y1) = Alpha;
  f_s(<z:Beta>,y1) = f_s(<z.1>,y1);
  f_h(<z:Beta>.1,y1) = Beta(f_s(<z.2>,Alpha),Alpha);
  f_h(<z:Beta>.2,y1) = f_h(<z>,y1);
  f_s(<z:Alpha>,y1) = f_h(<z>,y1);
END_MAT
```

Abb. 3.12: Macro attributed tree transducer in NoName–Syntax.

In NoName weichen tree transducer aber etwas von der eben vorgestellten und aus der Literatur bekannten Form ab. Während der erste Parameter — das Rekursionsargument — auch hier ein Baum über einem algebraischen Datentyp sein muß, können die weiteren Parameter einen beliebigen Typ haben. Lediglich Funktionstypen sind nicht erlaubt.

Auch dürfen auf der rechten Seite der tree transducer Funktionsgleichungen externe Funktionen, also Funktionen, die nicht zum spezifizierten tree transducer gehören, aufgerufen werden. Hier ergibt sich jedoch ein Problem: Soll sich die aufgerufene Funktion f auf Teile des Eingabebaums beziehen, so kann dies nicht mit f(<z>) geschehen, da <z> für eine Pfadangabe und nicht für einen Baum steht. Um dennoch den gewünschten Teilbaum zu erhalten, schreibt man anstelle von <z> bzw. <z>.k nun id(<z>) bzw. id(<z>.k), wobei k die Nummer des gewünschten Teilbaums ist.

Die weiteren Arten von tree transducern und deren Realisierung in NoName werden wir hier nicht näher erläutern. Die folgende Übersicht soll nur kurz die Unterschiede zwischen den verschiedenen tree transducer Arten andeuten.

- top–down tree transducer
  In top–down tree transducern dürfen nur synthetische Funktionen mit einem Parameter, nämlich dem Rekursionsargument, verwendet werden. Die Angabe eines Namens für das Wurzelsymbols entfällt, da in einem top–down tree transducer keine inheriten Funktionen enthalten sein können.
- macro tree transducer
  Auch macro tree transducer enthalten nur synthetische Funktionen, die aber, wie auch mats, beliebig viele Parameter haben dürfen. Die Angabe eines Namens für das Wurzelsymbol entfällt auch hier.
- attributed tree transducer
  Ein attributed tree transducer darf synthetische und inherite Funktionen besitzen. Allerdings haben alle Funktionen nur einen Parameter, nämlich das Rekursionsargument. Die Angabe eines Namens für das Wurzelsymbol ist notwendig, da inherite Funktionen verwendet werden dürfen.
- modular tree transducer
  Mehrere top–down und macro tree transducer können in einem modular tree transducer zusammengefaßt werden. Bei gegenseitigen Aufrufen von Funktionen aus verschiedenen tree transducern ist eine bestimmte Aufrufhierarchie einzuhalten.

Zur genauen Syntax der tree transducer in NoName sei der Leser auch hier auf die Sprachbeschreibung [EGH+94] verwiesen.

Weitere Eigenschaften von NoName

Kommentare in NoName–Programmen werden durch die Zeichenkombinationen `(%` und `%)` begrenzt. Geschachtelte Kommentare sind nicht möglich.

Wie oben angeführt, kann für jede Funktion eine der Auswertungsstrategien *Call–by–Value* oder *Call–by–Name* angegeben werden. Betrachten wir dazu die Funktionsgleichung

$$\texttt{f(x) = } e\texttt{,}$$

wobei e ein beliebiger Ausdruck ist. Bei der Auswertungsstrategie Call–by–Value wird der Wert von `x` bestimmt, bevor der Ausdruck e ausgewertet wird. Im Falle der Auswertungsstrategie Call–by–Name wird mit der Auswertung von e begonnen, und immer wenn in e der Wert für `x` benötigt wird, wird dieser bestimmt.

Die Angabe der Auswertungstrategie Call–by–Value erfolgt, indem hinter das Schlüsselwort `FUNC_SPEC` bzw. `BEGIN_MAT` der Text

$$\texttt{CALL\_BY\_VALUE } id_1\texttt{, } id_2\texttt{, } \ldots\texttt{, } id_n\texttt{;}$$

gesetzt wird. Die Bezeichner $id_1, \ldots, id_n$ sind dabei die Namen der Funktionen, für die eine Festlegung bezüglich der anzuwendenden Auswertungsstrategie getroffen werden soll.

Die Angabe der Auswertungsstrategie Call–by–Name erfolgt in analoger Weise. Für eine Funktion darf höchstens eine Auswertungsstrategie angegeben werden.

Mit dem Import–Statement ist es möglich, Funktionen und Datentypen aus anderen Programmen im eigenen Programm zu verwenden. Durch die Angabe von

`FROM` *DateiName* `IMPORT` id_1, id_2, ..., id_n;

im Programm P können $id_1, \ldots, id_n$ sowie alle von $id_1, \ldots, id_n$ benötigten Hilfsfunktionen und Datentypen in P verwendet werden.

3.2 Reduzierung von NoName auf NoName*

Im Rahmen dieser Diplomarbeit werden wir für eine Teilsprache von NoName, die wir mit NoName* bezeichnen, einen Übersetzer definieren und implementieren. Die Haupteinschränkung von NoName* gegenüber NoName besteht darin, daß keine Funktionen höherer Ordnung zugelassen sind. Weitere Restriktionen sind:

- Als einzige tree transducer Art werden nur mats unterstützt.
  Da sich aber jeder der von NoName unterstützten tree transducer als mat aufschreiben läßt, stellt diese Vereinfachung keine wirkliche Einschränkung dar. Diese Transformation kann, wie in [Ern95] gezeigt, automatisch erfolgen.
- Es werden keine Typspezifikationen mittels `TYPE_DEF` ... `END_TYPE_DEF` unterstützt. Derartig eingeführte Typdefinitionen erhöhen nur die Lesbarkeit von Funktionsdeklarationen (vgl. [EGH+94], Seite 8). Mittels eines einfachen Algorithmus ist es möglich, diese Typdefinitionen automatisch aufzulösen:
  - ⋄ Erstellen eines Abhängigkeitsgraphen der Typdefinitionen
  - ⋄ topologisches Sortieren der Typdefinitionen des Abhängigkeitsgraphen
  - ⋄ schrittweises Ersetzen der Typdefinitionen entlang der Sortierung

  In dem folgenden Beispiel wird diese Vorgehensweise verdeutlicht.

  Beispiel 3.1

  Gegeben sei ein NoName–Programm P mit den Typdefinitionen, wie sie in Abbildung 3.13 angegeben sind.

```
TYPE_DEF
  Komplex = REAL x REAL;
  Kugel = Komplex x INTEGER;
  Objekt = Quad x Kugel;
  Quad = Komplex x Komplex;
END_TYPE_DEF
```

Abb. 3.13: Abhängige Typdefinitionen.

Zwischen diesen Datentypen bestehen gewisse Abhängigkeiten: so verwendet beispielsweise die Definition von `Quad` den Datentyp `Komplex`. Diese Typdefinitionen induzieren also einen Abhängigkeitsgraphen; dieser ist in Abbildung 3.14 dargestellt.

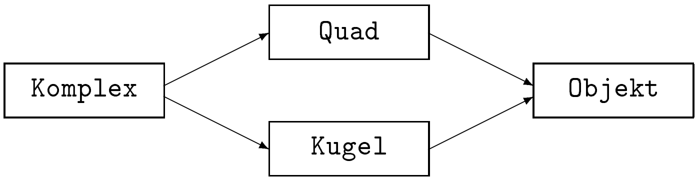


Abb. 3.14: Induzierter Abhängigkeitsgraph.

Eine mögliche topologische Sortierung, die aus diesen Abhängigkeitsgraphen ermittelt werden kann, ist

$$\texttt{Komplex} \rightarrow \texttt{Kugel} \rightarrow \texttt{Quad} \rightarrow \texttt{Objekt}.$$

Diese Sortierung besagt nun, daß auf der rechten Seite der Definition des j–kleinsten Elements in der Sortierung nur Basisdatentypen sowie die Elemente $1, \dots, (j-1)$ der Sortierung verwendet werden.

Anhand dieser Sortierung können wir nun die Typdefinitionen schrittweise auflösen und erhalten die folgenden Ersetzungen.

`Komplex`	durch	`REAL x REAL`
`Kugel`	durch	`REAL x REAL x INTEGER`
`Quad`	durch	`REAL x REAL x REAL x REAL`
`Object`	durch	`REAL x REAL x REAL x REAL x REAL x REAL x INTEGER`

Diese Ersetzungen können nun auf P angewandt werden. Somit sind die obigen Typdefinitionen für das modifizierte P unnötig und können entfernt werden. □

- Das Import–Statement wird nicht unterstützt.
  Dies läßt sich damit begründen, daß das Import–Statement keine Spracheigenschaft im eigentlichen Sinne ist, sondern vielmehr eine Vereinfachung beim Entwickeln von Programmen darstellt. Auch das Import–Statement läßt sich automatisch auflösen. In [EHM94] findet sich dafür eine mögliche Implementierung.
- Das Schlüsselwort `WHERE` darf nicht verwendet werden.
  Außer zur Definition von Funktionen höherer Ordnung konnte `WHERE` auch verwendet werden, um Hilfsfunktionen zu definieren. Dies stellt aber auch keine wirkliche Erweiterung des Sprachumfangs dar. Die mittels `WHERE` eingeführten Hilfsfunktionen lassen sich, ggf. durch Umbenennung, ebenfalls ohne `WHERE` ausdrücken.

Falls also ein NoName–Programm vorliegt, das keine Funktionen höherer Ordnung verwendet, so läßt sich dieses aufgrund der eben gemachten Überlegungen automatisch in eine Form bringen, die den eben erwähnten Restriktionen genügt.

Eine weitere Änderung von NoName* gegenüber NoName besteht darin, daß die Anfrage an das NoName*–Programm zu diesem dazugehört. An dem Programm `EinlBsp` in Abbildung 1.1 ist dies beispielsweise zu sehen. Dieser Punkt ist auf den ersten Blick ein wenig verwunderlich, da ein Programm in der Regel für viele verschiedene Anfragen gedacht ist. Diese Änderung wurde aber in Hinblick auf die zu definierende Implementierung gemacht. Die Sprache NoName besitzt keine Konstrukte, die die Ein– oder Ausgabe von Parametern bzw. Ergebnissen zuläßt. Da für eine Anfrage aber auch Code erzeugt werden muß, haben wir uns entschieden, die Anfrage als einen Bestandteil des zu übersetzenden Programms zu betrachten.

3.3 Formale Definition der Sprache NoName*

In der Art und Weise, wie bisher NoName– und NoName*–Programme aufgeschrieben worden sind, enthalten sie sehr viel strukturierende Syntax, die nichts zum Wesen der definierten Funktionen beiträgt. Auch wurden bisher die kontextsensitiven Nebenbedingungen nur kurz und informell beschrieben, so daß hier immer noch ein unvermeidlicher Interpretationsspielraum bleibt.

Deshalb wird in diesem Abschnitt eine formale Version von NoName*–Programmen eingeführt, die aber äquivalent zur bisher vorgestellten „syntaktischen Version" ist. Anhand der formalen Version wird es uns aber relativ leicht möglich sein, die kontextsensitiven Nebenbedingungen zu überprüfen und damit bezüglich der statischen Semantik korrekte Programme zu erhalten.

Die sich bei der Formalisierung ergebenden Definitionen werden leider teilweise ziemlich umfangreich. Wir wollen jedoch versuchen, durch Beispiele und Erläuterungen diese Definitionen verständlicher zu machen.

Um die korrekte Typisierung von Ausdrücken und Teilausdrücken sicherzustellen, verwenden wir sortierte Mengen. Die Menge aller Datentypen bildet dabei die zugrundeliegende Sortenmenge S. Um sicherzustellen, daß die Basisdatentypen von NoName, nämlich `CARDINAL`, `INTEGER`, `REAL`, `BOOLEAN` und `CHARACTER`, immer in S enthalten sind, führen wir den Begriff der Basis–Sortenmenge ein.

Definition 3.2 (Basis, Basis–Sortenmenge)

Die Menge $B = \{card, int, real, bool, char\}$ heißt *Basis*. Eine Sortenmenge S heißt *Basis–Sortenmenge*, wenn $B \subseteq S$ gilt. Mit $\hat{S}$ wird dann die Menge $S \backslash B$ bezeichnet. □

Die Menge $\hat{S}$ wird von den benutzerdefinierten algebraischen Datentypen gebildet. Vor diesem Hintergrund ist klar, daß die Menge *Var* der Variablen eine S–sortierte Menge ist, da jede Variable in einem Ausdruck einen bestimmten Typ hat. Die Menge F aller Funktionssymbole ist folglich eine $(S^* \times S^+)$–sortierte Menge. Der Eingabetyp einer Funktion ist ein Element der Menge S^*, da eine Funktion beliebig viele (evtl. auch keine) Eingabeparameter hat, und der Resultattyp ist ein Element der Menge S^+, da jede Funktion ein nichtleeres Resultat liefert.

Beispiel 3.3

Betrachten wir nochmals die bereits aus Abbildung 3.3 bekannte Funktionsspezifikation, die wir in Abbildung 3.15 wiederholt haben.

```
FUNC_SPEC
  f : REAL x REAL x BOOLEAN -> BOOLEAN.
  f(u,v,y) = y and (u = v);
END_FUNC_SPEC
```

Abb. 3.15: Einfache Funktionsspezifikation.

Die Funktion f hat folglich den Typ $(real\ real\ bool, bool)$, es gilt $f \in F^{(real\ real\ bool, bool)}$. Die sich aus dieser Funktionsspezifikation ergebende Menge von Variablen ist $Var = \{u^{(real)}, v^{(real)}, y^{(bool)}\}$. □

Mit einer analogen Überlegung folgt sofort, daß die Menge Δ der Konstruktorsymbole eine $(S^* \times \hat{S})$–sortierte Menge sein muß. Der Eingabetyp eines Konstruktors ist ein Element aus der Menge S^*, da ein Konstruktor beliebig viele Nachfolger haben kann, und der Resultattyp ist ein Element der Menge $\hat{S}$, da der Typ eines Ausdrucks $d\,(t_1, \ldots, t_n)$ mit $d \in \Delta$ immer ein algebraischer Datentyp, also ein Element aus $\hat{S}$, sein muß.

Beispiel 3.4

Betrachten wir die in Abbildung 3.16 angegebene Definition des bereits bekannten algebraischen Datentyps `Baum`.

```
ALG_DEF
   Baum = Knoten(Baum,Baum,BOOLEAN)
        | Blatt(INTEGER);
END_ALG_DEF
```

Abb. 3.16: Algebraischer Datentyp `Baum`.

Aus dieser Definition erhalten wir die folgende Menge der Konstruktoren.

$$\Delta = \{\texttt{Knoten}^{(Baum\ Baum\ bool, Baum)}, \texttt{Blatt}^{(int, Baum)}\} \qquad \square$$

Für den weiteren Abschnitt sei S eine Basis–Sortenmenge, Δ eine endliche $(S^* \times \hat{S})$–sortierte Menge von Konstruktorsymbolen und *Var* eine S–sortierte Menge von Variablensymbolen. Diese Mengen seien beliebig, aber fest.

Die tree transducer Funktionen bilden eine $(\hat{S}S^* \times S^+)$–sortierte Menge. Die Menge, aus der der Eingabetyp einer tree transducer Funktion genommen werden kann, ist $\hat{S}S^*$, da der Typ des ersten Arguments einer tree transducer Funktion immer ein algebraischer Datentyp sein muß.

In NoName können mehrere verschiedene tree transducer Funktionen in einer Spezifikation zusammengefaßt werden, wenn sie als zusammengehörig gekennzeichnet werden sollen und der Typ des ersten Arguments bei allen Funktionen gleich ist. Diese Eigenschaft der Zusammengehörigkeit wird in der formalisierten Version von NoName* durch eine tree transducer Nummern–Funktion tr_nr erreicht. Alle tree transducer Funktionen, die in derselben Spezifikation angegeben sind, werden durch tr_nr auf dieselbe Zahl abgebildet.

Definition 3.5 (tree transducer Nummern–Funktion)

Sei S eine Basis–Sortenmenge. Seien F_{syn} und F_{inh} disjunkte $(\hat{S}S^* \times S^+)$–sortierte Mengen von Funktionssymbolen. Mit F_{tr} werde $F_{syn} \cup F_{inh}$ bezeichnet. Eine Funktion $tr\_nr : F_{tr} \rightarrow \mathbb{N}$ heißt *tree transducer Nummern–Funktion über S, F_{syn} und F_{inh}*, wenn für alle $f_1 \in F_{tr}^{(s_0 s_1 \ldots s_n, \sigma_1 \ldots \sigma_m)}$ und $f_2 \in F_{tr}^{(t_0 t_1 \ldots t_\nu, \theta_1 \ldots \theta_\mu)}$ mit $n, \nu \geq 0$ sowie $m, \mu \geq 1$ die Aussage

$$tr\_nr\,(f_1) = tr\_nr\,(f_2) \Rightarrow s_0 = t_0$$

erfüllt ist. $\square$

Wenn sich eine tree transducer Funktion f beim Aufruf einer externen Funktion auf das aktuelle Rekursionsargument beziehen möchte, so geschieht dies durch Verwendung von

$id(\langle z \rangle)$ bzw. $id(\langle z \rangle.k)$. Der Grund hierfür liegt, wie wir bereits diskutiert hatten, darin, daß $\langle z \rangle$ und $\langle z \rangle.k$ Pfadangaben und keine Teilbäume bezeichnen. Der Typ eines solchen id–Ausdrucks ist natürlich der Typ des damit bestimmten Teilbaums.

Definition 3.6 (ID–Terme)

Sei $\delta \in \Delta \cup \{root\}$ ein Symbol und $\theta \in \hat{S}$ eine Sorte. Die Menge der *ID–Terme über S, Δ, θ und δ*, bezeichnet mit $ID(S, \Delta, \theta, \delta)$, ist die kleinste S–sortierte Menge E, für die gilt:

(i) falls $\delta \in \Delta$ mit $input(d) = t_1 \dots t_\nu$, $\nu \geq 0$, $result(\delta) = \theta$ und $t_1, \dots, t_\nu \in S$ beliebig, dann

- $id(\langle z \rangle) \in E^{(\theta)}$
- für $k \in [\nu]$ gilt $id(\langle z \rangle.k) \in E^{(t_k)}$

(ii) falls $\delta = root$: $id(\langle z \rangle.1) \in E^{(\theta)}$ □

Der Fall (ii) in der Definition 3.6 behandelt den Spezialfall des Wurzelsymbols *root*, das bei mats notwendig ist. In der hier verwendeten Weise ist das Wurzelsymbol polymorph, da es für alle vorkommenden mats ungeachtet des Typs des Rekursionsarguments verwendet wird. Deshalb muß dieser Typ im Falle des Wurzelsymbols der Definition mitgegeben werden. In Definition 3.6 erfolgt dies durch die Variable θ.

In der Definition 3.7 fassen wir die Menge der zulässigen Inxifoperatoren zu einer Menge zusammen, um diese in der nachfolgenden Definition einheitlich ansprechen zu können.

Definition 3.7 (Infixoperatoren)

Die Menge der *Infixoperatoren* ist die Menge {`+`, `-`, `*`, `/`, `div`, `and`, `or`, `<`, `>`, `>=`, `<=`, `=`, `<>`} und wird mit OP bezeichnet. □

3.3.1 Aufbau eines Ausdrucks

Mit den vorangegangenen Definitionen haben wir alle Begriffe zusammen, die wir benötigen, um einen korrekt typisierten Ausdruck zu definieren.

Definition 3.8 (Ausdruck)

Sei F eine $(S^* \times S^+)$–sortierte Menge von Funktionssymbolen, und seien F_{syn} und F_{inh} disjunkte $(\hat{S}S^* \times S^+)$–sortierte Mengen von synthetischen bzw. inheriten Funktionssymbolen.

Außerdem sei $\theta \in \hat{S}$ eine ausgezeichnete Sorte und $\delta \in \Delta^{(t_1 \dots t_\nu, \theta)}$ ein ausgezeichnetes Konstruktorsymbol, wobei $\nu \geq 0$ und $t_1, \dots, t_\nu \in S$ gilt.

Darüber hinaus sei ID eine Teilmenge von $ID(S, \Delta, \theta, \delta)$, nr eine natürliche Zahl und tr_nr eine tree transducer Nummern-Funktion über S, F_{syn} und F_{inh}.

Die Menge der *Ausdrücke (Expressions) über S, F, F_{syn}, F_{inh}, Δ, Var, θ, δ, ID, nr und tr_nr*, bezeichnet mit $EXPR(S, F, F_{syn}, F_{inh}, \Delta, Var, \theta, \delta, ID, nr, tr\_nr)$, ist die kleinste S^+–sortierte Menge E, für die folgende Bedingungen gelten, wobei $s, \sigma \in S$ und für $i \in \mathbb{N}_0$ die Aussage $s_i, \sigma_i \in S$ gilt.

(i) $\mathbb{N}_0 \subseteq E^{(card)}$

(ii) $\mathbb{Z} \subseteq E^{(int)}$

(iii) $\mathbb{R} \subseteq E^{(real)}$

(iv) $\{$ `TRUE`, `FALSE` $\} \subseteq E^{(bool)}$

(v) $CH \subseteq E^{(char)}$

(vi) wenn $x \in Var^{(\sigma)}$, dann $x \in E^{(\sigma)}$

(vii) wenn $d \in \Delta^{(\varepsilon,\sigma)}$, dann $d \in E^{(\sigma)}$

(viii) wenn $id \in ID^{(\sigma)}$, dann $id \in E^{(\sigma)}$

(ix) wenn $d \in \Delta^{(s_1 \dots s_n,\sigma)}$ mit $n \geq 1$, $k \geq 1$ und $e_i \in E^{(s_{i1} \dots s_{ir_i})}$ für $i \in [k]$, wobei $r_i \geq 1$ und

$$\underbrace{s_{11}s_{12}\dots s_{1r_1}}\,\underbrace{s_{21}s_{22}\dots s_{2k_2}}\,\cdots\,\underbrace{s_{k1}s_{k2}\dots s_{kr_k}} = s_1 \dots s_n$$

gilt, dann liegt $d(e_1, \dots, e_k)$ in $E^{(\sigma)}$

(x) wenn $f \in F^{(s_1 \dots s_n, \sigma_1 \dots \sigma_m)}$ mit $n \geq 0$, $m \geq 1$, $k \geq 0$ und $e_i \in E^{(s_{i1} \dots s_{ir_i})}$ für $i \in [k]$, wobei $r_i \geq 1$ und

$$\underbrace{s_{11}s_{12}\dots s_{1r_1}}\,\underbrace{s_{21}s_{22}\dots s_{2k_2}}\,\cdots\,\underbrace{s_{k1}s_{k2}\dots s_{kr_k}} = s_1 \dots s_n$$

gilt, dann liegt $f(e_1, \dots, e_k)$ in $E^{(\sigma_1 \dots \sigma_m)}$

(xi) Sei $F_{tr} = F_{syn} \cup F_{inh}$, $f \in F_{tr}^{(s_0 s_1 \dots s_n, \sigma_1 \dots \sigma_m)}$ mit $s_0 = \theta$, $n \geq 0$, $m \geq 1$, $tr\_nr\,(f) = nr$, $k \geq 0$ und $e_i \in E^{(s_{i1} \dots s_{ir_i})}$ für $i \in [k]$, wobei $r_i \geq 1$ und

$$\underbrace{s_{11}s_{12}\dots s_{1r_1}}\,\underbrace{s_{21}s_{22}\dots s_{2k_2}}\,\cdots\,\underbrace{s_{k1}s_{k2}\dots s_{kr_k}} = s_1 \dots s_n$$

gilt. Wenn

- $f \in F_{syn}$ und $t_k = \theta$ für ein $k \in [\nu]$ gilt, dann liegt $f(\langle z \rangle.k, e_1, \dots, e_k)$ in $E^{(\sigma_1 \dots \sigma_m)}$
- $f \in F_{inh}$, dann liegt $f(\langle z \rangle, e_1, \dots, e_k)$ in $E^{(\sigma_1 \dots \sigma_m)}$.

(xii) für $e_1, e_2 \in E^{(s)}$ liegt $(e_1\ op\ e_2)$ in $E^{(\sigma)}$, wobei $op \in OP$, falls eine der folgenden Bedingungen erfüllt ist

- ($\sigma = s = int$ oder $\sigma = s = card$) und $op \in \{$`+`, `-` , `*`, `div`$\}$
- $s \in \{int,\ card\}$, $\sigma = real$ und $op =$ `/`
- $\sigma = s = real$ und $op \in \{$`+`, `-` , `*`, `/`$\}$
- $s \in \{int,\ card,\ real\}$, $\sigma = bool$ und $op \in \{$`<`, `>`, `<=`, `>=`$\}$
- $s \in S$, $\sigma = bool$ und $op \in \{$ `=`, `<>`$\}$
- $\sigma = s = bool$ und $op \in \{$`and`, `or`$\}$ □

Diese doch etwas umfangreiche Definition bedarf nun einiger Erläuterungen:

Die Zahl nr sowie die tree transducer Funktion tr_nr spielen nur für den Punkt (xi) eine Rolle und sollen erst dort erläutert werden.

Die Punkte (i) bis (v) behandeln die möglichen Konstanten. Das Problem der Mehrdeutigkeit der positiven ganzen Zahlen, das sich hierbei ergibt, wird in diesem Abschnitt gelöst, indem jede Zahl n, die aus der Menge $\mathbb{Z}$ der ganzen Zahlen genommen ist, unterstrichen dargestellt wird. So hat beispielsweise 1 den Typ $card$ und $\underline{1}$ den Typ int. Zahlen aus der Menge der reellen Zahlen werden immer in Gleitpunktdarstellung notiert.

Der Punkt (vi) behandelt Variablensymbole. Bei nullstelligen Konstruktoren (vii) ist zu beachten, daß hinter diesen, im Gegensatz zu nullstelligen Funktionssymbolen, kein Klammerpaar steht. Punkt (viii) behandelt die in Definition 3.6 eingeführten ID–Terme. Ab Punkt (ix) beginnen die induktiv aufgebauten Ausdrücke: (ix) behandelt das Zusammenfassen mehrerer Ausdrücke durch ein Konstruktorsymbol und (x) das Zusammenfassen durch ein Funktionssymbol. Im folgenden Beispiel 3.9 wollen wir dies verdeutlichen.

Beispiel 3.9 (Ausdruck)

Sei $d \in \Delta^{(int\ bool\ int\ real, tree)}$ ein Konstruktorsymbol, $tree \in \hat{S}$ ein benutzerdefinierter algebraischer Datentyp und $x \in Var^{(int)}$ eine Variable des Typs int. Außerdem seien $f \in F^{(int, real)}$ sowie $g \in F^{(int, bool\ int)}$ Funktionssymbole.

Dann sind folgende fünf Terme

$$17,\ x,\ f(\underline{234}),\ g\,(x),\ f(x+\underline{3})$$

Ausdrücke. Der Term $d\,(x, g\,(\underline{2}), f(x))$ ist ebenfalls ein Ausdruck, da die Typen von x, $g\,(\underline{2})$ und $f(x)$ aneinandergereiht den Typ $input(d)$ bilden. □

In (xi) wird dann der Fall behandelt, daß eine mat Funktion f von einer anderen mat Funktion f' aus demselben Spezifikationsblock aufgerufen wird. Der Ausdruck, den wir definieren, soll Bestandteil der rechten Seite einer Gleichung für $f'(\langle z\ :\ \delta\rangle, \ldots)$ sein. Da $result(\delta) = \theta$, ist also θ der Typ des Rekursionsarguments von f'. Außerdem sei $tr\_nr\,(f') = nr$. Durch die Bedingung $tr\_nr\,(f) = nr$ wird nun sichergestellt, daß f und f' im selben Spezifikationsblock liegen.

Betrachten wir zuerst den Fall, daß f eine synthetische Funktion ist. In diesem Fall wird durch die Forderung $t_k = \theta$ für ein $k \in [\nu]$ gewährleistet, daß der Teilbaum, in den abgestiegen werden soll, denselben Typ wie auch das Rekursionsargument hat.

Für den Fall, daß f eine inherite Funktion ist, haben wir es einfacher. Durch Emporsteigen im Eingabebaum können wir in keine unzulässigen Teilbäume gelangen.

Der Punkt (xii) schließlich behandelt Infixausdrücke, wobei hier gewisse Restriktionen gelten. So ist beispielsweise ein Vergleich auf größer nur auf Zahlen desselben Typs erklärt, und jeder Vergleich liefert als Ergebnis einen Wahrheitswert.

Nachdem wir nun allgemeine Ausdrücke definiert haben, betrachten wir im nächsten Abschnitt den Aufbau einer rechten Seite einer Funktionsgleichung.

3.3.2 Aufbau der rechten Seite einer Funktionsgleichung

Im folgenden wollen wir, ausgehend vom Aufbau allgemeiner Ausdrücke, wie sie im vorigen Abschnitt eingeführt wurden, den Aufbau einer rechten Seite einer Funktionsgleichung definieren.

Betrachten wir jetzt die rechte Seite einer Funktionsgleichung einer gewöhnlichen Funktion. Dort sind freilich nicht alle Ausdrücke, die wir gerade definiert haben, zugelassen. So kann dort beispielsweise kein Funktionsaufruf der Form $f(\langle z \rangle, \ldots)$ stehen, da dies nur bei tree transducern Sinn macht. Es sind also, bezogen auf die Definition 3.8, gewisse Einschränkungen zu treffen. Andererseits ist es mittels der Ausdrücke aus Definition 3.8 nicht möglich, ein Tupel von Konstanten als Funktionsergebnis zu erzeugen.

Diese beiden Überlegungen führen uns zur nachfolgenden Definition eines Funktionsausdrucks und zur Definition der rechten Seite einer Funktionsgleichung.

Definition 3.10 (Funktionsausdruck, rechte Seite einer Funktionsgleichung)
Sei F eine $(S^* \times S^+)$–sortierte Menge von Funktionssymbolen.

- Die Menge der *Funktionsausdrücke über den Mengen S, F, Δ und Var*, bezeichnet mit $F\text{–}EXPR\,(S, F, \Delta, Var)$, ist die Menge
$$\bigcup_{\sigma \in S, d \in \Delta} EXPR\,(S, F, \emptyset, \emptyset, \Delta, Var, \sigma, d, \emptyset, 1, tr\_nr_1) \text{ mit } tr\_nr_1\,(x) = 1 \text{ für alle } x.$$
- Die Menge der *rechten Seiten von Funktionsgleichungen über S, F, Δ und Var*, bezeichnet mit $RS\_F(S, F, \Delta, Var)$, ist die kleinste S^+–sortierte Menge E, für die gilt:
  (i) wenn $e \in F\text{–}EXPR\,(S, F, \Delta, Var)^{(s_1 \ldots s_n)}$ mit $n \geq 1$, wobei $s_1, \ldots, s_n \in S$, dann $e \in E^{(s_1 \ldots s_n)}$
  (ii) Sei $k \geq 1$. Wenn für alle $i \in [k]$ gilt, daß $e_i \in F\text{–}EXPR\,(S, F, \Delta, Var)^{(s_{i1} \ldots s_{ir_i})}$, wobei $r_i \geq 1$ und $s_{ij} \in S$, dann liegt $(e_1, \ldots, e_k)$ in $E^{(s_{11} \ldots s_{1r_1} s_{21} \ldots s_{2r_2} \ldots s_{k1} \ldots s_{kr_k})}$

□

Die gewünschten Einschränkungen haben wir erreicht, indem wir die Mengen F_{syn}, F_{inh} und ID im Term $EXPR\,(\ldots)$ gleich der leeren Menge gesetzt haben. Der Punkt (ii) ermöglicht es uns, auch Tupel als Funktionsergebnisse anzugeben.

Beispiel 3.11 (rechte Seite einer Funktionsgleichung)
Betrachten wir nochmals die Symbole S, x, f und g aus Beispiel 3.9. Eine mögliche rechte Seite einer Funktionsgleichung wäre der Term

$$(\texttt{TRUE}, \underline{89} + x).$$

Diese rechte Seite könnte beispielsweise der Funktion g zugeordnet sein. □

Die Begriffe des Funktionsausdrucks und der rechten Seite einer Funktionsgleichung führen wir in analoger Weise nun auch für tree transducer ein.

Definition 3.12 (tree transducer Ausdruck,
rechte Seite einer tree transducer Gleichung)

Sei F eine $(S^* \times S^+)$–sortierte Menge von Funktionssymbolen, und seien F_{syn} und F_{inh} disjunkte $(\hat{S}S^* \times S^+)$–sortierte Mengen von synthetischen bzw. inheriten Funktionssymbolen.

Außerdem sei $\theta \in \hat{S}$ eine ausgezeichnete Sorte und $\delta \in \Delta^{(t_1 \dots t_\nu, \theta)}$ ein ausgezeichnetes Konstruktorsymbol, wobei $\nu \geq 0$ und $t_1, \dots, t_\nu \in S$ beliebig. Mit ID sei die Menge $ID(S, \Delta, \theta, \delta)$ bezeichnet.

Darüber hinaus sei tr_nr eine tree transducer Nummern-Funktion über S, F_{syn} und F_{inh} und $nr \in I\!N$ sei eine natürliche Zahl.

- Die Menge der *tree transducer Ausdrücke über S, F, F_{syn}, F_{inh}, Δ, Var, θ, δ, nr und tr_nr*, bezeichnet mit $TR\text{–}EXPR\,(S, F, F_{syn}, F_{inh}, \Delta, Var, \theta, \delta, nr, tr\_nr)$, ist die Menge $EXPR\,(S, F, F_{syn}, F_{inh}, \Delta, Var, \theta, \delta, ID, nr, tr\_nr)$
- Die Menge der *rechten Seiten von tree transducer Gleichungen über S, F, F_{syn}, F_{inh}, Δ, Var, θ, δ, nr und tr_nr*, bezeichnet mit $RS\_TR\,(S, F, F_{syn}, F_{inh}, \Delta, Var, \theta, \delta, nr, tr\_nr)$, ist die kleinste Menge S^+–sortierte Menge E, für die gilt:
  - (i) wenn $e \in TR\text{–}EXPR\,(S, F, F_{syn}, F_{inh}, \Delta, Var, \theta, \delta, ID, nr, tr\_nr)^{(s_1 \dots s_n)}$ mit $n \geq 1$ und $s_1, \dots, s_n \in S$, dann $e \in E^{(s_1 \dots s_n)}$
  - (ii) Sei $k \geq 1$. Wenn für alle $i \in [k]$ gilt, daß $e_i \in TR\text{–}EXPR\,(S, F, F_{syn}, F_{inh}, \Delta, Var, \theta, \delta, ID, nr, tr\_nr)^{(s_{i1} \dots s_{ir_i})}$, wobei $r_i \geq 1$ und $s_{ij} \in S$, dann liegt $(e_1, \dots, e_k)$ in $E^{(s_{11} \dots s_{1r_1} s_{21} \dots s_{2k_2} \dots s_{k1} \dots s_{kr_k})}$ □

Da nun, anders als in Definition 3.10, die Mengen F_{syn}, F_{inh} und ID im Term $EXPR\,(\dots)$ nicht leer sind, können die so erzeugten Ausdrücke alle Strukturen enthalten, die in tree transducern erlaubt sind.

Beispiel 3.13 (rechte Seite einer tree transducer Gleichung)

Sei $Baum \in S$ ein benutzerdefinierter algebraischer Datentyp. Die Menge Δ der Konstruktoren sei $\{\mathtt{Knoten}^{(Baum\ Baum\ bool, Baum)}, \mathtt{Blatt}^{(int, Baum)}\}$ und die Menge der inheriten Funktionssymbole sei $F_{inh} = \{f_h^{(Baum\ int, int\ bool)}\}$. Der Wert der tree transducer–Nummern Funktion $tr\_nr\,(f_h)$ sei 1. Dann wären die folgenden beiden Terme rechte Seiten einer tree transducer Gleichung ϕ, wobei wir annehmen wollen, daß die linke Seite von ϕ die Form $f_h(\langle z : \mathtt{Knoten} \rangle, n)$ hat.

$$(f_h(\langle z \rangle, \underline{17}), \mathtt{TRUE}) \qquad (\underline{23} + f_h(id\,(\langle z \rangle.2), \underline{2}), \mathtt{FALSE})$$

Eine Verwendung von $id(\langle z \rangle.3)$ anstelle von $id(\langle z \rangle.2)$ im zweiten Term wäre aber unzulässig gewesen, da der dritte Nachfolger eines mit `Knoten` beschrifteten Knotens nicht den Typ $Baum$ hat. □

Mit diesem Beispiel ist nun die Beschreibung des Aufbaus einer rechten Seite einer Funktionsgleichung abgeschlossen. Im nachfolgenden Abschnitt wenden wir uns dem Aufbau einer linken Seite einer Funktionsgleichung zu.

3.3.3 Aufbau der linken Seite einer Funktionsgleichung

Nachdem wir uns bisher mit der Struktur der rechten Seite einer Funktionsgleichung beschäftigt haben, betrachten wir nun die linke Seite einer Funktionsgleichung. Bei tree transducer Funktionen und Funktionen, die via IF definiert sind, dürfen auf der linken Seite in den Parameterpositionen nur Variablen oder, im Falle der ersten Parameterposition einer tree transducer Funktion, Strukturen der Form $\langle z : d \rangle$ auftreten. Ist eine Funktion aber mittels Pattern Matching definiert, so befinden sich auf den Parameterpositionen der linken Seite Muster. Ein Muster ist entweder eine Variable, eine Konstante oder ein gemischter Ausdruck. Ein solcher gemischter Ausdruck kann entweder die Form $x + n$ mit $n \in \mathbb{N}$ oder die Form $d\,(x_1, \ldots, x_k)$ haben. Dabei ist natürlich zu beachten, daß Konstanten oder gemischte Ausdrücke nur auf Parameterpositionen erlaubt sind, deren Typ entweder *card*, *bool* oder ein benutzerdefinierter algebraischer Datentyp ist. Die folgende Definition 3.14 formalisiert diese Eigenschaften.

Definition 3.14 (Muster)

Die Menge der *Muster (Pattern) über S, Δ und Var*, bezeichnet mit $PAT(S, \Delta, Var)$, ist die kleinste S–sortierte Menge P, die folgende Elemente enthält.

(i) $\mathbb{N}_0 \subseteq P^{(card)}$

(ii) `TRUE`, `FALSE` $\in P^{(bool)}$

(iii) wenn $x \in Var^{(\sigma)}$ für $\sigma \in S$, dann $x \in P^{(\sigma)}$

(iv) wenn $d \in \Delta^{(\varepsilon,\sigma)}$ für $\sigma \in S$, dann $d \in P^{(\sigma)}$

(v) wenn $x \in Var^{(card)}$, dann ist $x + n$ für $n \in \mathbb{N}$ in $P^{(card)}$

(vi) wenn $d \in \Delta^{(s_1 \ldots s_n,\sigma)}$ mit $n \geq 1$, $x_i \in Var^{(s_i)}$ für $i \in [n]$ und $\sigma, s_1, \ldots, s_n \in S$, dann liegt $d(x_1, \ldots, x_n)$ in $P^{(\sigma)}$, wobei alle x_i mit $i \in [n]$ paarweise verschieden sein müssen. □

Betrachten wir nun eine synthetische mat Funktion f des Typs $(s_0 s_1 \ldots s_n, \sigma_1 \ldots \sigma_m)$, wobei $s_0 \in \hat{S}$ ein benutzerdefinierter algebraischer Datentyp ist. In der Menge der Funktionsgleichungen muß nun für jeden Konstruktor d des Typs s_0 eine Regel

$$f\,(\langle z : d\,\rangle, \ldots) = \ldots$$

existieren. Um die Menge dieser Konstruktoren zu bestimmen, verwenden wir die folgende Hilfsdefinition.

Definition 3.15 (Menge der Konstruktoren mit Resultattyp θ, $\Delta[\theta]_S$)

Sei $\theta \in S$ eine Sorte. Die *Menge der Konstruktoren mit Resultattyp θ bezogen auf S und Δ*, bezeichnet mit $\Delta[\theta]_S$, ist die Menge $D \subseteq \Delta$, für die die Aussage $d \in D \Leftrightarrow result(d) = \theta$ gilt. □

Nun haben wir alle nötigen Begriffe beisammen, um die formalisierte Fassung eines strukturell korrekten NoName*–Programms zu definieren.

3.3.4 Formalisierte Fassung eines NoName*–Programms

Mit Hilfe der Begriffe, die wir in den vorigen Abschnitten definiert haben, ist es uns nun möglich, ein strukturell korrektes NoName*–Programm zu definieren. Wir wollen an dieser Stelle daran erinnern, daß das in der folgenden Definition 3.16 enthaltene Symbol *root* das Wurzelsymbol ist, das in Zusammenhang mit mats benötigt wird.

Definition 3.16 (strukturell korrektes NoName*–Programm)
Ein strukturell korrektes *NoName*–Programm* ist ein Tupel $(S, \mathcal{F}, eval, tr\_nr, \Delta, Var, anfr, \mathcal{R})$ mit

- Basis–Sortenmenge S,
- Funktionssystem $\mathcal{F} = (F_{if}, F_{pat}, F_{syn}, F_{inh}, F)$, wobei $F = F_{if} \cup F_{pat} \cup F_{syn} \cup F_{inh}$ die Menge aller Funktionssymbole ist und die Mengen F_{if}, F_{pat}, F_{syn} und F_{inh} paarweise disjunkt sind. Dabei seien F_{if} und F_{pat} $(S^* \times S^+)$–sortierte Mengen und F_{syn} sowie F_{inh} $(\hat{S}S^* \times S^+)$–sortierte Mengen.
- $eval$ ist eine Funktion $F \to \{CbV, CbN\}$,
- tr_nr ist eine tree transducer Nummern-Funktion über S, F_{syn} und F_{inh},
- Δ ist eine $(S^* \times \hat{S})$–sortierte Menge von Konstruktorsymbolen,
- Var ist eine S–sortierte Menge von Variablensymbolen,
- $anfr \in F\text{–}EXPR\,(S, F \backslash F_{inh}, \Delta, Var)^{(s_1 \dots s_n)}$ mit $n \geq 1$, $s_1, \dots, s_n \in S$ ist ein Ausdruck, der die Anfrage an das NoName*–Programm darstellt,
- $\mathcal{R}$ ist eine Menge von Funktionsgleichungen mit
  - ◇ $\mathcal{R}$ enthält für jedes $f \in F_{if}$ genau eine Regel der Form

$$\begin{aligned} f(x_1, \dots, x_n) &= rs_1, b_1, \\ &= rs_2, b_2, \\ &\vdots \\ &= rs_l, b_l \end{aligned}$$

    wobei $l \geq 1$, $f \in F_{if}^{(s_1 \dots s_n, \sigma_1 \dots \sigma_m)}$ mit $n \geq 0$, $m \geq 1$, $s_1, \dots, s_n, \sigma_1, \dots, \sigma_m \in S$, für alle $i \in [n]$ sei $x_i \in Var^{(s_i)}$ und $rs_1, \dots, rs_l \in RS\_F(S, F \backslash F_{inh}, \Delta, V)^{(\sigma_1 \dots \sigma_m)}$, wobei $V = \{x_1, \dots, x_n\}$.
    Außerdem müssen die x_i für alle $i \in [n]$ paarweise verschieden sein.
    Für die $b_1, \dots, b_l$ gilt: b_i ist Element der Menge

$$\{\texttt{IF}\ e \mid e \in F\text{–}EXPR\,(S, F \backslash F_{inh}, \Delta, V)^{(bool)}\} \cup \{\texttt{OTHERWISE}\},$$

    und die b_i sind paarweise disjukt.

- $\diamond$ $\mathcal{R}$ enthält für jedes $f \in F_{pat}$ mindestens eine Regel der Form

$$f(p_1, \dots, p_n) = rs$$

wobei $f \in F_{pat}^{(s_1 \dots s_n, \sigma_1 \dots \sigma_m)}$ mit $n \geq 0$, $m \geq 1$, $s_1, \dots, s_n, \sigma_1, \dots, \sigma_m \in S$, und für alle $i \in [n]$ sei $p_i \in PAT(S, \Delta, Var)^{(s_i)}$. Die rechte Seite der Funktionsgleichung rs ist aus $RS\_F(S, F \backslash F_{inh}, \Delta, V)^{(\sigma_1 \dots \sigma_m)}$, wobei $V = \bigcup_{1 \leq i \leq n} Var(p_i)$.

Mit $Var(p_i)$ werden dabei die Variablensymbole bezeichnet, die im Muster p_i vorkommen. Die einzelnen Mengen $Var(p_i)$ müssen paarweise disjunkt sein.

- $\diamond$ $\mathcal{R}$ enthält für jedes $f \in F_{syn}^{(s_0 s_1 \dots s_n, \sigma_1 \dots \sigma_m)}$ mit $n \geq 0$, $m \geq 1$ sowie $s_i, \sigma_j \in S$ und jedes $d \in \Delta[s_0]_S$ genau eine Regel der Form

$$f(\langle z : d \rangle, x_1, \dots, x_n) = trs,$$

wobei für alle $1 \leq i \leq n$ die $x_i \in Var^{s_i}$ paarweise verschieden sind.
Die rechte Seite der tree transducer Gleichung trs ist ein Element der Menge $RS\_TR(S, F \backslash F_{inh}, F_{syn}, F_{inh}, \Delta, V, s_0, d, tr\_nr(f), tr\_nr)^{(\sigma_1 \dots \sigma_m)}$,
wobei $V = \{x_1, \dots, x_n\}$.
Außerdem enthält $\mathcal{R}$ für jedes $f \in F_{syn}$ genau eine Regel der Form

$$f(\langle z : root \rangle, x_1, \dots, x_n) = trs,$$

wobei dieselben Bedingungen gelten, außer daß trs in diesem Fall aus der Menge $RS\_TR(S, F \backslash F_{inh}, F_{syn}, \emptyset, \Delta, V, s_0, root, tr\_nr(f), tr\_nr)^{(\sigma_1 \dots \sigma_m)}$ stammt.

- $\diamond$ $\mathcal{R}$ enthält für jedes $f \in F_{inh}^{(s_0 s_1 \dots s_n, \sigma_1 \dots \sigma_m)}$ mit $n \geq 0$, $m \geq 1$ sowie $s_i, \sigma_j \in S$, jedes $d \in \Delta[s_0]_S$ mit $input(d) = t_1 \dots t_\nu$ und jedes $k \in [\nu]$ genau eine Regel der Form

$$f(\langle z : d \rangle.k, x_1, \dots, x_n) = trs.$$

Für alle $1 \leq i \leq n$ seien die Variablensymbole x_i paarweise disjunkt und aus der Menge $Var^{(s_i)}$ genommen.
Für die rechte Seite der tree transducer Gleichung trs gelte, daß sie ein Element der Menge $RS\_TR(S, F \backslash F_{inh}, F_{syn}, F_{inh}, \Delta, V, s_0, d, tr\_nr(f), tr\_nr)^{(\sigma_1 \dots \sigma_m)}$ ist, wobei $V = \{x_1, \dots, x_n\}$.
Außerdem enthält $\mathcal{R}$ für jedes $f \in F_{inh}$ genau eine Regel der Form

$$f(\langle z : root \rangle.1, x_2, \dots, x_n) = trs,$$

wobei dieselben Bedingungen gelten, außer daß trs in diesem Fall aus der Menge $RS\_TR(S, F \backslash F_{inh}, F_{syn}, \emptyset, \Delta, V, s_0, root, tr\_nr(f), tr\_nr)^{(\sigma_1 \dots \sigma_m)}$ stammt.

Außerdem müssen die Mengen S, F, Δ, *Var* und *Rule* paarweise disjunkt sein. $\square$

Auch hier sind noch einige Punkte ein wenig erklärungsbedürftig: Die Funktion *eval* gibt für jede Funktion an, ob diese nach der Auswertungsstrategie Call–by–Name (*CbN*) oder Call–by–Value (*CbV*) ausgewertet werden soll. Der Grund, warum die Anfrage *anfr* ein Bestandteil eines NoName*–Programms sein sollte, wurde in Abschnitt 3.2 besprochen.

Die Forderung nach Linkslinearität wird durch die Bedingung der paarweise disjunkten Variablensymbole erfüllt.

Bei den rechten Seiten von Funktionsgleichungen ist zu beachten, daß hier inherite Funktionen nicht aufgerufen werden können. Der Aufruf einer inheriten Funktionen f_h ist nur einer tree transducer Funktionen f_s gestattet, und dies auch nur dann, wenn f_s und f_h im selben mat spezifiziert sind, also bzgl. *tr_nr* dieselbe Nummer haben. Diese Restriktion ist notwendig, da eine inherite Funktion zuerst einen Schritt in Richtung Wurzel im Eingabebaum durchführt. Durch einen Aufruf von „außen“ steht sie aber bereits auf der Wurzel, wodurch ein solcher Schritt unmöglich gemacht ist. Die Bedeutung des Wurzelsymbols *root* haben wir bereits bei der Einführung von mats (siehe 3.1, Seite 22) diskutiert.

Beispiel 3.17 (strukturell korrektes NoName*–Programm)

Erinnern wir uns an dieser Stelle an Abbildung 1.1, in der wir das NoName*–Programm `EinlBsp` vorgestellt haben. Das Tupel $P = (S, \mathcal{F}, eval, tr\_nr, \Delta, Var, anfr, \mathcal{R})$ mit

- der Sortenmenge $S = \{card, int, real, bool, char, Baum\}$,
- dem Funktionssystem $\mathcal{F} = (F_{if}, F_{pat}, Fsyn, F_{inh}, F)$, wobei

  $$\begin{array}{lcllcl} F_{if} & = & \{\texttt{gen\_baum}^{(card\ int\ bool, Baum)}\} & F_{pat} & = & \{\texttt{get\_zahl}^{(Baum,int)}\} \\ F_{syn} & = & \{\texttt{f\_down}^{(Baum,int)}\} & F_{inh} & = & \{\texttt{f\_up}^{(Baum\ int,int)}\} \end{array}$$

  sowie $F = F_{if} \cup F_{pat} \cup F_{syn} \cup F_{inh}$,
- der Funktion *eval* mit $eval(\texttt{get\_zahl}) = eval(\texttt{f\_down}) = eval(\texttt{f\_up}) = CbV$ und $eval(\texttt{gen\_baum}) = CbN$,
- der tree transducer–Nummern Funktion *tr_nr* mit $tr\_nr\,(\texttt{f\_down}) = tr\_nr\,(\texttt{f\_up}) = 1$,
- der Menge $\Delta = \{\texttt{Knoten}^{(Baum\ Baum\ bool, Baum)}, \texttt{Blatt}^{(int,Baum)}\}$ von Konstruktorsymbolen,
- der Menge $Var = \{\texttt{count}^{(card)}, \texttt{n}^{(int)}, \texttt{b}^{(bool)}, \texttt{t1}^{(Baum)}, \texttt{t2}^{(Baum)}\}$ von Variablensymbolen,
- der Anfrage $anfr = \texttt{f\_down}\,(\texttt{gen\_baum}\,(3, \underline{2}, \texttt{TRUE}))$,
- der Menge $\mathcal{R}$ von Funktionsgleichungen mit

$$\begin{array}{ll} \mathcal{R} = \{ & \texttt{gen\_baum}\,(\texttt{count}, \texttt{n}, \texttt{b}) \\ & \quad = \texttt{Blatt}\,(\texttt{n}),\ \texttt{IF}\ (\texttt{count} = 0), \\ & \quad = \texttt{Knoten}\,(\texttt{gen\_baum}\,((\texttt{count} - 1), \texttt{n}, \texttt{b}), \texttt{Blatt}\,(\texttt{n}), \texttt{b}),\ \texttt{OTHERWISE} \\ & \texttt{get\_zahl}\,(\texttt{Knoten}\,(\texttt{t1}, \texttt{t2}, \texttt{b})) = \underline{0} \\ & \texttt{get\_zahl}\,(\texttt{Blatt}\,(\texttt{n})) = \texttt{n} \\ & \texttt{f\_down}\,(\langle z : root \rangle) = \texttt{f\_down}\,(\langle z \rangle.1) \\ & \texttt{f\_down}\,(\langle z : \texttt{Knoten} \rangle) = (\texttt{f\_down}\,(\langle z \rangle.1) + \texttt{f\_down}\,(\langle z \rangle.1)) \\ & \texttt{f\_down}\,(\langle z : \texttt{Blatt} \rangle) = \texttt{f\_up}\,(\langle z \rangle, \texttt{get\_zahl}\,(id(\langle z \rangle))\,) \\ & \texttt{f\_up}\,(\langle z : root \rangle.1, \texttt{n}) = \underline{0} \\ & \texttt{f\_up}\,(\langle z : \texttt{Knoten} \rangle.1, \texttt{n}) = (\,\texttt{n} + \texttt{f\_up}\,(\langle z \rangle, \texttt{n})\,) \\ & \texttt{f\_up}\,(\langle z : \texttt{Knoten} \rangle.2, \texttt{n}) = \texttt{f\_up}\,(\langle z \rangle, \texttt{n}) \qquad \} \end{array}$$

ist ein strukturell korrektes NoName*–Programm.

Außerdem ist P die formale Variante des NoName*–Programms `EinlBsp`. □

Die strukturell korrekten NoName*–Programme, die in Definition 3.16 eingeführt worden sind, haben aber noch einen gravierenden Nachteil: Die Forderung nach erschöpfendem und überschneidungsfreiem Pattern Matching ist nicht notwendigerweise erfüllt.

Beispiel 3.18 (nicht erschöpfendes und überschneidungsfreies Pattern Matching)

Gegeben sei die Funktion $f_p \in F_{pat}^{(Baum\ card\ bool,bool)}$, wobei *Baum* derselbe algebraische Datentyp ist, wie er in Beispiel 3.4 eingeführt wurden ist. Die Menge Δ der Konstruktorsymbole sei $\{\mathtt{Knoten}^{(Baum\ Baum\ bool,Baum)}, \mathtt{Blatt}^{(int,Baum)}\}$.

Betrachten wir nun die in Abbildung 3.17 angegebenen Funktionsgleichungen der Funktion f_p, die mittels Pattern Matching definiert ist. Da die rechten Seiten der Funktionsgleichungen hier nicht relevant sind, haben wir sie der Einfachheit halber weggelassen.

$$\begin{array}{l} f_p\,(\mathtt{Knoten}\,(x_1,x_2,x_3),y,\mathtt{TRUE}) = \ldots \\ f_p\,(\mathtt{Knoten}\,(x_1,x_2,x_3),0,b) = \ldots \\ f_p\,(\mathtt{Blatt}\,(x),y,\mathtt{TRUE}) = \ldots \\ f_p\,(\mathtt{Blatt}\,(x),0,\mathtt{FALSE}) = \ldots \end{array}$$

Abb. 3.17: Funktionsgleichungen für f_p.

Dieses Pattern Matching ist weder überschneidungsfrei noch erschöpfend. Die Bedingung der Überschneidungsfreiheit ist verletzt, da im Falle eines Funktionsaufrufs der Form $f_p(\mathtt{Knoten}(\ldots),0,\mathtt{TRUE})$ sowohl die erste als auch die zweite Funktionsgleichung angewandt werden könnte.

Das Pattern Matching ist auch nicht erschöpfend, da es für einen Funktionsaufruf der Form $f_p(\mathtt{Blatt}(\ldots),1,\mathtt{FALSE})$ keine Funktionsgleichung gibt, die Anwendung finden könnte. □

Wir brauchen also einen Test, der uns diese Eigenschaft sichert. Dabei wählen wir eine Variante, die nur theoretisch relevant ist: Wir betrachten alle möglichen Eingabevektoren, also Belegungen der Funktionsparameter einer Funktion f und überprüfen, wieviele Funktionsgleichungen von f auf diesen Eingabevektor passen. Wenn für alle Eingabevektoren immer nur genau eine Funktionsgleichung paßt, dann ist das Testergebnis positiv. Diese Vorgehensweise ist für einen Testalgorithmus freilich untauglich, da unendlich viele Möglichkeiten getestet werden müßten. Eine realisierbare, aber wesentlich aufwendigere Variante findet sich z.B. in [Ern95].

Zuerst definieren wir nun die möglichen Elemente eines Eingabevektors.

Definition 3.19 (Normalterm)

Die *Menge der Normalterme über S und Δ*, bezeichnet mit $VAL(S,\Delta)$, ist die kleinste S–sortierte Menge E, die folgende Eigenschaften erfüllt.

(i) $\mathbb{N}_0 \subseteq E^{(card)}$

(ii) $\mathbb{Z} \subseteq E^{(int)}$

(iii) $\mathbb{R} \subseteq E^{(real)}$

(iv) $\{ \texttt{TRUE}, \texttt{FALSE} \} \subseteq E^{(bool)}$

(v) $CH \subseteq E^{(char)}$

(vi) wenn $d \in \Delta^{(\varepsilon,\sigma)}$ mit $\sigma \in \hat{S}$, dann $d \in E^{(s)}$

(vii) wenn $d \in \Delta^{(s_1 \ldots s_n,\sigma)}$ mit $n \geq 1$, $s_1, \ldots, s_n \in S$, $\sigma \in \hat{S}$ und für alle $i \in [n]$ gilt, daß $e_i \in E^{(s_i)}$, dann liegt $d(e_1, \ldots, e_n)$ in $E^{(\sigma)}$ □

Nun müssen wir festlegen, wann ein Eingabevektor und eine linke Seite zusammenpassen. Wir definieren dazu zuerst die Beziehung: „Ein Wert paßt zu einem Muster“. Dies geschieht in naheliegender Weise. So paßt beispielsweise ein Muster in Form einer Konstanten c nur auf einen Wert c.

Definition 3.20 (passendes Muster)

Sei p ein Muster aus $PAT(S, \Delta, Var)^{(\sigma)}$ mit $\sigma \in S$ und $x \in VAL(S, \Delta)$ ein Normalterm über S und Δ.

Man sagt, *p paßt zu x*, wenn eine der folgenden Bedingungen erfüllt ist.

- $p \in Var^{(\sigma)}$ und $x \in VAL(S, \Delta)^{(\sigma)}$
- $\sigma \in \{card,\ bool\}$ und $p = x$
- $\sigma = card$, $p = v + n$ mit $n \geq 1$, $x \in I\!N$ und $x \geq n$
- $\sigma \in \hat{S}$, $p = d$ mit $d \in \Delta^{(\varepsilon,\sigma)}$ und $x = d$
- $\sigma \in \hat{S}$, $p = d(x_1, \ldots, x_n)$, $d \in \Delta^{(s_1 \ldots s_n,\sigma)}$ mit $s_1, \ldots, s_n \in S$ und $x = d(e_1, \ldots, e_n)$ mit $e_i \in VAL(S, \Delta)^{(s_i)}$ für alle $i \in [n]$. □

Beispiel 3.21 (passendes Muster)

Sei wiederum *Baum* der in Beispiel 3.4 eingeführte algebraische Datentyp und bezeichne Δ die Menge der Konstruktorsymbole $\{\texttt{Knoten}^{(Baum\ Baum\ bool,Baum)}, \texttt{Blatt}^{(int,Baum)}\}$. Die Menge der Variablensymbole sei $Var = \{x_1^{(Baum)}, x_2^{(Baum)}, x_3^{(bool)}, n^{(card)}\}$.

Sei p das Muster $\texttt{Knoten}(x_1, x_2, x_3)$. Ein möglicher zu p passender Normalterm ist dann $\texttt{Knoten}(\texttt{Blatt}(\underline{2}), \texttt{Blatt}(\underline{7}), \texttt{TRUE})$.

Der Normalterm 2.45 paßt aber nicht zu dem Muster $n + 1$, da der Normalterm 2.45 den Typ *real* hat, während der Typ von $n + 1$ die Sorte *card* ist. □

Nun muß noch der bereits erwähnte Test über alle möglichen Eingabevektoren in Bezug auf eine Funktion f formalisiert werden.

Definition 3.22 (erschöpfendes und überschneidungsfreies Pattern Matching)

Sei P ein strukturell korrektes NoName*–Programm $(S, \mathcal{F}, eval, tr\_nr, \Delta, Var, anfr, \mathcal{R})$ mit $\mathcal{F} = (F_{if}, F_{pat}, F_{syn}, F_{inh}, F)$ und f eine Funktion mit $f \in F_{pat}^{(s_1 \ldots s_n,\sigma_1 \ldots \sigma_m)}$, wobei $n \geq 0$, $m \geq 1$ und $s_i, \sigma_j \in S$.

Man sagt, die Funktion f *induziert ein erschöpfendes und überschneidungsfreies Pattern Matching*, wenn die beiden folgenden Bedingungen erfüllt sind.

- Für alle $i \in [n]$ und alle $x = (x_1, \ldots, x_n)$ mit $x_i \in \mathit{VAL}(S, \Delta)^{(s_i)}$ gilt:

  Es gibt mindestens eine Regel $f(p_1, \ldots, p_n) = rs$ in $\mathcal{R}$, so daß für alle $i \in [n]$ gilt: p_i paßt zu x_i (erschöpfendes Pattern Matching)

- Für je zwei Regeln $f(p_1, \ldots, p_n) = rs$ und $f(p'_1, \ldots, p'_n) = rs'$ in $\mathcal{R}$ gibt es kein $x = (x_1, \ldots, x_n)$ mit $x_i \in \mathit{VAL}(S, \Delta)^{(s_i)}$ für alle $i \in [n]$, so daß für alle $i \in [n]$ sowohl p_i als auch p'_i zu x_i paßt (überschneidungsfreies Pattern Matching) □

Beispiel 3.23 (erschöpfendes und überschneidungsfreies Pattern Matching)

Das Pattern Matching, das in Beispiel 3.17 durch die Funktion `get_zahl` induziert wird, ist erschöpfend und überschneidungsfrei. □

Wenn ein strukturell korrektes NoName*–Programm P für alle Funktionen, die mittels Pattern Matching definiert sind, diesen Test besteht, dann können wir es als korrekt bezeichnen.

Definition 3.24 (NoName*–Programm)

Ein strukturell korrektes NoName*–Programm $P = (S, \mathcal{F}, eval, tr\_nr, \Delta, \mathit{Var}, \mathit{anfr}, \mathcal{R})$ mit $\mathcal{F} = (F_{if}, F_{pat}, F_{syn}, F_{inh}, F)$ heißt *(korrektes) NoName*–Programm*, wenn für jede Funktion $f \in F_{pat}$ das durch diese Funktion induzierte Pattern Matching erschöpfend und überschneidungsfrei ist. Die Menge aller (korrekten) NoName*–Programme wird mit $NONAME^*$ bezeichnet. □

Beispiel 3.25 (NoName*–Programm)

Das strukturell korrekte NoName*–Programm P aus Beispiel 3.17 ist auch ein (korrektes) NoName*–Programm, da alle Funktionen $f \in F_{pat}$ ein überschneidungsfreies und erschöpfendes Pattern Matching induzieren. □

Am Anfang dieses Abschnittes haben wir erwähnt, daß zwischen einem formalisierten NoName*–Programm, so wie es gerade eingeführt wurde, und einem auf der zuvor besprochenen NoName–Syntax basierenden Programm kein inhaltlicher Unterschied besteht. Dies bedeutet, daß eine Überführung der beiden Darstellungsformen ineinander möglich sein muß. Da diese Überführung jedoch naheliegend ist — die einzelnen Definitionen wurden unter dem Blickwinkel einer Überführung motiviert — soll diese hier nicht formalisiert werden.

Die Vorstellung der Sprache NoName* ist nun abgeschlossen. Wir haben zuerst die Sprache NoName* an Hand einiger Beispiele eingeführt und dann formalisiert. Dieser Formalismus ermöglicht es uns, auf relativ einfache Art und Weise festzulegen, wann ein NoName*–Programm bzgl. der statischen Semantik korrekt ist. Ein Test auf der Ebene der realen NoName*–Programme ist wesentlich aufwendiger, wenn auch durchführbar (nicht zuletzt wegen der eben erwähnten Überführbarkeit). Der interessierte Leser sei hierzu auf [Ern95] verwiesen, wo ein solcher Test für NoName im Rahmen eines syntaxgesteuerten Editors auf Attributgrammatiken basierend beschrieben wird.

Kapitel 4

Die abstrakte NoName–Maschine

Bevor wir daran gehen, die Übersetzung für NoName* zu definieren, müssen wir zuerst das Ziel der Übersetzung festlegen. Hier soll keine reale Maschine Verwendung finden, sondern eine abstrakte, also selbstdefinierte Maschine. Dies hat den Vorteil, daß wir uns bei der Übersetzung nicht mit Problemen beschäftigen müssen, deren Ursachen in der Architektur der Zielmaschine liegen und nichts mit der eigentlichen Aufgabenstellung der Implementierung zu tun haben. Die Zielmaschine ist aber dennoch so definiert, daß die Nähe zu real existierenden Maschinen gewahrt bleibt.

In diesem Kapitel werden wir nun diese Zielmaschine, eine sogenannte abstrakte NoName–Maschine, kurz NoMach, einführen und beschreiben. Dazu stellen wir deren Aufbau, den Befehlssatz der Maschine und die Wirkungsweise der Maschinenbefehle vor.

Im Abschnitt 4.1 werden wir zuerst den Aufbau einer NoMach erläutern und ihre einzelnen Bestandteile definieren. In Abschnitt 4.2 werden wir dann die einzelnen Maschinenbefehle detailliert beschreiben und in Abschnitt 4.3 angeben, wie die NoMach ein Maschinenprogramm abarbeitet. Mit einem größeren Beispiel endet dann dieses Kapitel.

4.1 Der Aufbau der abstrakten NoName–Maschine

Eine NoMach ist eine Runtime–Stack Maschine, also eine Maschine, deren Kernstück ein Stack ist, auf dem zur Laufzeit alle Berechnungen ausgeführt werden. Neben diesem Stack besitzt die Maschine noch weitere Komponenten, in denen Hilfs– und Verwaltungsinformationen gespeichert werden. Insgesamt besitzt eine NoMach also die folgenden acht Komponenten.

- Berechnungsstack
- Sprungkeller
- Umgebungskeller
- Halde
- Programmzähler
- Kellerrahmen–Zeiger
- Rückgabewert–Register
- Maschinenprogramm

Auf dem Sprungkeller werden die Rücksprungadressen von Unterprogrammen verwaltet. Der Umgebungskeller wird dazu dienen, Verweise auf Umgebungen zu speichern. Auf der Halde werden Objekte veränderlicher Größe abgelegt. Der Programmzähler gibt an, welcher Befehl als nächster ausgeführt werden soll. Für jeden Funktionsaufruf wird später ein sogenannter Kellerrahmen auf dem Berechnungsstack angelegt. Der Kellerrahmen–Zeiger zeigt dann auf den jeweils aktuellen Kellerrahmen. Der Wert im Rückgabewert–Register gibt beim Beenden des Maschinenlaufs den Grund der Terminierung an.

Der Aufbau einer NoMach orientiert sich an der abstrakten Maschine MaMa, die in [WM92] vorgestellt wird. Die Grundgedanken dieser Art von Architektur gehen aber auf die SECD–Maschine von Landin [Lan64, Hen80] zurück. Die SECD–Maschine ist eine abstrakte Maschine, die zur Ausführung von übersetzten Lisp–Programmen konzipiert wurde. Während aber die Halde in der MaMa eine mehr zentrale Rolle spielt, hat sie in der NoMach eher den Charakter einer Hilfsstruktur. Dies liegt daran, daß die Programme, die im Rahmen dieser Arbeit übersetzt werden, keine Funktionen höherer Ordnung enthalten dürfen. Bei der NoMach dient die Halde nur dazu, Strukturen variabler Größe zu speichern.

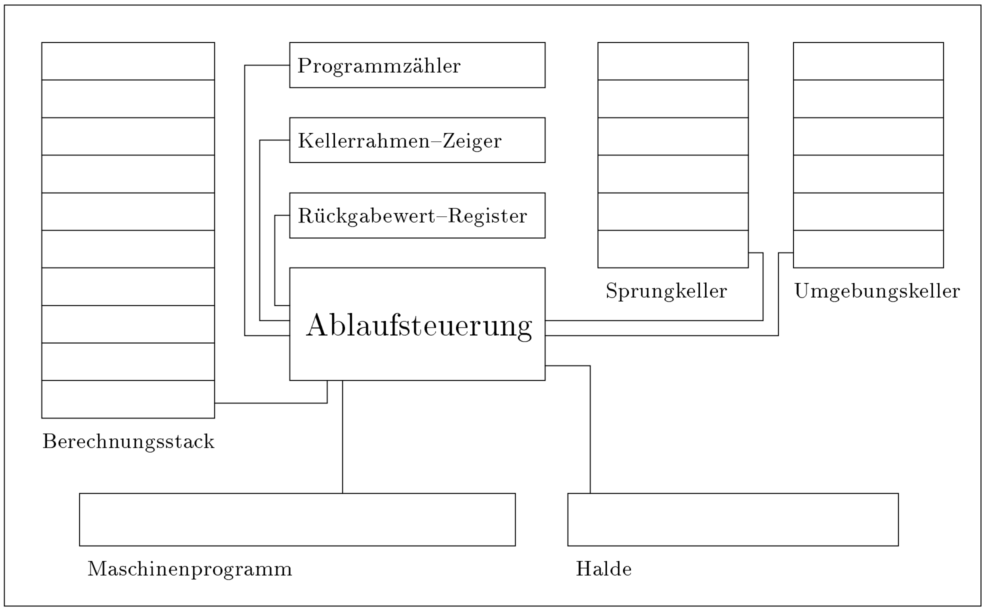


Abb. 4.1: Schematische Darstellung einer abstrakten NoName–Maschine.

In Abbildung 4.1 sind nun diese Komponenten noch einmal schematisch dargestellt. Bei dem Kasten in der Mitte, der mit „Ablaufsteuerung“ beschriftet ist, handelt es sich nicht um eine Komponente der NoMach. Die Ablaufsteuerung ist vielmehr ein übergeordneter Mechanismus, die „treibende Kraft“ der Maschine, die bewirkt, daß sich die Komponenten der NoMach gemäß dem auszuführenden Maschinenbefehl verändern.

In den folgenden Abschnitten werden wir die einzelnen Komponenten einer NoMach detaillierter betrachten und formal definieren.

4.1.1 Berechnungsstack, Halde, Keller und Register

Bei der Beschreibung der einzelnen Komponenten einer NoMach wollen wir uns zuerst mit den Komponenten beschäftigen, die zur Laufzeit verändert werden. Dies sind der

Berechnungsstack, die Halde, die beiden Keller sowie die drei Register.

Auf dem Berechnungsstack werden alle Werte, die im Verlauf einer Berechnung anfallen, abgespeichert. Da die Sprache NoName* verschiedene Arten von Werten kennt, sollten sich diese Arten auch in den möglichen Elementen auf dem Berechnungsstack unterscheiden. Dazu führen wir vier verschiedene Wertelemente ein, nämlich Zahlenwerte, Zeichen, Wahrheitswerte und Konstruktoren. Die Unterscheidung zwischen den verschiedenen Arten von Zahlen (Integer, Cardinal, Real) geben wir jedoch auf.

Die Unterscheidung der verschiedenen Wertelemente wird dadurch erreicht, daß dem eigentlichen Wert eine Markierung, die den Typ des Werts angibt, vorangestellt wird. Die verschiedenen Arten von Wertelementen erhalten also die Markierungen Y_{num}, Y_{char}, Y_{bool} und Y_{cons} für Zahlenwerte, Zeichen, Wahrheitswerte bzw. Konstruktoren.

Es wäre natürlich auch möglich, jede Unterscheidung von Wertelementen aufzuheben und alle Werte numerisch zu codieren. In diesem Fall würden jedoch die durch die Übersetzung erzeugten Befehlssequenzen unleserlicher und die Funktion der Maschinenbefehle wäre weniger deutlich. Der Ansatz der Unterschiedung von verschieden Arten von Stack–Elementen findet sich beispielsweise auch in [FGV91].

Über algebraischen Datentypen können Terme aufgebaut werden. Solche Terme haben die Eigenschaft, daß sie möglicherweise eine beliebige Größe haben können. Da jedes Element des Berechnungsstacks eine feste Größe haben soll, ist es klar, daß sich solche Terme nicht in einem Berechnungsstack–Element speichern lassen.

Wir werden deshalb diese variabel großen Strukturen auf der Halde ablegen und nur die Verweise auf diese Strukturen im Berechnungsstack ablegen. Die Elemente, die einen solchen Verweis repräsentieren, heißen Verweiselemente und sind mit der Markierung P gekennzeichnet. In Abbildung 4.2 ist eine solche Situation wiedergegeben.

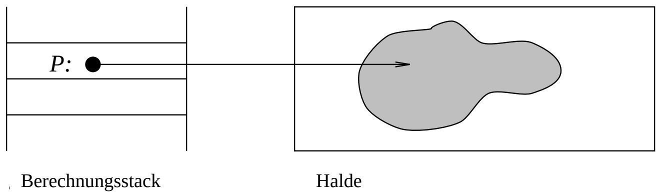


Abb. 4.2: Verweis auf variabel große Struktur.

In NoName* ist es möglich, Funktionen nach der Strategie Call–by–Name auswerten zu lassen. Bezogen auf eine Funktion

$$f(x) = e$$

bedeutet das, daß mit der Auswertung von e begonnen wird, ohne zuvor den Wert des Parameters x ermittelt zu haben. Dieser Wert wird erst bestimmt, wenn er im Laufe der Auswertung von e benötigt wird. Dieser Umstand führt dazu, daß es neben Wert– und Verweiselementen noch eine dritte Art von Berechnungsstack–Elementen geben muß, nämlich Funktionselemente. In einem Funktionselement wird eine Programmadresse adr gespeichert. Wenn die Routine, also die Folge von Maschinenbefehlen, die an der Adresse adr steht, angesprungen wird, wird der Wert des benötigten Parameters bestimmt.

Da die Routine, die an der Stelle adr steht, möglicherweise selbst Parameter benötigt, muß neben der Adresse auch noch ein Verweis fp abgelegt werden, der angibt, wo sich diese Parameterwerte im Berechnungsstack befinden. Funktionselemente besitzen die Markierung

F. In Definition 4.1 findet sich die obige Beschreibung eines Berechnungsstack–Elements formal angegeben. Dabei sei im folgenden Δ eine beliebige, aber feste und endliche Menge von Konstruktorsymbolen. An dieser Stelle wollen wir nochmals daran erinnern, daß wir mit *root* das Wurzelsymbol bezeichnen, welches bei macro attributed tree transducern benötigt wird.

Definition 4.1 (Wert–, Verweis-, Funktionselement, Berechnungsstack–Element)
Die Menge der *Wertelemente über* Δ, bezeichnet mit $Y\_ELEM(\Delta)$, ist die Menge

$$\{(Y_{num} : x) \mid x \in \mathbb{R}\} \cup \{(Y_{bool} : b) \mid b \in \{\texttt{TRUE}, \texttt{FALSE}\}\} \cup \\ \{(Y_{char} : c) \mid c \in CH\} \cup \{(Y_{cons} : d, n) \mid d \in \Delta \cup \{root\}, n \in \mathbb{N}_0\}.$$

Die Menge der *Verweiselemente*, kurz P_ELEM, ist die Menge

$$\{(P : adr) \mid adr \in \mathbb{N}\}.$$

Die Menge der *Funktionselemente*, kurz F_ELEM, ist die Menge

$$\{(F : adr,\ fp) \mid adr,\ fp \in \mathbb{N}\}.$$

Die *Menge der Berechnungsstack–Elemente über* Δ, bezeichnet mit $CST\_ELEM(\Delta)$, ist die Menge $Y\_ELEM(\Delta) \cup P\_ELEM \cup F\_ELEM$. □

Der numerische Wert, der neben dem eigentlichen Konstruktor in einem Wertelement mit Markierung Y_{cons} abgelegt ist, wird später im Rahmen der Übersetzung eine Rolle spielen.

Im folgenden Beispiel 4.2 ist eine mögliche Konfiguration eines Berechnungsstacks wiedergegeben. Wir wollen die Gelegenheit nutzen und an dieser Stelle darauf hinweisen, daß im folgenden alle Darstellungen von Stacks und Kellern von unten nach oben wachsen.

Beispiel 4.2
In Abbildung 4.3 ist eine mögliche Konfiguration eines Berechnungsstacks CSt dargestellt.

	$\vdots$
4	$(F : 13, 8)$
3	$(P : 17)$
2	$(Y_{cons} : \texttt{Knoten}, 1)$
1	$(Y_{num} : 1.223)$

Abb. 4.3: Konfiguration eines Berechnungsstacks.

Das Berechnungsstack–Element $CSt[1]$ ist beispielsweise ein numerisches Wertelement mit Wert 1.223. Das Berechnungsstack–Element $CSt[4]$ ist ein Funktionselement mit Programmadresse 13 und Verweis 8. □

Auf der Halde werden, wie bereits erwähnt, Objekte variabler Größe gespeichert. Zum einen sind dies Knotenelemente, zum anderen Pfadverweiselemente. Ein Knotenelement entspricht einem Knoten in einem Baum, der über einem algebraischen Datentyp gebildet werden kann. In einem solchen Element werden der zum Knoten gehörende Konstruktor sowie die Nachfolgerelemente abgelegt. Der prinzipielle Aufbau eines Knotenelements ist in Abbildung 4.4 dargestellt.

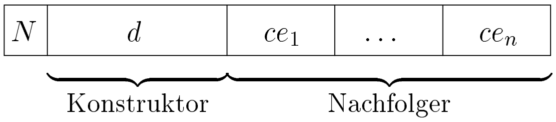


Abb. 4.4: Prinzipieller Aufbau eines Knotenelements.

Knotenelemente werden durch die Markierung N gekennzeichnet. Die Einträge d sowie $ce_1, \ldots, ce_n$ sind wiederum Berechnungsstack–Elemente. So enthält, wie auch im nachfolgenden Beispiel 4.3 angedeutet, das Feld für den Konstruktor neben dem eigentlichen Konstruktornamen auch die Markierung Y_{cons} und einen numerischen Wert.

Beispiel 4.3

Betrachten wir wiederum den algebraischen Datentyp *Baum* sowie die Menge Δ der Konstruktorsymbole mit $\Delta = \{\texttt{Knoten}^{(Baum\ Baum\ bool, Baum)}, \texttt{Blatt}^{(int, Baum)}\}$.

Der über dem algebraischen Datentyp *Baum* aufgebaute Term

$$\texttt{Knoten}(\texttt{Blatt}(\underline{1}), \texttt{Blatt}(\underline{32}), \texttt{TRUE})$$

läßt sich in der in Abbildung 4.5 dargestellten Weise mittels Knotenelementen auf der Halde speichern. Der Verweis auf den Wurzelknoten kann als Verweiselement $(P : adr_0)$ auf dem Berechnungsstack abgelegt werden. □

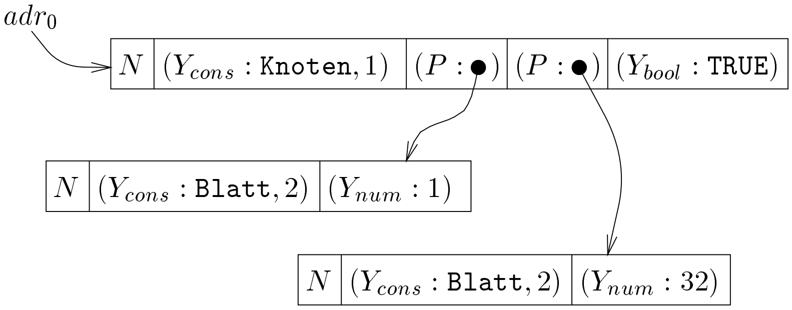


Abb. 4.5: Haldendarstellung von $\texttt{Knoten}(\texttt{Blatt}(\underline{1}), \texttt{Blatt}(\underline{32}), \texttt{TRUE})$.

Die zweite Sorte von Haldenelementen sind die Pfadverweiselemente. Diese werden wir im Zusammenhang mit macro attributed tree transducern (mat) benötigen. Wie wir in Abschnitt 3.1 gezeigt haben, bewegt sich ein mat mittels einer Pfadangabe in einem gegebenen Eingabebaum, wobei die Bewegung sowohl in Richtung der Wurzel als auch in Richtung der Blätter möglich ist. Eine Möglichkeit, dieses Verhalten nachzubilden, wäre das Ablegen eines Verweises auf einen Eingabebaum t und eine Pfadangabe *path*, wobei *path* eine Folge von Zahlen ist. Bei Bewegungen im Eingabebaum würde dann die Pfadangabe verlängert oder verkürzt werden. Wenn die Beschriftung eines Knotens k in

t, der mit der Pfadangabe *path* erreicht werden kann, benötigt wird, dann muß t mit den Angaben aus *path* durchlaufen werden.

Diese Vorgehensweise läßt sich aber verbessern: Anstatt die Pfadangabe als eine Folge von Zahlen zu realisieren, kann man die Pfadangabe auch direkt als Folge von Verweisen in den Eingabebaum implementieren. In Beispiel 4.4 diskutieren wir diese beiden Alternativen an einem konkreten Eingabebaum.

Beispiel 4.4

Sei S die Basissortenmenge $\{int, card, char, bool, real, Baum\}$ mit dem bekannten, benutzerdefinierten algebraischen Datentyp $Baum$. Die Menge Δ der Konstruktorsymbole sei also wiederum $\{\texttt{Knoten}^{(Baum\ Baum\ bool, Baum)}, \texttt{Blatt}^{(int, Baum)}\}$.

In der Mitte der Abbildung 4.6 befindet sich eine vereinfachte Darstellung eines Eingabebaums der Sorte $Baum$. Anhand eines Verweises auf den Knoten, der mit der Pfadangabe 2.2 erreicht wird, stellen wir nun die beide oben angesprochenen Möglichkeiten graphisch dar. Die klein an den Knotenelementen annotieren Zahlen sind die Pfadangaben, mit denen die Knoten im Eingabebaum erreicht werden können.

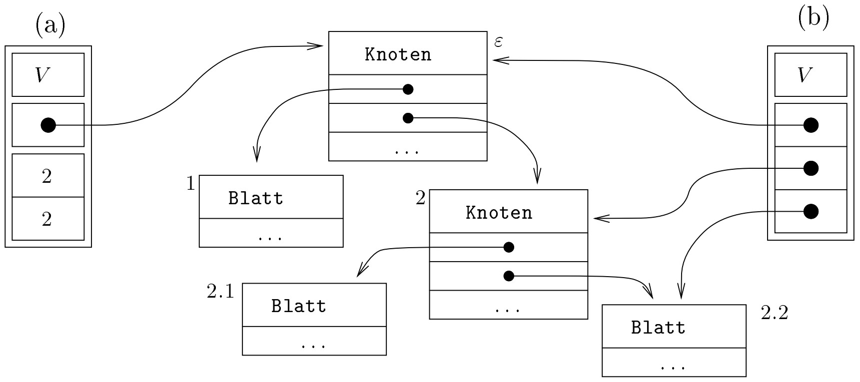


Abb. 4.6: Mögliche Realisierung von Pfadverweiselementen.

In der linken Hälfte der Abbildung ist mit (a) die erste Möglichkeit der Realisierung eines Pfadverweiselementes illustriert. Nach der Markierung V folgt zuerst ein Verweis in den Eingabebaum und dann eine Folge von Nachfolgerverweisen (hier zweimal die 2, da auf dem Knoten mit der Pfadangabe 2.2 verwiesen werden soll).

In der rechten Hälfte der Abbildung ist mit (b) die zweite Realisierungsmöglichkeit eines Pfadverweiselementes gezeigt. Nach der Markierung V folgen direkt die Verweise zu allen Knoten, die auf dem Weg von der Wurzel zum Knoten mit der Pfadangabe 2.2 berührt werden.

Soll nun der Konstruktor des Knotens 2.2 ermittelt werden, so muß im Fall (a) der gesamte Eingabebaum gemäß den Nachfolgernummern traversiert werden. Im Fall (b) kann direkt auf den betreffenden Knoten zugegriffen werden. □

Durch die Verwendung einer Folge von Verweisen können also die benötigten Knotenbeschriftungen ohne vorheriges Durchlaufen des Eingabebaums direkt ermittelt werden.

Auf einer solchen Folge von Verweisen sind nun einige Operationen erforderlich:

- Entfernen des letzten Verweises der Folge (Bewegung in Richtung der Wurzel)
- Anhängen eines Verweises an die Folge (Bewegung in Richtung der Blätter)
- Wert des letzten Verweises ermitteln (Mit diesem Verweis läßt sich dann die Beschriftung des aktuell betrachteten Knotens im Eingabebaum bestimmen.)

Diese Operationen legen die Realisierung dieser Verweisfolge in Gestalt eines Kellers nahe. Ein Pfadverweiselement enthält also einen Keller über Verweisen. Die Markierung eines Pfadverweiselements ist V.

Definition 4.5 faßt diese Überlegungen zu den beiden verschiedenen Arten von Haldenelementen formal zusammen.

Definition 4.5 (Knotenelement, Pfadverweiselement, Haldenelement)
Die Menge der *Knotenelemente* ist die Menge

$$\{(N : d, ce_1 \dots ce_k) \mid \begin{array}{l} d \in \mathit{CST\_ELEM}(\Delta), k \geq 0 \text{ und} \\ ce_i \in \mathit{CST\_ELEM}(\Delta) \text{ für } i \in [k]\}. \end{array}$$

Die Menge der *Pfadverweiselemente* ist die Menge

$$\{(V : pd) \mid pd \text{ ist Keller über } \mathbb{N}\}.$$

Die Menge der *Haldenelemente* ist die Vereinigung der Menge der Knotenelemente mit der Menge der Pfadverweiselemente und wird mit $\mathit{HEAP\_ELEM}(\Delta)$ bezeichnet. □

Auf den beiden Kellern, dem Sprungkeller und dem Umgebungskeller, werden natürliche Zahlen abgelegt. Im Falle des Sprungkellers handelt es sich dabei um Programmadressen, die angeben, an welcher Stelle des aktuell ausgeführten Maschinenprogramms nach der Rückkehr aus einem Unterprogramm weitergemacht werden soll. Auf dem Umgebungskeller werden Verweise in den Berechnungsstack abgelegt. Diese Verweise werden benötigt, um den korrekten Zugriff auf Variablen zu ermöglichen.

Die verschiedenen Register können nur ganze Zahlen als Werte enthalten. Der Programmzähler gibt an, welcher Befehl des Maschinenprogramms als nächster ausgeführt werden soll. Der Kellerrahmen–Zeiger verweist auf den aktuellen Kellerrahmen. Nachdem die Maschine angehalten hat, befindet sich im Rückgabewert–Register ein Wert, der den Grund hierfür enthält.

Nachdem wir uns in diesem Abschnitt mit den Komponenten beschäftigt haben, die sich zur Laufzeit verändern, kommen wir in Abschnitt 4.1.2 zu der während der Laufzeit unveränderlichen Komponente, dem Maschinenprogramm.

4.1.2 Maschinenprogramm

Bevor wir eine Definition für ein Maschinenprogramm angeben können, müssen wir zuerst festlegen, was wir unter einem Maschinenbefehl verstehen wollen. In Definition 4.6 haben wir alle möglichen Maschinenbefehle aufgeführt. Ein Teil des Befehlssatzes besteht aus Maschinenbefehlen, wie sie sich üblicherweise in den meisten abstrakten und realen Maschinen finden. Zu dieser Kategorie zählen die Sprung– und Unterprogrammbefehle, die arithmetischen und logischen Befehle sowie die Transportbefehle. Der andere Teil des Befehlssatzes besteht aus spezielleren Maschinenbefehlen, die in den wenigsten anderen Maschinen ein Gegenstück finden. Die Wirkungsweise der einzelnen Befehle wird jedoch im Abschnitt 4.2 genau erläutert, weswegen wir hier nicht näher auf die Bedeutung der Befehle eingehen wollen. An dieser Stelle wollen wir nochmals daran erinnern, daß wir mit Δ die Menge der Konstruktorsymbole bezeichnen.

Definition 4.6 (Maschinenbefehl)

Die möglichen Elemente der *Menge aller Maschinenbefehle*, kurz $CMD(\Delta)$, sind im folgenden aufgeführt.

Sprung– und Unterprogrammbefehle			
stop	jump adr	jump_on_true adr	jump_idx adr
call adr	ret	ret_param	
Arithmetische und logische Befehle			
add	sub	mul	div
div_int	and	or	grt
geq	les	leq	equ
neq			
Transportbefehle			
loadRC n	set_AFP n	push AFP	push CSTACK[AFP]
push_const Y	push_adr adr AFP	push_adr adr ENV	
Befehle für Konstruktoren			
make_node n	deref_node	extract n	get_son_nr
Befehle zur Manipulation von Pfadverweiselementen			
make_stack	make_copy	heap_top	heap_pop
heap_push			
Sonstige Befehle			
eval q	eval_param q	write_back q	dup
cleanup	cons_to_num		

Dabei bezeichnet $n \in \mathbb{N}_0$ eine natürliche Zahl, $q \in \mathbb{Z}$ eine ganze Zahl und $adr \in \mathbb{N}$ eine Adresse. Das Symbol $Y \in Y\_ELEM(\Delta)$ steht für ein Wertelement. □

Ein Maschinenprogramm ist nun eine Folge von Maschinenbefehlen, wobei jedem Befehl seine Position in der Folge als Adresse zugeordnet ist.

Betrachtet man diese Beziehung nun aus der Sicht der Adressen, so erhält man eine Funktion, die manchen Adressen einen Maschinenbefehl zuweist.

Definition 4.7 (Maschinenprogramm, *LNoMach*)

Ein *Maschinenprogramm über* Δ ist eine partielle Funktion

$$PStore : \mathbb{N} - \!\!\!\rightarrow CMD(\Delta).$$

Die Menge aller Maschinenprogramme über Δ wird mit $LNoMach(\Delta)$ bezeichnet. □

Das folgende Beispiel soll diese Definition ein wenig erläutern. Außerdem führen wir darin eine vereinfachende Notation für ein Maschinenprogramm ein.

Beispiel 4.8 (Maschinenprogramm)

Sei Δ eine beliebige Menge von Konstruktoren. Die Funktion *PStore* mit $PStore(1) =$ push_const $(Y_{num} : 1.0)$, $PStore(2) =$ add und $PStore(3) =$ jump 1 ist ein Maschinenprogramm über Δ. Anstelle dieser etwas unhandlichen Notation werden wir im folgenden die besser lesbare Schreibweise

1 : push_const $(Y_{num} : 1.0)$
2 : add
3 : jump 1

für ein Maschinenprogramm verwenden. □

Nun haben wir alle für eine NoName–Maschine nötigen Begriffe erklärt.

4.1.3 NoName–Maschine

Eine NoName–Maschine setzt sich nun aus den Komponenten, die wir in den vorigen Abschnitten bereits ausführlich besprochen haben, zusammen.

Definition 4.9 (NoName–Maschine, NoMach)

Eine *NoName–Maschine*, kurz NoMach, ist ein Tupel $\mathcal{M} = (\Delta, PStore, STATE)$, wobei für die einzelnen Komponenten folgendes gilt.

- Δ ist eine endliche Menge von Konstruktorsymbolen.
- $PStore \in LNoMach(\Delta)$ ist ein Maschinenprogramm.
- $STATE = PC \times AFP \times RC \times CSTACK \times JPD \times ENVPD \times HP$ ist die Menge der Maschinenzustände. Dabei ist
  - ⋄ $PC = \mathbb{N}_0$ der Programmzähler (*program counter*),
  - ⋄ $AFP = \mathbb{N}$ der Kellerrahmen–Zeiger (*active frame pointer*),
  - ⋄ $RC = \mathbb{Z}$ das Rückgabewert–Register (*return code*),
  - ⋄ $CSTACK = STACK(CST\_ELEM(\Delta))$ der Berechnungsstack (*calculation stack*),
  - ⋄ $JPD = \mathbb{N}^*$ der Sprungkeller (*jump pushdown*). Eine Konfiguration *jpd* eines Sprungkellers wird durch eine Sequenz $j.1 : j.2 : \ldots : j.n$ beschrieben, wobei die $j.i \in \mathbb{N}$ für alle $i \in [n]$. Das Element $j.1$ heißt oberstes Kellerelement.
  - ⋄ $ENVPD = \mathbb{N}^*$ der Umgebungskeller (*environment pushdown*). Eine Konfiguration *envpd* eines Umgebungskellers wird durch eine Sequenz $e.1 : e.2 : \ldots : e.n$ beschrieben, wobei für alle $i \in [n]$ gilt, daß $e.i \in \mathbb{N}$ ist. Das Element $e.1$ heißt oberstes Kellerelement.
  - ⋄ $HP = HEAP(HEAP\_ELEM(\Delta))$ die Halde (*Heap*). □

Der Zustand einer NoMach, auch Konfiguration genannt, hängt von den Zuständen der einzelnen Komponenten ab.

Definition 4.10 (Konfiguration einer NoMach)

Eine *Konfiguration einer NoMach* $\mathcal{M} = (\Delta, PStore, STATE)$ ist ein Element der Menge $STATE$ und wird durch ein Tupel $(pc, afp, rc, cst, jpd, envpd, hp)$ dargestellt. □

Dem Leser wird auffallen, daß das Maschinenprogramm, obgleich weiter oben ebenfalls als Komponente einer NoMach eingeführt, nicht zur Maschinenkonfiguration beiträgt. Dies liegt daran, daß das Maschinenprogramm einer NoMach nicht verändert werden kann und somit nie seinen Zustand ändert.

Nach dieser Beschreibung der einzelnen Komponenten werden wir nun die Wirkungsweise der einzelnen Befehle beachten.

4.2 Befehlssemantik

Bevor wir aber die einzelnen Befehle detailliert beschreiben, wollen wir zuerst einige Befehle informell erläutern, um ein intuitives Verständnis für die prinzipielle Arbeitsweise einer NoMach zu schaffen. Dabei werden bewußt nicht alle Befehle beschrieben, sondern nur die wesentlichen Vertreter einer jeden Befehlsgruppe.

Bei manchen Befehlen haben wir versucht, die Wirkungsweise durch Grafiken zu verdeutlichen. Um die Grafiken übersichtlich zu halten, haben wir jeweils nur die Komponenten der Maschine dargestellt, an denen sich Veränderungen ergeben. Die linke Hälfte jeder Grafik zeigt die Konfiguration vor Ausführung des Maschinenbefehls, die rechte Hälfte nach Ausführung. Die Abkürzungen PC, AFP und RC in den Grafiken stehen dabei für Programmzähler, Kellerrahmen–Zeiger und Rückgabewert–Register.

An dieser Stelle wollen wir nochmals darauf hinweisen, daß in den folgenden Abbildungen der Berechnungsstack sowie der beiden Hilfskeller so dargestellt werden, daß sie von unten nach oben wachsen.

Die Sprung– und Unterprogrammbefehle steuern den Programmfluß, indem sie den Wert des Programmzählers modifizieren. Der Programmzähler gibt an, welcher Befehl als nächstes zu bearbeiten ist. Hat der Programmzähler den Wert 0, so hält die Maschine an.

Der Befehl stop hält die Maschine an, indem eben der Programmzähler auf 0 gesetzt wird. Mit dem Befehl jump *adr* wird der Programmzähler auf den Wert *adr* gesetzt. Der nächste auszuführende Befehl ist also $PStore(adr)$.

Während der Befehl jump den Programmzähler in jedem Fall verändert, führt der Befehl jump_on_true nur dann einen Sprung durch, wenn das oberste Element des Berechnungsstacks das Wertelement $(Y_{bool}$: TRUE) ist. Der call–Befehl führt ähnlich wie der jump–Befehl einen unbedingten Sprung durch, allerdings wird zuvor der alte Wert des Programmzählers auf den Sprungkeller gerettet. Mittels des Befehls ret kann, wie Abbildung 4.7 zeigt, diese gerettete Adresse wieder aufgegriffen werden. Mit Hilfe der beiden Befehle call und jump können also Unterprogramme realisiert werden.

Abb. 4.7: Wirkungsweise des ret–Befehls.

Die arithmetischen und logischen Befehlen entsprechen den Infixoperatoren in NoName*. Diese Befehle entfernen die beiden obersten Elemente des Berechnungsstacks und legen statt deren ein neues Wertelement auf, dessen Wert der Verknüpfung der beiden entfernten Werte entspricht. In Abbildung 4.8 wird diese Vorgehensweise anhand des add–Befehls verdeutlicht.

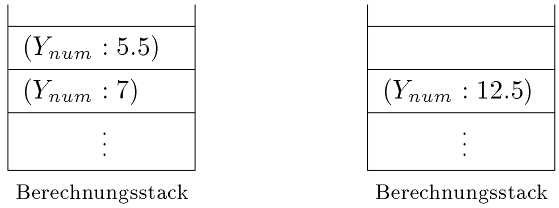


Abb. 4.8: Wirkungsweise des add–Befehls.

Die meisten Infixoperatoren sind nur auf Zahlen definiert. Die Operatoren = und <> jedoch sind auf allen Datentypen definiert, also insbesondere auch auf algebraischen Datentypen. Wie wir oben erläutert haben, werden Werte, die mit Hilfe von algebraischen Datentypen aufgebaut werden, auf der Halde gespeichert. Daraus folgt, daß ein direkter Vergleich der beiden obersten Berechnungsstack–Elemente, bei denen es sich in diesem Fall um Verweiselemente handeln muß, nicht das gewünschte Ergebnis liefert. Die Befehle equ und neq sind so definiert, daß sie in dem Fall, daß Verweiselemente $(P : p_1)$ und $(P : p_2)$ zu vergleichen sind, nicht p_1 und p_2 miteinander vergleichen, sondern die damit verbundenen Haldenelemente. Falls die Haldenelemente Knotenelemente sind, in denen wiederum Verweiselemente enthalten sind, so werden rekursiv anstelle der Verweiselemente wiederum die damit verbundenen Haldenelemente verglichen.

Die Transportbefehle bewirken eine direkte Veränderung bestimmter Register oder des Berechnungsstacks. Mit loadRC m wird das Rückgabewert–Register auf den Wert m gesetzt. set_AFP bewirkt eine Modifizierung des AFP–Registers. Die Befehle push, push_const und push_adr bewirken das Auflegen von Wert– und Funktionselementen auf dem Berechnungsstack. In Abbildung 4.9 ist dies für den Befehl push_const $(Y_{num} : 1.23)$ gezeigt.

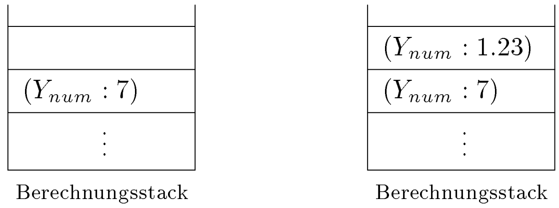


Abb. 4.9: Wirkungsweise des Befehls push_const $(Y_{num} : 1.23)$.

Wir bitten zu beachten, daß mit den verschiedenen push–Befehlen keine Verweiselemente auf dem Berechnungsstack abgelegt werden können. Dies kann nur indirekt mit Befehlen der Befehlsgruppen „Befehle für Konstruktoren“ und „Befehle zur Manipulation von Pfadverweiselementen“ erfolgen. Dadurch ist sichergestellt, daß alle Verweise, die in Verweiselementen enthalten sind, auch wirklich auf Haldenelemente zeigen.

Die Befehle für Konstruktoren kümmern sich nun um eine bestimmte Art von Haldenelementen, die Knotenelemente. Mit make_node m werden, wie Abbildung 4.10 zeigt, die obersten $(m+1)$ Elemente des Berechnungsstacks zu einem Knotenelement zusammengefaßt.

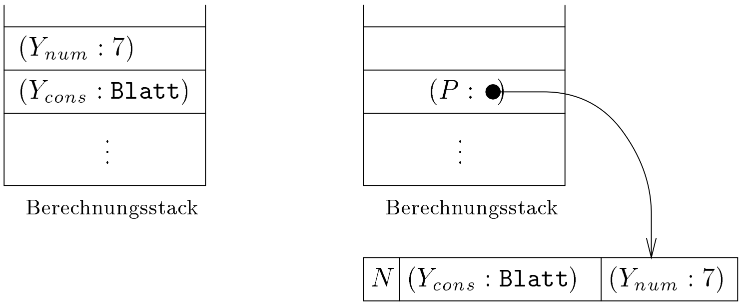


Abb. 4.10: Wirkungsweise des Befehls make_node 1.

Der Befehl deref_node macht dieses Zusammenfassen wieder rückgängig. Mit extract m kann die m–te Komponente eines Knotenelements herausprojiziert und oben auf dem Berechnungsstack abgelegt werden.

Die Befehle zur Manipulation von Pfadverweiselementen kümmern sich, wie der Name bereits andeutet, um die zweite Sorte von Haldenelementen, die Pfadverweiselemente. Mit make_stack kann ein neues Pfadverweiselement angelegt werden. Die Befehle heap_top, heap_pop und heap_push realisieren die Operationen, die auf dem Keller von Verweisen zugelassen sind. In Abbildung 4.11 ist dies anhand des Befehls heap_push veranschaulicht.

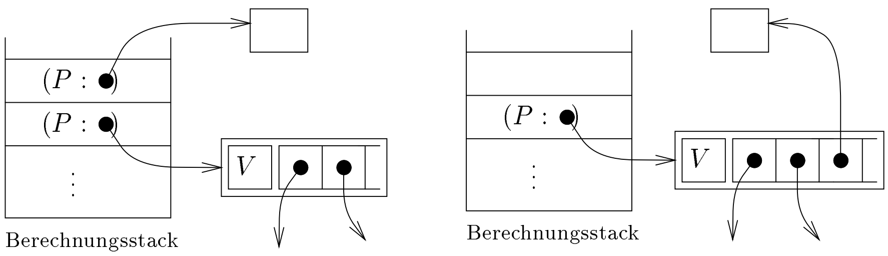


Abb. 4.11: Wirkungsweise des Befehls heap_push.

Bei der Abarbeitung eines übersetzten NoName*–Programms wird für jeden Funktionsaufruf ein sogenannter Kellerrahmen auf dem Berechnungsstack abgelegt. In Abbildung 4.12 ist ein solcher Kellerrahmen skizziert. Der Beginn des aktuellen Kellerrahmens wird durch den Wert des Kellerrahmen–Zeigers markiert.

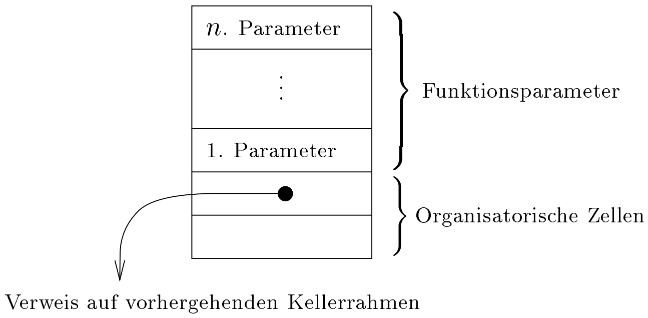


Abb. 4.12: Aufbau eines Kellerrahmens.

Der Befehl eval k bewirkt nun, daß der k–te Parameter aus dem aktuellen Funktionsrahmen nach oben auf den Berechnungsstack kopiert wird. Falls das betrachtete Berechnungsstack–Element ein Funktionselement $(F : adr,\ fp)$ ist, so wird das Unterprogramm an der Stelle adr angesprungen. Nach Abarbeitung dieses Unterprogramms liegt dann der berechnete Wert als oberstes Element auf dem Berechnungsstack.

Mit write_back k wird das oberste Element des Berechnungsstacks an die Stelle des k–ten Parameters zurückgeschrieben. Der Maschinenbefehl dup veranlaßt die Verdopplung des obersten Elements des Berechnungsstacks. Mit dem Maschinenbefehl cleanup wird, wie auch in Abbildung 4.13 dargestellt, der aktuelle Kellerrahmen aufgegeben, der Verweis auf den alten Kellerrahmen wiederhergestellt und das Berechnungsergebnis verschoben.

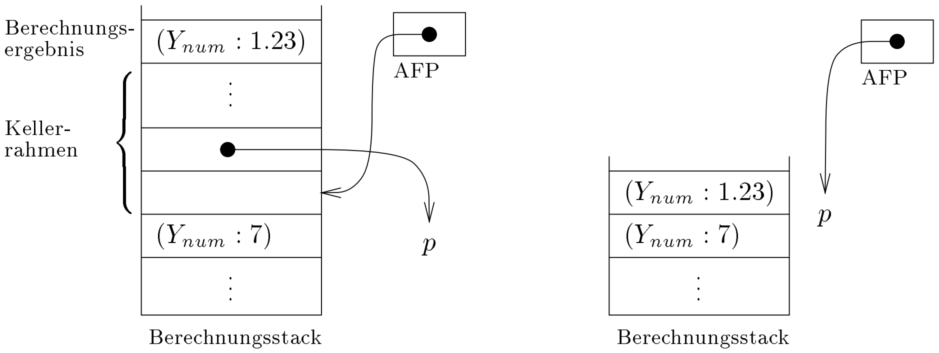


Abb. 4.13: Wirkungsweise des Befehls cleanup.

Nach dieser ersten, informellen Vorstellung der Befehle folgt nun eine formale Beschreibung der Wirkungsweise aller Befehle. Die Beschreibung der Wirkungsweise eines Befehls wird Befehlssemantik genannt.

Definition 4.11 (Befehlssemantik)
Die Befehlssemantik einer NoMach $\mathcal{M} = (\Delta, PStore, STATE)$ ist eine Funktion

$$\mathcal{C}[\![\ \cdot\ ]\!] : CMD(\Delta) \to (STATE \to STATE). \qquad \square$$

Im folgenden sind die einzelnen Zustandstransformationen, die durch die verschiedenen Befehle ausgelöst werden, aufgeführt und genauer erläutert. Dazu wird jeweils angegeben, wie sich die Komponenten einer Konfiguration bei Ausführung eines Maschinenbefehls verändern. Um die geänderten Komponenten der Konfiguration einfacher erkennen zu können, werden diese unterstrichen dargestellt.

4.2.1 Zustandstransformationen

4.2.1.1 Sprung– und Unterprogrammbefehle

stop–Befehl

Der Befehl stop bewirkt das Anhalten der Maschine. Dazu wird der Programmzähler auf den Wert 0 gesetzt.

$$\begin{array}{l}\mathcal{C}[\![\,\mathsf{stop}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\ \quad (\underline{0}, afp, rc, cst, jpd, envpd, hp)\end{array}$$

jump–Befehl

Der Befehl jump adr mit $adr \in I\!N$ setzt den Programmzähler auf den Wert adr. Die anderen Komponenten der Maschine werden von diesem Befehl nicht verändert.

$$\begin{array}{l}\mathcal{C}[\![\,\mathsf{jump}\ adr\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\ \quad (\underline{adr}, afp, rc, cst, jpd, envpd, hp)\end{array}$$

jump_on_true–Befehl

Der Befehl jump_on_true adr mit $adr \in I\!N$ setzt den Programmzähler auf den Wert adr, falls das oberste Element auf dem Berechnungsstack das Wertelement $(Y_{bool} : \mathtt{TRUE})$ ist. Falls dieses oberste Element $(Y_{bool} : \mathtt{FALSE})$ ist, wird der Programmzähler um 1 inkrementiert. Das oberste Element des Berechnungsstacks wird in jedem Fall entfernt. Sollte das oberste Berechnungsstack–Element nicht eines der eben erwähnten Wertelemente sein, so hält die Maschine an und es wird der Fehlercode -1 in das Rückgabewert–Register geschrieben.

$$\begin{array}{l}\mathcal{C}[\![\,\mathsf{jump\_on\_true}\ adr\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\ \quad \textbf{if } top(cst) = (Y_{bool} : b) \\ \quad\quad \textbf{if } b = \mathtt{TRUE} \\ \quad\quad\quad (\underline{adr}, afp, rc, \underline{pop(cst)}, jpd, envpd, hp) \\ \quad\quad \textbf{otherwise} \\ \quad\quad\quad (\underline{pc+1}, afp, rc, \underline{pop(cst)}, jpd, envpd, hp) \\ \quad \textbf{otherwise} \\ \quad\quad (\underline{0}, afp, \underline{-1}, cst, jpd, envpd, hp)\end{array}$$

jump_idx–Befehl

Der Befehl jump_idx adr bewirkt einen indizierten Sprung. Dazu wird das oberste Element des Berechnungsstacks, das die Form $(Y_{num} : n)$ mit $n \geq 1$ haben muß, heruntergenommen und der Programmzähler auf den Wert $adr + \lfloor n \rfloor - 1$ gesetzt. Die Bildung des größten Ganzen kleiner oder gleich n ist für den Fall notwendig, daß n keine natürliche Zahl ist. Sollte das oberste Berechnungsstack–Element nicht die obige Form haben, hält die Maschine an und es wird der Fehlercode -2 in das Rückgabewert–Register geschrieben.

$$\begin{array}{l} \mathcal{C}[\![\, \mathsf{jump\_idx}\ adr \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\ \quad \textbf{if } top(cst) = (Y_{num} : n) \textbf{ and } n \geq 1 \\ \quad\quad (\underline{adr + \lfloor n \rfloor - 1}, afp, rc, \underline{pop(cst)}, jpd, envpd, hp) \\ \quad \textbf{otherwise} \\ \quad\quad (\underline{0}, afp, \underline{-2}, cst, jpd, envpd, hp) \end{array}$$

call–Befehl

Der Befehl call adr bewirkt einen Unterprogramm–Aufruf. Dazu wird der um eins erhöhte Wert des Programmzählers oben auf den Sprungkeller gelegt und der Programmzähler auf den neuen Wert adr gesetzt. Das Gegenstück zum call–Befehl ist der ret–Befehl.

$$\begin{array}{l} \mathcal{C}[\![\, \mathsf{call}\ adr \,]\!](pc, afp, rc, cst, j.1 : \ldots : j.k, envpd, hp) = \\ \quad (\underline{adr}, afp, rc, cst, \underline{(pc+1) : j.1 : \ldots : j.k}, envpd, hp) \end{array}$$

ret–Befehl

Der Befehl ret bewirkt die Rückkehr aus einem Unterprogramm, das mit call aufgerufen wurde. Dazu wird der Programmzähler auf den Wert gesetzt, der oben auf dem Sprungkeller liegt. Dieses oberste Sprungkellerelement wird anschließend entfernt. Sollte kein Element auf dem Sprungkeller liegen, so hält die Maschine an und es wird der Fehlercode -3 in das Rückgabewert–Register geschrieben.

$$\begin{array}{l} \mathcal{C}[\![\, \mathsf{ret} \,]\!](pc, afp, rc, cst, j.1 : \ldots : j.k, envpd, hp) = \\ \quad \textbf{if } k \geq 1 \\ \quad\quad (\underline{j.1}, afp, rc, cst, \underline{j.2 : \ldots : j.k}, envpd, hp) \\ \quad \textbf{otherwise} \\ \quad\quad (\underline{0}, afp, \underline{-3}, cst, j.1 : \ldots : j.k, envpd, hp) \end{array}$$

ret_param–Befehl

Der Befehl ret_param bewirkt die Rückkehr aus einem Unterprogramm, das mit eval oder eval_param aufgerufen wurde. Dazu wird der Programmzähler auf den Wert gesetzt, der oben auf dem Sprungkeller liegt. Dieses oberste Sprungkellerelement wird anschließend entfernt. Außerdem wird das oberste Element des Umgebungskellers entfernt. Sollte kein Element auf dem Sprungkeller oder Umgebungskeller liegen, so hält die Maschine an, und der Fehlercode -4 wird in das Rückgabewert–Register geschrieben.

$$\begin{array}{l} \mathcal{C}[\![\, \mathsf{ret\_param} \,]\!](pc, afp, rc, cst, j.1 : \ldots : j.k, e.1 : \ldots : e.l, hp) = \\ \quad \textbf{if } k \geq 1 \textbf{ and } l \geq 1 \\ \quad\quad (\underline{j.1}, afp, rc, cst, \underline{j.2 : \ldots : j.k}, \underline{e.2 : \ldots : e.l}, hp) \\ \quad \textbf{otherwise} \\ \quad\quad (\underline{0}, afp, \underline{-4}, cst, j.1 : \ldots : j.k, e.1 : \ldots : e.l, hp) \end{array}$$

4.2.1.2 Arithmetische und logische Befehle

add, sub, mul–Befehl

Die Befehl add, sub und mul bilden jeweils die arithmetische Verknüpfung der Werte der beiden obersten Elemente des Berechnungsstacks. Dann werden die beiden betrachteten Elemente entfernt und ein neues Wertelement auf dem Berechnungsstack aufgelegt, dessen Wert gleich dem Ergebnis der Verknüpfung ist. Sollten keine zwei Elemente auf dem Berechnungsstack gelegen haben oder war mindestens ein Element kein Wertelement des Typs Y_{num}, so hält die Maschine mit dem Fehlercode -5 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, op \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } top(cst) = (Y_{num} : n_2) \textbf{ and } top(pop(cst)) = (Y_{num} : n_1) \\
\qquad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-5}, cst, jpd, envpd, hp) \\
\quad \textbf{where} \\
\qquad cst' = push(pop(pop(cst)), (Y_{num} : (n_1\ op'\ n_2))) \\
\qquad op' = \begin{cases} +, & \text{falls } op = \mathsf{add} \\ -, & \text{falls } op = \mathsf{sub} \\ *, & \text{falls } op = \mathsf{mul} \end{cases}
\end{array}
$$

div, div_int–Befehl

Die Befehle div und div_int arbeiten im Prinzip genau wie die eben besprochenen Befehle add, sub und mul. Allerdings kann bei ihrer Anwendung noch ein weiterer Fehler auftreten, nämlich die Division durch Null. Diese wird mit dem Fehlercode -6 gemeldet und bewirkt ebenfalls ein Anhalten der Maschine.

$$
\begin{array}{l}
\mathcal{C}[\![\, op \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } top(cst) = (Y_{num} : n_2) \textbf{ and } top(pop(cst)) = (Y_{num} : n_1) \\
\qquad \textbf{if } n_2 \neq 0 \\
\qquad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\qquad \textbf{otherwise} \\
\qquad\quad (\underline{0}, afp, \underline{-6}, cst, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-5}, cst, jpd, envpd, hp) \\
\quad \textbf{where} \\
\qquad cst' = push(pop(pop(cst)), (Y_{num} : (n_1\ op'\ n_2))) \\
\qquad op' = \begin{cases} /, & \text{falls } op = \mathsf{div} \\ \texttt{div}, & \text{falls } op = \mathsf{div\_int} \end{cases}
\end{array}
$$

and, or–Befehl

Die Befehle and und or bilden jeweils die logische Verknüpfung der Werte der beiden obersten Elemente des Berechnungsstacks. Dann werden die beiden obersten Elemente des Berechnungsstacks entfernt und ein neues Wertelement aufgelegt, dessen Wert das Ergebnis der Verknüpfung ist. Sollten keine zwei Elemente auf dem Berechnungsstack gelegen haben oder war mindestens ein Element kein Wertelement des Typs Y_{bool}, so hält die Maschine mit dem Fehlercode -7 an.

$$
\begin{aligned}
&\mathcal{C}[\![op]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
&\quad \textbf{if } top(cst) = (Y_{bool} : b_2) \textbf{ and } top(pop(cst)) = (Y_{bool} : b_1) \\
&\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
&\quad \textbf{otherwise} \\
&\quad\quad (\underline{0}, afp, \underline{-7}, cst, jpd, envpd, hp) \\
&\quad \textbf{where} \\
&\quad\quad cst' = push(pop(pop(cst)), (Y_{num} : (b_1\, op'\, b_2))) \\
&\quad\quad op' = \begin{cases} \texttt{and}, & \text{falls } op = \mathsf{and} \\ \texttt{or}, & \text{falls } op = \mathsf{or} \end{cases}
\end{aligned}
$$

grt, geq, les, leq–Befehl

Die Befehle grt, geq, les und leq betrachten die beiden Werte der obersten Elemente des Berechnungsstacks und vergleichen diese. Dann werden die beiden betroffenen Elemente vom Berechnungsstack entfernt und ein neues Wertelement aufgelegt, dessen Wert gleich dem Ergebnis des Vergleichs ist. Sollten keine zwei Elemente auf dem Berechnungsstack gelegen haben oder war mindestens ein Element kein Wertelement des Typs Y_{num}, so hält die Maschine mit dem Fehlercode -5 an.

$$
\begin{aligned}
&\mathcal{C}[\![op]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
&\quad \textbf{if } top(cst) = (Y_{num} : n_2) \textbf{ and } top(pop(cst)) = (Y_{num} : n_1) \\
&\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
&\quad \textbf{otherwise} \\
&\quad\quad (\underline{0}, afp, \underline{-5}, cst, jpd, envpd, hp) \\
&\quad \textbf{where} \\
&\quad\quad cst' = push(pop(pop(cst)), (Y_{bool} : (n_1\, op'\, n_2))) \\
&\quad\quad op' = \begin{cases} >, & \text{falls } op = \mathsf{grt} \\ \geq, & \text{falls } op = \mathsf{geq} \\ <, & \text{falls } op = \mathsf{les} \\ \leq, & \text{falls } op = \mathsf{leq} \end{cases}
\end{aligned}
$$

equ, neq–Befehl

Die Befehle equ und neq betrachten die Werte der beiden obersten Elemente des Berechnungsstacks und testen diese auf Gleichheit bzw. Ungleichheit. Falls es sich bei den betrachteten Elementen um Verweiselemente handelt, so werden nicht die Verweise, sondern die von diesen Werten referenzierten Haldenelemente verglichen. Falls diese Haldenelemente Knotenelemente sind und ihrerseits ebenfalls Verweiselemente beinhalten, so werden auch die dadurch referenzierten Haldenelemente rekursiv verglichen. Die Operatoren $=_p$ und $\neq_p$ bewirken diesen Vergleich von Verweiselementen.

Das Ergebnis des Vergleichs wird in einem booleschen Wertelement oben auf dem Berechnungsstack abgelegt. Sollten keine zwei Elemente auf dem Berechnungsstack gelegen haben, eines kein Wertelement bzw. Verweiselement gewesen sein oder waren die Elemente von unterschiedlichem Typ sein, so hält die Maschine mit dem Fehlercode -8 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, op \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } top(cst) = (Y_{typ_1} : y_2) \textbf{ and } top(pop(cst)) = (Y_{typ_2} : y_1) \textbf{ and } typ_1 = typ_2 \\
\qquad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \textbf{if } top(cst) = (P : p_2) \textbf{ and } top(pop(cst)) = (P : p_1) \\
\qquad (\underline{pc+1}, afp, rc, \underline{cst''}, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-8}, cst, jpd, envpd, hp) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
cst' & = & push(pop(pop(cst)), (Y_{bool} : (y_1\, op'\, y_2))\,) \\
cst'' & = & push(pop(pop(cst)), (Y_{bool} : (p_1\, op''\, p_2))\,) \\
op' & = & \begin{cases} =, & \text{falls } op = \mathsf{equ} \\ \neq, & \text{falls } op = \mathsf{neq} \end{cases} \\
op'' & = & \begin{cases} =_p, & \text{falls } op = \mathsf{equ} \\ \neq_p, & \text{falls } op = \mathsf{neq} \end{cases}
\end{array}
\end{array}
$$

4.2.1.3 Transportbefehle

loadRC–Befehl

Der Befehl loadRC n mit $n \in I\!N_0$ setzt den Wert des Rückgabewert–Registers auf n. Damit ist es möglich, den Rückgabestatus eines Programms selbst zu verändern. Es ist jedoch nicht möglich, negative Werte in das Rückgabewert–Register zu schreiben, da die negativen Werte die internen Fehlercodes einer NoMach sind.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{loadRC}\ n \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad (\underline{pc+1}, afp, \underline{n}, cst, jpd, envpd, hp)
\end{array}
$$

set_AFP–Befehl

Mit dem Befehl set_AFP n mit $n \in I\!N_0$ ist es möglich, den Wert des Kellerrahmen–Zeigers auf die Höhe des Berechnungsstacks minus $(n + 1)$ zu setzen. Sollte diese Zahl negativ sein, so hält die Maschine mit dem Fehlercode -9 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{set\_AFP}\ n \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } height(cst) - (n+1) \geq 0 \\
\qquad (\underline{pc+1}, \underline{height(cst) - (n+1)}, rc, cst, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-9}, cst, jpd, envpd, hp)
\end{array}
$$

push–Befehl

Der push–Befehl ermöglicht es, ein neues Element oben auf dem Berechnungsstack abzulegen. Der Wert des Elements hängt von den beiden möglichen Parametern AFP bzw. CSTACK[AFP] ab.

Ist der Parameter AFP, so wird ein Wertelement $(Y_{num} : a)$ aufgelegt, wobei a der aktuelle Wert des Kellerrahmen–Zeigers ist. Im Falle des Parameters CSTACK[AFP] wird das a–te Element des Berechnungsstacks nach oben kopiert, wobei a der aktuelle Wert des

Kellerrahmen–Zeigers ist. In diesem Fall muß die referenzierte Speicherstelle existieren; ansonsten hält die Maschine mit dem Fehlercode -10 an.

$\mathcal{C}[\![\,\mathsf{push\ AFP}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
$\quad(\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp)$
$\quad\textbf{where } cst' = push(cst, (Y_{num} : afp))$

$\mathcal{C}[\![\,\mathsf{push\ CSTACK[AFP]}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
$\quad\textbf{if } 1 \leq afp \leq height(cst)$
$\qquad(\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp)$
$\quad\textbf{otherwise}$
$\qquad(\underline{0}, afp, \underline{-10}, cst, jpd, envpd, hp)$
$\quad\textbf{where } cst' = push(cst, cst[afp])$

push_const–Befehl

Der Befehl push_const Y mit $Y \in Y\_ELEM(\Delta)$ legt das Wertelement Y oben auf dem Berechnungsstack ab.

$\mathcal{C}[\![\,\mathsf{push\_const}\ Y\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
$\quad(\underline{pc+1}, afp, rc, \underline{push(cst, Y)}, jpd, envpd, hp)$

push_adr–Befehl

Der Befehl push_adr adr typ mit $adr \in I\!N$ und $typ \in \{\mathsf{AFP}, \mathsf{ENV}\}$ legt oben auf dem Berechnungsstack ein Funktionselement der Form $(F : adr,\ fp)$ ab. Der Wert für fp wird in Abhängigkeit des Befehlsparameters typ bestimmt. Ist dieser gleich AFP, so wird für fp der Wert des Kellerrahmen–Zeigers verwendet. Ist der zweite Befehlsparameter ENV, so wird fp gleich dem obersten Element des Umgebungskellers gesetzt. Ist der Umgebungskeller leer, so hält die Maschine im Falle des Befehlsparameters ENV mit dem Fehlercode -11 an.

$\mathcal{C}[\![\,\mathsf{push\_adr}\ adr\ \mathsf{AFP}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
$\quad(\underline{pc+1}, afp, rc, \underline{push(cst, (F : adr,\ afp))}, jpd, envpd, hp)$

$\mathcal{C}[\![\,\mathsf{push\_adr}\ adr\ \mathsf{ENV}\,]\!](pc, afp, rc, cst, jpd, e.1 : \ldots : e.k, hp) =$
$\quad\textbf{if } k \geq 1$
$\qquad(\underline{pc+1}, afp, rc, \underline{cst'}, jpd, e.1 : \ldots : e.k, hp)$
$\quad\textbf{otherwise}$
$\qquad(\underline{0}, afp, \underline{-11}, cst, jpd, e.1 : \ldots : e.k, hp)$
$\quad\textbf{where } cst' = push(cst, (F : adr,\ e.1))$

4.2.1.4 Befehle für Konstruktoren

make_node–Befehl

Der Befehl make_node k mit $k \in I\!N_0$ faßt die obersten $k{+}1$ Elemente des Berechnungsstacks zu einem Knotenelement zusammen, speichert dieses in der Halde und legt den Verweis auf dieses Element in Form eines Verweiselements oben auf dem Berechnungsstack ab. Sollten keine $k+1$ Elemente auf dem Berechnungsstack liegen, hält die Maschine mit dem Fehlercode -12 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{make\_node}\ k\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \mathbf{if}\ height(cst) \geq (k+1) \\
\qquad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, \underline{hp'}) \\
\quad \mathbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-12}, cst, jpd, envpd, hp) \\
\quad \mathbf{where}\ d = cst[height(cst) - k] \\
\qquad\qquad ce_1 = cst[height(cst) + 1 - k] \\
\qquad\qquad\quad \vdots \\
\qquad\qquad ce_k = cst[height(cst)] \\
\qquad\qquad (hp', p) = new(hp, (N : d, ce_1 \ldots ce_k)) \\
\qquad\qquad cst' = push(\underbrace{pop(pop(\ldots pop(}_{k+1 \text{ mal}} cst) \ldots), (P : p))
\end{array}
$$

deref_node–Befehl

Der deref_node–Befehl ist die Umkehrung des make_node–Befehls. Der Befehl nimmt das oberste Element des Berechnungsstacks, welches ein Verweiselement $(P : p)$ sein muß, und ersetzt dies durch die Werte des Knotenelements, auf das p gezeigt hat. Sollte kein passendes Element oben auf dem Berechnungsstack liegen oder das referenzierte Haldenelement kein Knotenelement sein, so hält die Maschine mit dem Fehlercode -13 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{deref\_node}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \mathbf{if}\ top(cst) = (P : p)\ \mathbf{and}\ hp[p] = (N : d, ce_1 \ldots ce_k) \\
\qquad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \mathbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-13}, cst, jpd, envpd, hp) \\
\quad \mathbf{where}\ cst' = push(push(\ldots push(pop(cst), ce_1) \ldots, ce_k), d)
\end{array}
$$

extract–Befehl

Der extract k–Befehl erwartet ein Verweiselement $(P : p)$ als oberstes Element des Berechnungsstacks, wobei p auf ein Knotenelement auf der Halde zeigt. Der Befehl ersetzt dieses Wertelement dann durch die k–te Komponente des betreffenden Knotenelements. Die 0–te Komponente eines Knotenelements $(N : d, ce_1 \ldots ce_n)$ ist dabei d, die 1–te Komponente ist ce_1. Sollten die Voraussetzungen des Befehls nicht erfüllt sein oder das referenzierte Element nicht existieren, so hält die Maschine mit dem Fehlercode -14 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{extract}\ k\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \mathbf{if}\ top(cst) = (P : p)\ \ \mathbf{and}\ hp[p] = (N : d, ce_1 \ldots ce_n)\ \mathbf{and}\ 0 \leq k \leq n \\
\qquad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \mathbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-14}, cst, jpd, envpd, hp) \\
\quad \mathbf{where}\ x = (\mathbf{if}\ k = 0\ \mathbf{then}\ d\ \mathbf{else}\ ce_k) \\
\qquad\qquad cst' = push(pop(cst), x)
\end{array}
$$

get_son_nr–Befehl

Der Befehl get_son_nr erwartet zwei Verweiselemente oben auf dem Berechnungsstack, wobei das obere auf ein Knotenelement zeigt, welches als Vaterknoten n_1 interpretiert wird. Das untere Element wird als Verweis auf einen Sohnknoten n_2 aufgefaßt. Der Befehl löscht nun die beiden obersten Elemente des Berechnungsstacks und legt statt deren ein Wertelement $(Y_{num} : k)$ auf, wobei k angibt, der wievielte Sohn n_2 in Bezug auf n_1 ist. Läßt sich keine Beziehung ermitteln, so ist k gleich 0. Falls die Vorbedingungen des Befehls nicht erfüllt sind, so hält die Maschine mit Fehlercode -15 an.

$$
\begin{aligned}
&\mathcal{C}[\![\,\mathsf{get\_son\_nr}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
&\quad \textbf{if } top(cst) = (P : p_1) \textbf{ and } top(pop(cst)) = (P : p_2) \textbf{ and } hp[p_1] = (N : d, ce_1 \ldots ce_n) \\
&\qquad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
&\quad \textbf{otherwise} \\
&\qquad (\underline{0}, afp, \underline{-15}, cst, jpd, envpd, hp) \\
&\quad \textbf{where } nr = (\textbf{if } ce_j = (P : p_2) \text{ für } j \in [n] \textbf{ then } j \textbf{ else } 0) \\
&\qquad\qquad cst' = push(pop(pop(cst)), (Y_{num} : nr))
\end{aligned}
$$

4.2.1.5 Befehle zur Manipulation von Pfadverweiselementen

make_stack–Befehl

Der make_stack–Befehl nimmt das oberste Element des Berechnungsstacks, welches ein Verweiselement sein muß, und ersetzt dieses durch ein anderes Verweiselement, welches auf ein neu angelegtes Pfadverweiselement mit Keller pd zeigt. Der einzige Eintrag von pd ist dabei der Wert des ursprünglichen Wertelements. Sollte kein passendes Element auf dem Berechnungsstack gelegen haben, so hält die Maschine mit Fehlercode -16 an.

$$
\begin{aligned}
&\mathcal{C}[\![\,\mathsf{make\_stack}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
&\quad \textbf{if } top(cst) = (P : p) \\
&\qquad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, \underline{hp'}) \\
&\quad \textbf{otherwise} \\
&\qquad (\underline{0}, afp, \underline{-16}, cst, jpd, envpd, hp) \\
&\quad \textbf{where } (hp', p') = new(hp, (V : p)) \\
&\qquad\qquad cst' = push(pop(cst), (P : p'))
\end{aligned}
$$

make_copy–Befehl

Der make_copy–Befehl nimmt das oberste Element des Berechnungsstacks, welches ein Verweiselement sein muß und das auf ein Pfadverweiselement v zeigt. Eine Kopie von v wird als neues Pfadverweiselement angelegt. Ein Verweiselement auf diese Kopie wird dann auf den Berechnungsstack aufgelegt. Sollte kein passendes Element auf dem Berechnungsstack gelegen haben, so hält die Maschine mit Fehlercode -17 an.

$\mathcal{C}[\![\,\textsf{make\_stack}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
$\quad \textbf{if } top(cst) = (P : p) \textbf{ and } hp[p] = (V : (p.1 : \ldots : p.n))$
$\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, \underline{hp'})$
$\quad \textbf{otherwise}$
$\quad\quad (\underline{0}, afp, \underline{-17}, cst, jpd, envpd, hp)$
$\quad \textbf{where } (hp', p') = new(hp, (V : (p.1 : \ldots : p.n)))$
$\quad\quad\quad cst' = push(pop(cst), (P : p'))$

heap_top–Befehl

Der Befehl heap_top erwartet als oberstes Element auf dem Berechnungsstack ein Verweiselement $(P : p)$, wobei p auf ein Pfadverweiselement $(V : st)$ zeigt und st ungleich dem leeren Keller ist. In diesem Fall ersetzt der Befehl dieses Verweiselement durch ein neues Verweiselement $(P : p')$, wobei p' das oberste Element des Kellers st ist. Im Fehlerfall hält die Maschine mit dem Fehlercode -18 an.

$\mathcal{C}[\![\,\textsf{heap\_top}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
$\quad \textbf{if } top(cst) = (P : p) \textbf{ and } hp[p] = (V : (p.1 : \ldots : p.n)) \textbf{ and } n \geq 1$
$\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp)$
$\quad \textbf{otherwise}$
$\quad\quad (\underline{0}, afp, \underline{-18}, cst, jpd, envpd, hp)$
$\quad \textbf{where } cst' = push(pop(cst), (P : p.1))$

heap_pop–Befehl

Der Befehl heap_pop erwartet als oberstes Element auf dem Berechnungsstack ein Verweiselement $(P : p)$, wobei p auf ein Pfadverweiselement $(V : st)$ zeigt und st ungleich dem leeren Keller ist. In diesem Fall löscht der Befehl zum einen das oberste Element des Berechnungsstacks und zum anderen das oberste Element des Kellers st. Im Fehlerfall hält die Maschine mit dem Fehlercode -18 an.

$\mathcal{C}[\![\,\textsf{heap\_pop}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
$\quad \textbf{if } top(cst) = (P : p) \textbf{ and } hp[p] = (V : (p.1 : \ldots : p.n)) \textbf{ and } n \geq 1$
$\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, \underline{hp'})$
$\quad \textbf{otherwise}$
$\quad\quad (\underline{0}, afp, \underline{-18}, cst, jpd, envpd, hp)$
$\quad \textbf{where } cst' = pop(cst)$
$\quad\quad\quad hp' = modify(hp, p, (V : (p.2 : \ldots : p.n)))$

heap_push–Befehl

Der Befehl heap_push erwartet als oberstes Element auf dem Berechnungsstack ein Verweiselement $(P : p)$ und als zweitoberstes Element ein Verweiselement $(P : p')$, wobei p' auf ein Pfadverweiselement $(V : st)$ zeigt. Der Befehl löscht dann das oberste Element des Berechnungsstacks und legt p auf den Keller st auf. Im Fehlerfall hält die Maschine mit dem Fehlercode -19 an.

$\mathcal{C}[\![\,\mathsf{heap\_push}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) =$
if $top(cst) = (P : p_2)$ **and** $top(pop(cst)) = (P : p_1)$ **and** $hp[p_1] = (V : (p.1 : \ldots : p.n))$
$(\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, \underline{hp'})$
otherwise
$(\underline{0}, afp, \underline{-19}, cst, jpd, envpd, hp)$
where $cst' = pop(pop(cst))$
$hp' = modify(hp, p, (V : (p_2 : p.1 : \ldots : p.n)))$

4.2.1.6 Sonstige Befehle

eval–Befehl

Der Befehl eval q mit $q \in \mathbb{Z}$ versucht das Element, das im Berechnungsstack an der $(a+1+q)$–ten Stelle liegt, oben auf den Berechnungsstack zu kopieren. Dabei ist a der aktuelle Wert des Kellerrahmen–Zeigers. Handelt es sich bei dem betrachteten Element um ein Funktionselement $(F : adr,\ fp)$, so wird die Auswertung dieses Funktionselementes angestoßen. Dazu wird die Adresse $pc+1$ als Rücksprungadresse auf den Sprungkeller gelegt, der Programmzähler auf adr gesetzt und fp oben auf dem Umgebungskeller abgelegt. Existiert das referenzierte Element des Berechnungsstacks nicht, so hält die Maschine mit dem Fehlercode -20 an.

$\mathcal{C}[\![\,\mathsf{eval}\ q\,]\!](pc, afp, rc, cst, j.1 : \ldots : j.l, e.1 : \ldots : e.n, hp) =$
if $cst[afp + 1 + q])$ definiert
if $cst[afp + 1 + q] = (F : adr,\ fp)$
$(\underline{adr}, afp, rc, cst, \underline{(pc+1) : j.1 : \ldots : j.l}, \underline{fp : e.1 : \ldots : e.n}, hp)$
otherwise
$(\underline{pc+1}, afp, rc, \underline{cst'}, j.1 : \ldots : j.l, e.1 : \ldots : e.n, hp)$
otherwise
$(\underline{0}, afp, \underline{-20}, cst, j.1 : \ldots : j.l, e.1 : \ldots : e.n, hp)$
where $cst' = push(cst, cst[afp + 1 + q])$

eval_param–Befehl

Der Befehl eval_param q mit $q \in \mathbb{Z}$ arbeitet im Grunde genauso wie der Befehl eval, nur bezieht sich der eval_param–Befehl auf das $(e.1 + 1 + q)$-te Berechnungsstack–Element, wobei $e.1$ der Wert des obersten Umgebungskellers–Elements ist. Bei diesem Befehl hält die Maschine im Fehlerfalle mit dem Fehlercode -21 an.

$\mathcal{C}[\![\,\mathsf{eval\_param}\ q\,]\!](pc, afp, rc, cst, j.1 : \ldots : j.l, e.1 : \ldots : e.n, hp) =$
if $cst[e_1 + 1 + q])$ definiert
if $cst[e_1 + 1 + k] = (F : adr,\ fp)$
$(\underline{adr}, afp, rc, cst, \underline{(pc+1) : j.1 : \ldots : j.l}, \underline{fp : e.1 : \ldots : e.n}, hp)$
otherwise
$(\underline{pc+1}, afp, rc, \underline{cst'}, j.1 : \ldots : j.l, e.1 : \ldots : e.n, hp)$
otherwise
$(\underline{0}, afp, \underline{-21}, cst, j.1 : \ldots : j.l, e.1 : \ldots : e.n, hp)$
where $cst' = push(cst, cst[e.1 + 1 + q])$

write_back–Befehl

Der Befehl write_back q mit $q \in \mathbb{Z}$ nimmt das oberste Element des Berechnungsstacks und verschiebt dieses an die $(a+1+q)$–te Stelle im Berechnungsstack. Dabei ist a der aktuelle Wert des Kellerrahmen–Zeigers. Sollte die $(a+1+q)$–te Stelle im Berechnungsstack nicht existieren oder der Berechnungsstack leer sein, so hält die Maschine an und schreibt den Wert -22 in das Rückgabewert–Register.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{write\_back}\ q \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } height(cst) \geq 1 \textbf{ and } cst[afp+1+q]) \text{ definiert} \\
\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\quad\quad (\underline{0}, afp, \underline{-22}, cst, jpd, envpd, hp) \\
\quad \textbf{where } cst' = write(pop(cst), afp+1+q, top(cst))
\end{array}
$$

dup–Befehl

Der dup–Befehl betrachtet das oberste Element des Berechnungsstacks und legt dasselbe Element nochmals oben auf dem Berechnungsstack ab. Sollte der Berechnungsstack leer sein, hält die Maschine mit Fehlercode -23 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{dup} \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } height(cst) \geq 1 \\
\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\quad\quad (\underline{0}, afp, \underline{-23}, cst, jpd, envpd, hp) \\
\quad \textbf{where } cst' = push(cst, top(cst))
\end{array}
$$

cleanup–Befehl

Der cleanup–Befehl verschiebt das oberste Element des Berechnungsstacks an die a–te Stelle im Berechnungsstack, wobei a der aktuelle Wert des Kellerrahmen–Zeigers ist. Außerdem wird der Wert des Kellerrahmen–Zeigers auf den Wert $\lfloor k \rfloor$ gesetzt, wobei an der $(a+1)$–ten Stelle im Berechnungsstack ein Wertelement $(Y_{num} : k)$ gelegen hat. Die Gauß-klammer ist für den Fall nötig, daß es sich bei k um keine ganze Zahl handelt. Schließlich werden im Berechnungsstack alle Elemente bis einschließlich dem $(a+1)$–ten gelöscht. Sollte bei der Ausführung dieses Befehls ein Fehler auftreten, so hält die Maschine mit dem Fehlercode -24 an.

$$
\begin{array}{l}
\mathcal{C}[\![\, \mathsf{cleanup} \,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } height(cst) \geq 1 \textbf{ and } cst[afp] \text{ definiert } \textbf{ and } cst[afp+1] = (Y_{num} : k) \\
\quad\quad (\underline{pc+1}, \underline{\lfloor k \rfloor}, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\quad\quad (\underline{0}, afp, \underline{-24}, cst, jpd, envpd, hp) \\
\quad \textbf{where } cst' = push(\underbrace{pop(pop(\ldots(pop(}_{height(cst)+1-afp} cst))\ldots), top(cst))
\end{array}
$$

cons_to_num–Befehl

Der cons_to_num bewirkt eine Umwandlung des obersten Berechnungsstack–Elementes der Gestalt $(Y_{cons} : d, n)$ in ein Element $(Y_{num} : n)$. Sollte kein Element auf dem Berechnungsstack liegen oder das oberste Element des Berechnungsstacks nicht den Typ Y_{cons} haben, so hält die Maschine mit dem Fehlercode -25 an.

$$
\begin{array}{l}
\mathcal{C}[\![\,\mathsf{cons\_to\_num}\,]\!](pc, afp, rc, cst, jpd, envpd, hp) = \\
\quad \textbf{if } top(cst) = (Y_{cons} : d, n) \\
\quad\quad (\underline{pc+1}, afp, rc, \underline{cst'}, jpd, envpd, hp) \\
\quad \textbf{otherwise} \\
\quad\quad (\underline{0}, afp, \underline{-25}, cst, jpd, envpd, hp) \\
\quad \textbf{where } cst' = push(pop(cst), (Y_{num} : n))
\end{array}
$$

4.2.2 Zusammenfassung der Fehlercodes der Maschinenbefehle

In diesem Abschnitt sind die Fehlercodes, die bei Abarbeitung eines Maschinenbefehls auftreten können, noch einmal in tabellarischer Form aufgeführt, wobei zu jedem Fehlercode eine kurze Erläuterung steht. Diese Auflistung hat aber mehr Überblicks– und Nachschlagecharakter und kann vom Leser getrost übergangen werden.

0 Fehlerfreie Ausführung.

-1 Fehler bei der Ausführung des jump_on_true–Befehls: Als oberstes Element befand sich auf dem Berechnungsstack kein Wertelement der Form $(Y_{bool} : b)$.

-2 Fehler bei der Ausführung des jump_idx–Befehls: Oben auf dem Berechnungsstack lag kein Element der Form $(Y_{num} : n)$.

-3 Fehler bei der Ausführung des ret–Befehls: Der Sprungkeller war leer.

-4 Fehler bei der Ausführung des ret_param–Befehls: Der Sprungkeller oder der Umgebungskeller war leer.

-5 Fehler bei der Ausführung eines arithmetischen oder vergleichenden Befehls: Auf dem Berechnungsstack befanden sich nicht mindestens zwei Elemente oder eines der beiden obersten Stackelemente hatte nicht die Form $(Y_{num} : n)$.

-6 Fehler bei Ausführung des div– oder div_int–Befehls: Der Divisor war 0, so daß die Berechnung zu einer Division durch 0 führt.

-7 Fehler bei der Ausführung eines logischen Befehls: Auf dem Berechnungsstack befanden sich nicht mindestens zwei Elemente oder eines der beiden obersten Stackelemente war nicht von der Form $(Y_{bool} : n)$.

-8 Fehler bei der Ausführung eines Tests auf Gleichheit bzw. Ungleichheit: Auf dem Berechnungsstack befanden sich nicht mindestens zwei Elemente desselben Typs.

-9 Fehler bei der Ausführung des Befehls set_AFP k: Die Höhe des Berechnungsstacks minus $(k + 1)$ ergibt einen negativen Wert.

-10 Fehler bei der Ausführung des Befehls push CSTACK[AFP]: Das j–te Element des Berechnungsstacks ist nicht definiert, wobei j der Wert des Kellerrahmen–Zeigers ist.

-11 Fehler bei der Ausführung des Befehls push_adr adr ENV: Der Umgebungskeller war leer.

-12 Fehler bei der Ausführung des Befehls make_node k: Auf dem Berechnungsstack lagen weniger als $(k+1)$ Elemente.

-13 Fehler bei der Ausführung des deref_node–Befehls: Das oberste Element des Berechnungsstacks war kein Verweiselement $(P : p)$ oder das von p referenzierte Objekt war kein Knotenelement.

-14 Fehler bei der Ausführung des Befehls extract k: Das oberste Element des Berechnungsstacks war kein Verweiselement $(P : p)$,das von p referenzierte Objekt war kein Knotenelement oder der Wert für k war außerhalb des zulässigen Bereichs.

-15 Fehler bei der Ausführung des Befehls get_son_nr: Auf dem Berechnungsstack lagen nicht mindestens zwei Verweiselemente oder das vom obersten Verweiselement referenzierte Haldenobjekt war kein Knotenelement.

-16 Fehler bei der Ausführung des make_stack–Befehls: Der Berechnungsstack war entweder leer oder das oberste Element war kein Verweiselement.

-17 Fehler bei der Ausführung des make_copy–Befehls: Der Berechnungsstack war entweder leer, das oberste Element war kein Verweiselement $(P : p)$ oder das von p referenzierte Haldenobjekt war kein Pfadverweiselement.

-18 Fehler bei der Ausführung des heap_top– oder heap_pop–Befehls: Oben auf dem Berechnungsstack lag kein Verweiselement $(P : p)$, das von p referenzierte Haldenobjekt war kein Pfadverweiselement $(V : st)$ oder st war gleich dem leeren Keller.

-19 Fehler bei der Ausführung des heap_push–Befehls: Oben auf dem Berechnungsstack lagen keine zwei Elemente e_1 und e_2, e_1 war nicht von der Form $(Y_{num} : n)$ oder e_2 nicht von der Form $(P : p)$ oder das von p referenzierte Haldenobjekt war kein Pfadverweiselement.

-20 Fehler bei der Ausführung des Befehls eval q: Die $(a+1+q)$–te Stelle des Berechnungsstack war undefiniert, wobei a der Wert des Kellerrahmen–Zeigers war.

-21 Fehler bei des Ausführung des Befehls eval_param q: Der Umgebungskeller war leer oder die $(e.1+1+q)$–te Stelle des Berechnungsstack war undefiniert, wobei $e.1$ der Wert des obersten Elements des Umgebungskellers ist.

-22 Fehler bei der Ausführung des Befehls write_back q: Der Berechnungsstack war leer oder die $(a+1+q)$–te Stelle des Berechnungsstacks war undefiniert, wobei a der Wert des Kellerrahmen–Zeigers ist.

-23 Fehler bei der Ausführung des dup–Befehls: Der Berechnungsstack war leer.

-24 Fehler bei der Ausführung des Befehls cleanup: Die a–te Stelle im Berechnungsstack war nicht definiert, wobei a der Wert des Kellerrahmen–Zeigers ist, oder an der $(a+1)$–ten Stelle im Berechnungsstack war kein Wertelement der Form $(Y_{num} : n)$.

-25 Fehler bei der Ausführung des cons_to_num–Befehls: Der Berechnungsstack war leer oder das oberste Element hatte nicht die Form $(Y_{cons} : c)$.

4.3 Iterationssemantik und Berechnungsergebnis

Nachdem wir die Wirkungsweise jedes einzelnen Befehls auf eine Maschinenkonfiguration genau beschrieben haben, führen wir nun die Wirkung eines Maschinenprogramms auf eine Maschinenkonfiguration ein.

Definition 4.12 (Iterationssemantik)

Die *Iterationssemantik* der Maschine NoMach ist eine Funktion

$$\mathcal{I}[\![\ \cdot\ ]\!] : LNoMach(\Delta) \to (STATE \to STATE)$$

mit $\mathcal{I}[\![\, PStore\,]\!](pc, afp, rc, cst, jpd, envpd, hp)$

$$= \begin{cases} \mathcal{I}[\![\, PStore\,]\!](\mathcal{C}[\![\, PStore(pc)\ ]\!](pc, afp, rc, cst, jpd, envpd, hp)), \\ \quad \textbf{falls } PStore(pc) \text{ definiert und } pc \neq 0 \\ (pc, afp, rc, cst, jpd, envpd, hp), \ \textbf{sonst} \end{cases}$$

wobei $PStore \in LNoMach(\Delta)$ und $(pc, afp, rc, cst, jpd, envpd, hp) \in STATE$. □

Um nun das Berechnungsergebnis einer NoMach anzugeben, müssen wir nur noch die Anfangskonfiguration, mit der die Abarbeitung des Maschinenprogramms beginnt, festlegen. Das Berechnungsergebnis ist dann die Maschinenkonfiguration, die sich durch Anwendung der Iterationssemantik auf die Anfangskonfiguration ergibt. In der Anfangskonfiguration sind naheliegenderweise der Berechnungsstack und die beiden Keller leer. Außerdem befindet sich auf der Halde kein Element, für alle $n \in \mathbb{N}$ ist also $hp(n)$ undefiniert. Der Programmzähler hat in der Anfangskonfiguration den Wert 1, damit die Programmausführung mit dem ersten Befehl des Maschinenprogramms beginnen kann. Der Wert des Rückgabewert–Registers wird auf 0 gesetzt, da zu Beginn der Programmausführung kein Fehler vorliegt. Die Festlegung des Wertes des Kellerrahmen–Zeigers kann willkürlich erfolgen. Wir haben uns für den Wert 1 entschieden.

Definition 4.13 (Programmsemantik, Berechnungsergebnis)

Sei $\mathcal{M} = (\Delta,\ PStore,\ STATE)$ eine NoName–Maschine. Die Programmsemantik von $\mathcal{M}$ ist eine Funktion

$$\mathcal{P} : LNoMach(\Delta) \to STATE$$

mit $\quad \mathcal{P}(PStore) = \mathcal{I}[\![\, PStore\,]\!](1, 1, 0, cst, \varepsilon, \varepsilon, hp)$
where $cst = (0, c)$ mit c beliebig.
für alle $p \in \mathbb{N}$ ist $hp[p]$ undefiniert

Das *Berechnungsergebnis von* $\mathcal{M}$ ist $\mathcal{P}(PStore)$. □

Bevor wir die Beschreibung der Zielmaschine beenden, wollen wir noch in einem umfangreichen Beispiel die Arbeitsweise einer NoMach verdeutlichen.

Beispiel 4.14 (Ablauf eines Maschinenprogramms)

In diesem Beispiel wollen wir den Ablauf des Maschinenprogramms, das in Abbildung 4.14 dargestellt ist, vorführen.

1:	jump 9	8:	ret
2:	eval 1	9:	push_const $(Y_{num} : 0)$
3:	push_const $(Y_{num} : 2)$	10:	push AFP
4:	mul	11:	push_const $(Y_{num} : 5)$
5:	eval 2	12:	push_const $(Y_{num} : 8.5)$
6:	add	13:	call 2
7:	cleanup	14:	stop

Abb. 4.14: Maschinenprogramm.

Die Berechnung startet, wie in Definition 4.13 festgelegt, auf der Maschinenkonfiguration

$$\begin{array}{llll} pc = 1 & afp = 1 & rc = 0 & cst = (0, c) \\ jpd = \varepsilon & envpd = \varepsilon & hp & \end{array}$$

wobei für alle $p \in I\!N$ gilt, daß $hp[p]$ undefiniert ist.

Der erste Maschinenbefehl, der abgearbeitet wird, ist folglich der Befehl an der Adresse 1, also jump 1. Dieser Maschinenbefehl setzt den Programmzähler auf den Wert 9, so daß der nächste zur Ausführung anstehende Befehl push_const $(Y_{num} : 0)$ ist.

Nach Ausführung der Maschinenbefehle mit den Adressen 9 bis 12 ergibt sich für den Berechnungsstack die in Abbildung 4.15 dargestellte Konfiguration.

4	$(Y_{num} : 8.5)$
3	$(Y_{num} : 5)$
2	$(Y_{num} : 1)$
1	$(Y_{num} : 0)$

Abb. 4.15: Berechnungsstack nach Abarbeitung des Maschinenbefehls an der Adresse 12.

Wie aus der Abbildung 4.15 zu ersehen ist, befindet sich im 2. Element des Berechnungsstacks das Wertelement $(Y_{num} : 1)$. Dieses Wertelement wurde vom Maschinenbefehl push AFP erzeugt. Der Wert 1 kommt daher, daß der Wert des Kellerrahmen–Zeigers, der eben durch diesen Maschinenbefehl auf dem Berechnungsstack abgelegt wird, 1 ist.

Der nun auszuführende Maschinenbefehl call 2 setzt den Programmzähler auf den Wert 2 und verändert den Sprungkeller zu $jpd = 14$.

Die bisher auf den Berechnungsstack aufgelegten Wertelemente wollen wir als Kellerrahmen verstehen. Die Elemente 1 und 2 bilden die organisatorischen Zellen und die Elemente 3 und 4 die beiden Funktionsparameter.

Mit dem nun zur Ausführung anstehenden Maschinenbefehl eval 1 wollen wir den ersten Funktionsparameter, also $(Y_{num} : 5)$ oben auf den Berechnungsstack kopieren. Betrachten wir dazu die auf Seite 63 angegebene Befehlssemantik des eval–Befehls. Das Element, das durch den Maschinenbefehl betrachtet wird, befindet sich im Berechnungsstack an der Position

$$\underbrace{\quad 1 \quad}_{\text{Wert von AFP}} + \underbrace{\quad 1 \quad}_{\text{Konstante}} + \underbrace{\quad 1 \quad}_{\text{Parameter von eval}} \qquad = \qquad 3.$$

Da dieses Element kein Funktionselement ist, kommt in der inneren if–Bedingung der otherwise–Fall zum Tragen und das 3. Berechnungsstack–Elements wird kopiert und auf den Berechnungsstack aufgelegt. Wir erhalten nun die in Abbildung 4.16 dargestellte Konfiguration des Berechnungsstacks.

5	$(Y_{num} : 5)$
4	$(Y_{num} : 8.5)$
3	$(Y_{num} : 5)$
2	$(Y_{num} : 1)$
1	$(Y_{num} : 0)$

Abb. 4.16: Berechnungsstack nach Abarbeitung des Maschinenbefehls eval 1.

Die auszuführenden Maschinenbefehle an den Adressen 3 bis 6 führen nun eine arithmetische Berechnung auf dem Berechnungsstack durch. Dabei wird der Wert

$$(1.\text{Parameter} * 2) + 2.\text{Parameter}$$

berechnet. Dabei ergibt sich der Wert $(5 * 2) + 8.5 = 18.5$. Nach Ausführung des Maschinenbefehls add ergibt sich also die in Abbildung 4.17 dargestellte Konfiguration des Berechnungsstacks.

5	$(Y_{num} : 18.5)$
4	$(Y_{num} : 8.5)$
3	$(Y_{num} : 5)$
2	$(Y_{num} : 1)$
1	$(Y_{num} : 0)$

Abb. 4.17: Berechnungsstack nach Abarbeitung des Maschinenbefehls add.

Der Befehl cleanup bewirkt nun, daß der aktuelle Kellerrahmen aufgegeben wird und das Ergebnis der Berechnung — also das Wertelement $(Y_{num} : 18.5)$ — im Berechnungsstack an die a–te Stelle verschoben wird, wobei a der aktuelle Wert des Kellerrahmen–Zeigers — hier also 1 — ist. Wir erhalten also die in Abbildung 4.18 dargestellte Konfiguration des Berechnungsstacks.

1	$(Y_{num} : 18.5)$

Abb. 4.18: Berechnungsstack nach Abarbeitung des Maschinenbefehls cleanup.

Der Maschinenbefehl `ret` an der Adresse 8 bewirkt nun eine Rückkehr aus dem aktuell bearbeiteten Unterprogramm. Die Rücksprungadresse liegt oben auf dem Sprungkeller. Der Programmzähler wird also auf den Wert 14 gesetzt und der Sprungkeller ändert sich zu $jpd = \varepsilon$.

Der letzte auszuführende Befehl, nämlich `stop`, bewirkt das Anhalten der Programmausführung. Die Konfiguration, die sich nach dieser Befehlsausführung ergibt, ist

$$\begin{array}{llll} pc = 0 & afp = 1 & rc = 0 & cst = (1, c) \\ jpd = \varepsilon & envpd = \varepsilon & hp & \end{array}$$

wobei $c[1] = (Y_{num} : 18.5)$ ist und für alle $p \in \mathbb{N}$ gilt, daß $hp[p]$ undefiniert ist. Diese Maschinenkonfiguration stellt auch das Berechnungsergebnis dar. □

An diesem Punkt ist die Definition der abstrakten NoName–Maschine abgeschlossen. Wir möchten an dieser Stelle noch darauf hinweisen, daß die Definition der abstrakten Maschine im Zusammenspiel mit der Übersetzung, die im folgenden Kapitel erläutert wird, betrachtet werden muß. Im Entwicklungsprozeß von Maschine und Übersetzer gab es natürlich immer Wechselwirkungen, die das jeweilige Aussehen, die jeweilige Definition beeinflußten. Dennoch hoffen wir, daß wir die wesentlichen Konzepte, die hinter einer NoMach stehen, verständlich machen konnten.

Kapitel 5

Die Übersetzung von NoName*

Nachdem wir in den vorangegangenen Kapiteln die zu übersetzende Sprache NoName* formal eingeführt und die Zielmaschine beschrieben haben, kommen wir nun zur Definition der Übersetzungsfunktionen, die aus einem gegebenen NoName*–Programm das zugehörige Maschinenprogramm erzeugen.

In Abschnitt 5.1 skizzieren wir die Ideen, die der Übersetzung zugrundeliegen. In den folgenden Abschnitten geben wir dann die einzelnen Übersetzungsfunktionen an, wobei wir auch hier immer wieder die zugrundeliegenden Konzepte und Gedanken erläutern.

Die Ideen, die hinter der vorgestellten Übersetzung stehen, sind entscheidend von dem Übersetzer für funktionale Sprachen, welcher in [WM92] vorgestellt wird, geprägt. Aber auch Konzepte zur Übersetzung imperativer Sprachen, wie sie in [ASU86] oder [WM92] vorgestellt werden, haben die hier vorgestellte Übersetzung beeinflußt.

5.1 Grundlegende Ideen bei der Übersetzung

Im folgenden wollen wir die grundlegenden Ideen, die bei der Übersetzung zum Tragen kommen, vorstellen. Als wiederkehrendes Beispiel, an dem wir bestimmte Phänomene diskutieren wollen, verwenden wir das in Abbildung 5.1 angegebene NoName*–Programm zur Berechnung der Fakultät der Zahl 3.

```
GOAL fak(3)
PROGRAM fakultaet;
  FUNC_SPEC
    fak : CARDINAL -> CARDINAL.
    fak(x) = 1, IF x <= 1,
           = fak(x-1) * x, OTHERWISE;
  END_FUNC_SPEC
END_PROGRAM fakultaet.
```

Abb. 5.1: NoName*–Programm zur Berechnung von `fak(3)`.

Wie wir bereits in Abschnitt 4.2 angedeutet haben, legen wir bei jedem Funktionsaufruf einen sogenannten Kellerrahmen auf dem Berechnungsstack an. In Abbildung 5.2 ist nochmals der schematische Aufbau eines Kellerrahmens dargestellt.

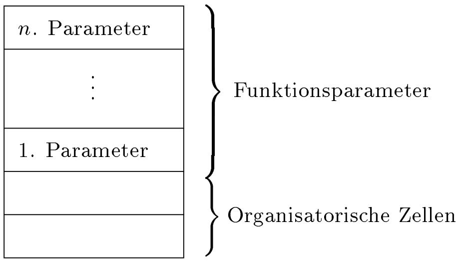


Abb. 5.2: Aufbau eines Kellerrahmens.

Der Beginn des aktuellen Kellerrahmens wird durch den Kellerrahmen–Zeiger festgelegt. Bezogen auf unser Beispiel der Fakultätsfunktion hat ein Kellerrahmen hier also 3 Zellen, wobei in der dritten Zelle der jeweilige Funktionsparameter liegt. Die untere organisatorische Zelle spielt nur bei Funktionen, deren Funktionsergebnis mehrere Komponenten hat, eine Rolle. Die obere organisatorische Zelle dient der Verkettung der Kellerrahmen und zeigt auf den jeweils vorhergehenden Kellerrahmen. In Abbildung 5.3 haben wir eine solche Verkettung anhand der rekursiven Aufrufe der Fakultätsfunktion dargestellt. Der aktuelle Kellerrahmen in diesem Beispiel gehört zum Funktionsaufruf `fak(1)`. Deshalb zeigt hier der Kellerrahmen–Zeiger AFP auf den Beginn des zugehörigen Kellerrahmens.

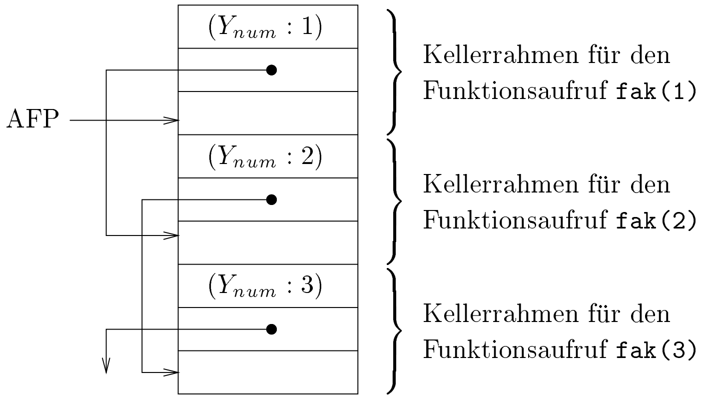


Abb. 5.3: Verkettete Kellerrahmen der Funktionsaufrufe `fak(1)`, `fak(2)` und `fak(3)`.

Ist die Bearbeitung eines Funktionsaufrufs abgeschlossen, so wird das Funktionsergebnis an die Stelle der unteren organisatorischen Zelle kopiert und anschließend der aktuelle Kellerrahmen bis auf diese Zelle aufgegeben. Dabei wird mit Hilfe der oberen organisatorischen Zelle der Kellerrahmen–Zeiger auf den Anfang des vorhergehenden Kellerrahmens gesetzt. In Abbildung 5.4 haben wir nun in (a) die Situation zu Beginn des Funktionsaufrufs `fak(1)` dargestellt. In (b) findet sich die Situation nach Freigabe des obersten Kellerrahmens. Das oben auf dem Berechnungsstack liegende Wertelement $(Y_{num} : 1)$ ist dabei das Ergebnis des betrachteten Funktionsaufrufs `fak(1)`. Das Umkopieren und die Freigabe des Kellerrahmens erfolgt durch den Maschinenbefehl cleanup, der eigentliche Rücksprung durch den ret–Befehl.

Bei Funktionen, die via IF oder mittels Pattern Matching definiert sind, muß in der Regel vor der eigentlichen Bestimmung des Funktionsergebnisses festgestellt werden, welcher

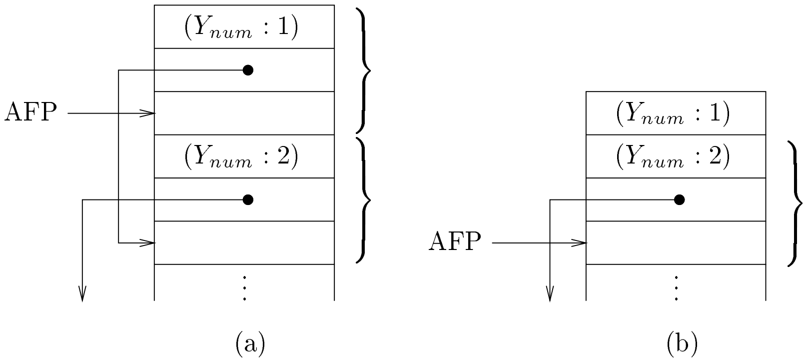


Abb. 5.4: Situation zu Beginn und nach Beendigung des Funktionsaufrufs `fak(1)`.

der angegebenen Fälle anzuwenden ist. Bezogen auf unser Beispiel der Fakultätsfunktion erhalten wir also den in Abbildung 5.5 dargestellten Aufbau des Maschinenprogramms der Funktion `fak`.

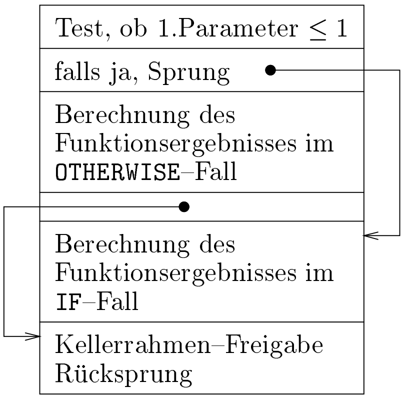


Abb. 5.5: Aufbau des Maschinenprogramms der Funktion `fak`.

Um allgemein einen Test durchführen zu können, müssen wir den zum Test gehörenden Ausdruck auswerten. Jedem Infixoperator, der in einem NoName*–Programm vorkommen kann, entspricht ein Maschinenbefehl. Die Erzeugung der Maschinenbefehle zu einem Ausdruck kann also in naheliegender Weise erfolgen. Bei der Übersetzung des Ausdrucks `x <= 1` beispielsweise erhalten wir die Maschinenbefehle

eval 1
push_const $(Y_{num} : 1)$
leq.

Damit kommen wir nun zu den Parameterzugriffen. Die aktuellen Funktionsparameter liegen ja im aktuellen Kellerrahmen ab der dritten Zelle aufwärts. Um also auf den n–ten Parameter zuzugreifen, müssen wir relativ zum Wert des Kellerrahmen–Zeigers die $(n + 2)$–te Zelle betrachten. Diese Relativadressierung wird durch den Maschinenbefehl eval n übernommen.

Soll ein Funktionsaufruf übersetzt werden, so müssen wir Maschinenbefehle erzeugen, die einen neuen Kellerrahmen anlegen, die Parameterwerte eintragen und die betreffende Funktion anspringen. Die folgende Abbildung 5.6 zeigt anhand des Funktionsaufrufs `fak(x-1)`, welche Befehle hierzu notwendig sind.

push_const $(Y_{num} : 1)$	erste organisatorische Zelle belegen
push AFP	Verweis auf momentan noch aktuellen Kellerrahmen ablegen
trans[x − 1]	Anlegen der Parameter, hier `x-1`
set_AFP 1	Kellerrahmen–Zeiger auf den Anfang des neuen Kellerrahmens setzen; die 1 bedeutet, daß der Kellerrahmen einen Funktionsparameter enthält
call `fak`	Aufruf der Funktion

Abb. 5.6: Schema der Maschinenbefehle eines Funktionsaufrufs.

Hier kommen wir nun zu einem Punkt, an dem die Auswertungsstrategie, die für eine Funktion festgelegt wurde, eine Rolle spielt. Soll die angerufene Funktion f nach der Strategie Call–by–Value ausgewertet werden, so müssen die Parameter der Funktion f vor dem Aufruf ausgerechnet werden. Diese Aufgabe übernimmt die aufrufende Funktion.

Soll die aufzurufende Funktion f aber nach der Strategie Call–by–Name ausgewertet werden, so dürfen die Funktionsparameter noch nicht ausgerechnet werden. Wie aber kann dann f seine Parameterwerte erhalten, falls sie diese benötigt? Wir lösen dieses Problem, indem wir nicht die Parameterwerte in die entsprechenden Zellen des Kellerrahmens legen, sondern Adressen auf Routinen. Durch Anspringen dieser Routinen kann f dann die benötigten Parameterwerte ermitteln. Diese Adresse wird, wie wir bereits in Kapitel 4 gesehen haben, in einem Funktionselement abgespeichert.

Nehmen wir nun also an, daß unsere Beispielsfunktion `fak` mit der Strategie Call–by–Name ausgewertet werden soll. Wir erhalten dann für den Funktionsaufruf `fak(x-1)` die in Abbildung 5.7 angegebenen Maschinenbefehle.

push_const $(Y_{num} : 1)$	$label_1$:	eval_param 1
push AFP		push_const $(Y_{num} : 1)$
push_adr $label_1$ AFP		sub
set_AFP 1		ret_param
call `fak`		

Abb. 5.7: Maschinenbefehle für den Funktionsaufruf `fak(x-1)`, falls die Funktion `fak` Call–by–Name ausgewertet werden soll.

Mit der Routine, die an der Adresse $label_1$ beginnt, kann der benötigte Funktionsparameter bestimmt werden. Hierbei fällt auf, daß für den Parameterzugriff in dieser Routine nicht der eval–Befehl, sondern der eval_param–Befehl Verwendung findet. Dies liegt daran, daß der Parameterzugriff hier nicht bezogen auf den aktuellen Kellerrahmen erfolgen soll, sondern bezogen auf den Kellerrahmen zum Zeitpunkt der Anlage des Funktionselements. Der Parameter AFP im Maschinenbefehl push_adr $label_1$ AFP bewirkt, daß ein Verweis auf diesen Kellerrahmen ebenfalls im angelegten Funktionselement gespeichert wird.

Wie weiß aber der eval_param–Befehl, auf welchen Funktionsrahmen sich der Zugriff zu beziehen hat? Oder, mit anderen Worten gesprochen, wie gelangt die benötigte Angabe vom Funktionselement zum eval_param–Befehl? Diese Kommunikation erfolgt über den Umgebungskeller. Stößt der eval– oder eval_param–Befehl bei der Auswertung eines Parameters auf ein Funktionselement, so legt er den darin enthaltenen Kellerrahmen–Verweis auf dem Umgebungskeller ab und springt dann die angegebene Routine als Unterprogramm an. Der Parameterzugriff des eval_param–Befehls erfolgt dann relativ zu dem Wert des obersten Umgebungskeller–Elements. Ist die Abarbeitung des Unterprogramms abgeschlossen, so bewirkt der Rücksprungbefehl ret_param, daß der auf dem Umgebungskeller aufgelegte Verweis wieder entfernt wird.

Diesen Mechanismus wollen wir nun anhand eines Beispiels verdeutlichen. Betrachten wir dazu das Maschinenprogramm in Abbildung 5.8, das die vollständige Übersetzung des NoName*–Programms aus Abbildung 5.1 darstellt.

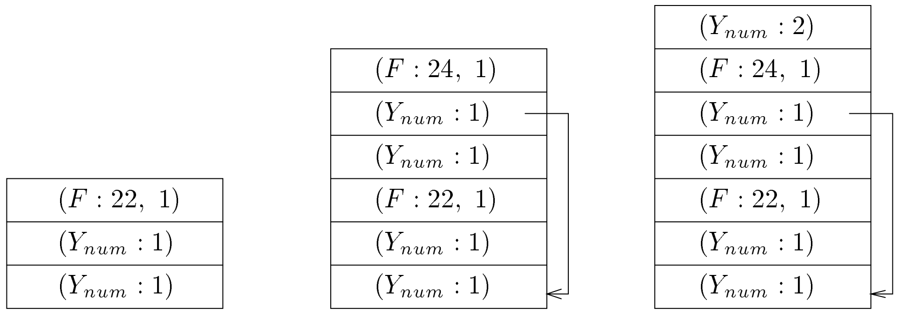
Anfrage `fak(3)`
  1: push_const ($Y_{num}: 1$)
  2: push AFP
  3: push_adr 22 AFP
  4: set_AFP 1
  5: call 7
  6: stop
Funktion `fak`:
Test auf `x <= 1`
  7: eval 1
  8: push_const ($Y_{num}: 1$)
  9: leq
  10: jump_on_true 19
Alternative `OTHERWISE`
  11: push_const ($Y_{num}: 1$)
  12: push AFP
  13: push_adr 24 AFP
  14: set_AFP 1
  15: call 7
  16: eval 1
  17: mul
  18: jump 20
Alternative `x <= 1`
  19: push_const ($Y_{num}: 1$)
Aufräumen
  20: cleanup
  21: ret

Parameterroutine für `fak(3)`
  22: push_const ($Y_{num}: 3$)
  23: ret_param

Parameterroutine für `fak(x-1)`
  24: eval_param 1
  25: push_const ($Y_{num}: 1$)
  26: sub
  27: ret_param

Abb. 5.8: Maschinenprogramm zur Berechnung von `fak(3)`.

In Abbildung 5.9 haben wir einige Konfigurationen des Berechnungsstacks im Ablauf der Ausführung des Maschinenprogramms dargestellt. Die Konfiguration (a) ergibt sich direkt nach der Ausführung des Unterprogrammaufrufs an Programmadresse 5. Der Maschinenbefehl eval 1 bewirkt durch Anspringen der Routine an der Adresse 22 das Auflegen des Wertelements ($Y_{num}: 3$). Der Test auf ≤ 1 ergibt natürlich falsch, so daß wir die Alternative `OTHERWISE` ausführen. Hier legen wir einen neuen Kellerrahmen für den Funktionsaufruf `fak(3-1)` auf, wobei wir den Parameterwert noch nicht bestimmen. In (b)



(a) (b) (c)

Abb. 5.9: Konfigurationen während der Abarbeitung des Maschinenprogramms aus Abbildung 5.8.

haben wir die Konfiguration nach Ausführung des Funktionsaufrufs an Programmadresse 15 dargestellt. Der nun folgende Befehl eval 1 bewirkt ein Anspringen der Routine an der Programmadresse 24, wobei zuvor der Wert 1 auf den Umgebungskeller gelegt wurde. Der Befehl eval_param 1 greift nun auf das erste Parameterelement bezogen auf den unteren Kellerrahmen zu und springt damit die Routine an Adresse 22 an, die dann $(Y_{num} : 3)$ zurückliefert. Anschließend wird dieses Ergebnis um 1 dekrementiert und die Berechnung des Parameterwerts ist damit beendet (Konfiguration (c)). An dieser Stelle wollen wir nun dieses Beispiel beenden.

Als letzten Punkt wollen wir noch kurz die Probleme und Lösungen ansprechen, die sich im Zusammenhang mit Funktionen ergeben, die ein mehrstelliges Ergebnis zurückliefern. Betrachten wir dazu die Funktionen `Tupel` und `Proj`, deren Definitionsgleichungen sich in Abbildung 5.10 finden.

```
Tupel() = (17.0/0.0, 2.0);
Proj(x,y) = y;
```

Abb. 5.10: Funktionsgleichungen für `Tupel` und `Proj`.

Die beiden Funktionen sollen mit der Auswertungsstrategie Call–by–Name ausgewertet werden. Für die Behandlung mehrstelliger Funktionen gibt es zwei Ansatzpunkte:

- Beim Aufruf einer Funktion werden alle Ergebniskomponenten auf einmal zurückgeliefert. In diesem Fall führt die Anfrage `Proj(Tupel())` zu einer Division durch Null, da durch Zugriff auf die zweite Ergebniskomponente von `Tupel` auch die erste Ergebniskomponente mitbestimmt wird.
- Eine Funktion, die ein n–stelliges Ergebnis liefert, wird aufgefaßt wie n Funktionen, die ein jeweils einstelliges Ergebnis zurückliefern. In diesem Fall führt die Anfrage `Proj(Tupel())` zum gewünschten Ergebnis 2.0.

Wir haben uns in dieser Arbeit für die zweite Variante entschieden. Diese Variante macht es aber erforderlich, der aufgerufenen Funktion mitzuteilen, welche Komponente des Funktionsergebnisses als Resultat des Funktionsaufrufs erwartet wird. Diese Angabe schreiben wir in die untere organisatorische Zelle eines jeden Kellerrahmens.

Bei einer Funktion, die ein mehrstelliges Ergebnis liefert, muß dann zwischen der Bestimmung der anzuwendenden Alternative und der Bestimmung des Funktionsergebnisses eine Sprungtabelle stehen, die in Abhängigkeit des Wertes der unteren organisatorischen Zelle die Routine zur Auswertung der gewünschten Ergebniskomponente anspringt.

Damit haben wir die Beschreibung der grundlegenden Ideen zur Übersetzung abgeschlossen. Im folgenden wenden wir uns nun dem eigentlichen Übersetzungsvorgang zu.

5.2 Vorüberlegungen zum Übersetzungsvorgang

Bei der Übersetzung gehen wir von einem korrekten NoName*–Programm P aus. Um den Übersetzungsvorgang ein wenig zu vereinfachen, fordern wir weiterhin, daß P zwei Eigenschaften erfüllt. Diese Eigenschaften sind:

- Die einzelnen Variablen in der linken Seite einer jeder Funktionsgleichung seien $x_1, x_2, \ldots$, d.h. aus den Variablennamen kann die Parameterposition bestimmt werden. An die Variablennamen in den gemischen Ausdrücken eines Parameters einer Funktion, die mittels Pattern Matching definiert ist, wird keine Bedingung gestellt. Falls diese Eigenschaft nicht vorliegt, so kann diese jedoch durch einfaches Umbenennen erfüllt werden, wie folgendes Beispiel zeigt.

  Beispiel 5.1 (Umbenennen von Parametern)

  Gegeben seien die aus der Einleitung bekannten Funktionsgleichungen in Abbildung 5.11, die die oben geforderte Eigenschaft nicht erfüllen.

```
gen_baum(count,n,b)
   = Blatt(n), IF count = 0,
   = Knoten(gen_baum(count - 1,n,b),Blatt(n),b), OTHERWISE;
get_zahl (Blatt(n)) = n;
f_up(<z:Knoten>.1,n) = n + f_up(<z>,n);
```

  Abb. 5.11: Funktionsgleichungen mit ursprünglichen Variablennamen.

  Durch einfaches Umbenennen kann die oben geforderte Eigenschaft erfüllt werden. In Abbildung 5.12 sind die umbenannten Funktionsgleichungen wiedergegeben.

```
gen_baum(x1,x2,x3)
   = Blatt(x2), IF x1 = 0,
   = Knoten(gen_baum(x1 - 1,x2,x3),Blatt(x2),x3), OTHERWISE;
get_zahl (Blatt(n)) = n;
f_up(<z:Knoten>.1,x2) = x2 + f_up(<z>,x2);
```

  Abb. 5.12: Funktionsgleichungen mit umbenannten Variablennamen.

  In der Gleichung für `get_zahl` wurde keine Umbenennung durchgeführt, da die einzige Variable, die in dieser Gleichung enthalten ist, in einem gemischten Ausdruck auftaucht. □

- Die Funktionen, die via IF definiert sind, tragen die Namen $f_1, f_2, \ldots$, die Funktionen, die mittels Pattern Matching definiert sind haben die Namen $g_1, g_2, \ldots$, die synthetischen Funktionen werden mit $s_1, s_2, \ldots$ bezeichnet und die inheriten Funktionen mit $h_1, h_2, \ldots$. Falls diese Eigenschaft nicht erfüllt ist, kann diese, wie auch eben für Variablennamen gezeigt, durch einfaches Umbenennen sichergestellt werden.

Wie wir in Kapitel 4 gesehen haben, arbeiten die dort vorgestellten Maschinenbefehle mit absoluten Adressen. Um bei der Übersetzung den erzeugten Maschinenbefehlen die korrekten Adressen mitzugeben, müssen die Übersetzungsfunktionen als Parameter auch Adressen besitzen. Die Verwendung von absoluten Adressen während der Übersetzung erschwert aber, wie das Beispiel 5.2 zeigt, den Übersetzungsvorgang.

Beispiel 5.2 (Verwendung absoluter Adressen bei der Übersetzung)

Betrachten wir den zu übersetzenden Ausdruck $(e_1 + e_2)$. Die nächste zu vergebende Programmadresse sei adr. Bei der Übersetzung des Ausdrucks wird nun zuerst e_1 übersetzt, dann e_2 übersetzt und zuletzt der add–Befehl erzeugt. Die Übersetzungsfunktion für einen Ausdruck $(e_1 + e_2)$ ist also

$$\begin{aligned} trans[(e_1+e_2)](adr) = \quad & trans[e_1](adr) \\ & trans[e_2](adr+k_1) \\ (adr+k_1+k_2): \ & \mathsf{add} \end{aligned}$$

Hier ergibt sich jedoch ein Problem: Die Werte für k_1 und k_2, die angeben, wie lange die Maschinenbefehlsfolgen für e_1 bzw. e_2 sind, sind unbekannt. Um also die Übersetzung $trans[e_2](adr + k_1)$ durchführen zu können, muß zuerst $trans[e_1](adr)$ bestimmt werden und die Anzahl der dadurch erzeugten Maschinenbefehlsfolge ermittelt werden. □

Da diese Vorgehensweise die Übersetzung unnötig erschwert, verwenden wir bei der Übersetzung sogenannte baumstrukturierte Adressen.

Definition 5.3 (Baumstrukturierte Adresse)

Die *Menge der baumstrukturierten Adressen*, bezeichnet mit B_Adr, ist die Menge $\mathbb{N}_0{}^*$, also die Menge aller Sequenzen über der Menge der natürlichen Zahlen. Eine baumstrukturierte Adresse wird durch $a_1.a_2. \ \ldots \ .a_n$ notiert, wobei $n \geq 0$ und $a_i \in \mathbb{N}_0$ für alle $i \in [n]$. □

Die Verwendung von baumstrukturierten Adressen hat nun den Vorteil, daß die einzelnen Übersetzungsfunktionen unabhängig voneinander angewandt werden können.

Beispiel 5.4 (Verwendung baumstrukturierter Adressen bei der Übersetzung)

Betrachten wir nochmals die Übersetzung des Ausdrucks $(e_1 + e_2)$ aus Beispiel 5.2. Bei der Verwendung von baumstrukturierten Adressen ändert sich die Übersetzungsfunktion für $(e_1 + e_2)$ zu

$$\begin{aligned} trans[(e_1+e_2)](badr) = \ & trans[e_1](badr.1) \\ & trans[e_2](badr.2) \\ & \mathsf{add} \end{aligned}$$

wobei *badr* eine baumstrukturierte Adresse ist. Die Übersetzung von e_2 benötigt nun keinerlei Angaben über das Ergebnis der Übersetzung von e_1, so daß die Übersetzung vollständig entkoppelt durchgeführt werden kann. Ein Adressenkonflikt, also die mehrfache Verwendung derselben Adresse, ist ausgeschlossen. □

Nun enthält aber das erzeugte Maschinenprogramm baumstrukturierte anstelle von absoluten Adressen. Die notwendige Umwandlung kann aber mittels der folgenden Schritte durchgeführt werden.

1. Setze die Menge *ADR* gleich der leeren Menge.
2. Numeriere die Befehle des Maschinenprogramms mit baumstrukturierten Adressen der Reihe nach durch. Merke dabei die Paare $(badr, adr)$ in der Menge *ADR*, wobei *badr* die baumstrukturierte Adresse des betrachteten Maschinenbefehls ist und *adr* die zugewiesene Nummer, also die absolute Adresse.
3. Ersetze jeden Maschinenbefehl b, der eine baumstrukturierte Adresse *badr* als Argument hat, durch den Maschinenbefehl b', der die absolute Adresse *adr* als Argument hat, wobei $(badr, adr)$ in *ADR* ist.

Dieser Vorgang der Umwandlung von baumstrukturierten Adressen in absolute Adresse wird *Linearisierung* genannt.

Beispiel 5.5 (Linearisieren eines Maschinenprogramms)

Betrachten wir das folgende Maschinenprogramm mit baumstrukturierten Adressen (Abbildung 5.13).

```
1.2:  jump_on_true 1.1
      push_const (Y_bool : TRUE)
      jump 1.2
1.1:  ret
```

Abb. 5.13: Maschinenprogramm mit baumstrukturierten Adressen.

Die Menge *ADR* enthält nach Durchführung von Schritt 2 die beiden Paare (1.2,1) und (1.1,4). Nach Abarbeitung der obigen Schritte erhalten wir also das Maschinenprogramm, das in Abbildung 5.14 wiedergegeben ist.

```
1:  jump_on_true 4
2:  push_const (Y_bool : TRUE)
3:  jump 1
4:  ret
```

Abb. 5.14: Maschinenprogramm mit absoluten Adressen. □

Bei der Übersetzung eines Funktionsaufrufs müssen wir die Adresse kennen, an der die aufzurufende Funktion steht, um diese Adresse dem erzeugten call–Befehl als Parameter mitzugeben. Um auch diese Problematik möglichst einfach zu lösen, führen wir eine sogenannte Adressfunktion ein. Diese Funktion weist jedem Funktionssymbol eine eindeutige baumstrukturierte Adresse zu.

Definition 5.6 (Adressfunktion)

Sei F eine Menge von Funktionssymbolen. Eine *Adressfunktion für F* ist eine bijektive Funktion $adr\_fkt : F \rightarrow B\_Adr$, die jedem Funktionssymbol $f \in F$ eine eindeutige baumstrukturierte Adresse zuordnet, so daß gilt: Es gibt keine zwei Funktionssymbole f und f', so daß $adr\_fkt(f)$ ein Teil von $adr\_fkt(f')$ ist. □

Erinnern wir uns nun daran, daß die Wertelemente für Konstruktoren neben dem eigentlichen Konstruktor noch eine Zahl enthalten (siehe Definition 4.1, Seite 44). Diese Zahl benötigen wir, um eine numerische Kodierung des Konstruktors aufzunehmen. Den Sinn dieser Kodierung erläutert das nachfolgende Beispiel.

Beispiel 5.7 (Übersetzung einer Fallunterscheidung über Konstruktoren)

Betrachten wir die aus der Einleitung bekannten Gleichungen der Funktion `get_zahl`.

```
get_zahl (Knoten(t1,t2,b)) = 0
get_zahl (Blatt(n)) = n;
```

Abb. 5.15: Gleichungen der Funktion `get_zahl`.

Uns interessiert hier, wie wir die Fallunterscheidung möglichst effizient übersetzen. Zuerst müssen wir sicherlich den Wert des aktuellen Funktionsparameters ermitteln. Dieser Wert wird ein Konstruktorbaum sein. Von diesem Konstruktorbaum benötigen wir dann den Konstruktor d, der an dessen Wurzel steht. Wir haben nun zwei Möglichkeiten, mit Hilfe von d die passende Funktionsgleichung zu ermitteln:

- Wir vergleichen der Reihe nach d mit den Konstruktoren `Knoten` und `Blatt`. Im Falle von Gleichheit haben wir die passende Alternative gefunden.
- Wir führen mit Hilfe einer numerischen Kodierung für d direkt einen indizierten Sprung zur entsprechenden Alternative durch.

Hat der zugrundeliegende algebraische Datentyp θ viele verschiedene Konstruktoren, so wird die erste Möglichkeit schnell ineffizient. Der Zeitaufwand zum Ermitteln der passenden Alternative wächst hier linear mit der Größe von θ an. Die zweite Möglichkeit benötigt einen konstanten Zeitaufwand, nämlich den indizierten Sprung. □

Als Kodierung wählen wir eine Aufzählung aller Konstruktoren. Da bei der Fallunterscheidung auf einer Parameterposition immer nur ein algebraischer Datentyp beteiligt ist, kann diese Aufzählung separat für jeden algebraischen Datentyp erfolgen.

In Definition 5.8 formalisieren wir nun diese Überlegung. Erinnern wir uns hier noch einmal an die Definition 3.15 auf Seite 34, in der die Menge der Konstruktoren mit Resultattyp θ bezogen auf S und Δ, kurz $\Delta[\theta]_S$, eingeführt wurde.

Definition 5.8 (Konstruktor–Funktion)

Sei S eine Basis–Sortenmenge und Δ eine $(S^* \times \hat{S})$–sortierte Menge von Konstruktorsymbolen. Eine Funktion $\delta\_to\_n : \Delta \rightarrow I\!N_0$ heißt *Konstruktor–Funktion bezüglich S und* Δ, wenn sie für jedes $\theta \in \hat{S}$ eine Aufzählungsfunktion für $\Delta[\theta]_S$ ist und $\delta\_to\_n(root) = 0$ gilt. □

Das Wurzelsymbol *root* haben wir deshalb in der Definition mit berücksichtigt, weil wir im Zusammenhang mit tree transducern auch eine Kodierung für dieses Symbol benötigen. Durch das folgende Beispiel wollen wir die Definition 5.8 noch ein wenig verdeutlichen.

Beispiel 5.9 (Konstruktor–Funktion)

Sei S die Basissortenmenge $\{int, card, char, bool, real, Baum\}$ mit dem bereits aus der Einleitung bekannten algebraischen Datentyp *Baum*. Die Menge Δ der Konstruktorsymbole sei $\{\texttt{Knoten}^{(Baum\ Baum\ bool, Baum)}, \texttt{Blatt}^{(int, Baum)}\}$. Dann ist δ_to_n mit

$$\delta\_to\_n(\texttt{Knoten}) = 1,\ \delta\_to\_n(\texttt{Blatt}) = 2,\ \delta\_to\_n(root) = 0$$

eine Konstruktorfunktion bezüglich S und Δ. □

Nachdem wir nun die Ideen und Konzepte, die hinter dem Übersetzungsvorgang stehen, erläutert haben, geben wir in den folgenden Abschnitten die einzelnen Übersetzungsfunktionen an.

5.3 Übersetzung des Rumpfs eines NoName*–Programms

Ein NoName*–Programm P besteht, wie aus Kapitel 3 bekannt, aus einer Anfrage und einer Folge von Spezifikationen. Da die Ausführung eines jeden Maschinenprogramms an der Programmadresse 1 beginnt, sollten an dieser Stelle die Maschinenbefehle für die Anfrage stehen. Wir übersetzen also zuerst die Anfrage von P. Danach folgt der Maschinenbefehl `stop`, um die Programmausführung nach Abarbeitung der Anfrage zu beenden. Die einzelnen Spezifikationen von P werden anschließend der Reihe nach übersetzt. Der prinzipielle Aufbau des Maschinenprogramms, das sich aus dieser Überlegung heraus ergibt, ist in Abbildung 5.16 graphisch dargestellt.

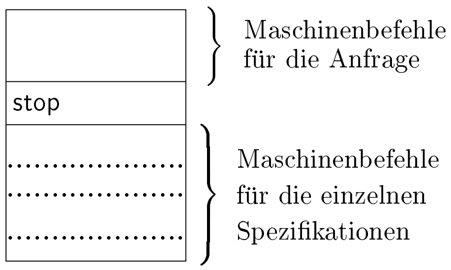


Abb. 5.16: Prinzipieller Aufbau eines übersetzten NoName*–Programms.

Kommen wir nun zur formalen Angabe der Übersetzung eines NoName*–Programms. Dabei sei im folgenden P ein beliebiges, aber festes NoName*–Programm (S, $\mathcal{F}$, *eval*, *tr_nr*, Δ, *Var*, *anfr*, $\mathcal{R}$), wobei $\mathcal{F}$ ein Funktionssystem (F_{if}, F_{pat}, F_s, F_h, F) ist.

Des weiteren seien $F_{if} = \{f_1, f_2, \ldots, f_{l_1}\}$, $F_{pat} = \{g_1, g_2, \ldots, g_{l_2}\}$, $F_{syn} = \{s_1, s_2, \ldots, s_{l_3}\}$ und $F_{inh} = \{h_1, h_2, \ldots, h_{l_4}\}$.

Außerdem sei $enum_S$ eine Aufzählungsfunktion für S, δ_to_n eine Konstruktorfunktion bezüglich S und Δ und *adr_fkt* eine Adressfunktion für F mit

$$\begin{array}{llll}
adr_fkt(f_1) = 1.1 & adr_fkt(g_1) = 2.1 & adr_fkt(s_1) = 3.1 & adr_fkt(h_1) = 4.1 \\
adr_fkt(f_2) = 1.2 & adr_fkt(g_2) = 2.2 & adr_fkt(s_2) = 3.2 & adr_fkt(h_2) = 4.2 \\
\vdots & \vdots & \vdots & \vdots \\
adr_fkt(f_{l_1}) = 1.l_1 & adr_fkt(g_{l_2}) = 2.l_2 & adr_fkt(s_{l_3}) = 3.l_3 & adr_fkt(h_{l_4}) = 4.l_4.
\end{array}$$

Die Übersetzung des NoName*–Programms P ist nun definiert als

$$\begin{array}{ll}
trans[P] = & \\
 & trans_{Anfr}[anfr](5) \\
 & \mathsf{stop} \\
1.1: & trans_{if}[f_1](1.1) \\
1.2: & trans_{if}[f_2](1.2) \\
 & \vdots \\
1.l_1: & trans_{if}[f_{l_1}](1.l_1) \\
2.1: & trans_{pat}[g_1](2.1) \\
2.2: & trans_{pat}[g_2](2.2) \\
 & \vdots \\
2.l_2: & trans_{pat}[g_{l_2}](2.l_2) \\
3.1: & trans_{syn}[s_1](3.1) \\
3.2: & trans_{syn}[s_2](3.2) \\
 & \vdots \\
3.l_3: & trans_{syn}[s_{l_3}](3.l_3) \\
4.1: & trans_{inh}[h_1](4.1) \\
4.2: & trans_{inh}[h_2](4.2) \\
 & \vdots \\
4.l_4: & trans_{inh}[h_{l_4}](4.l_4)
\end{array}$$

Wir wollen hier noch darauf hinweisen, daß die Reihenfolge, in der die einzelnen Funktionen übersetzt werden, im Grunde beliebig ist. Wir haben uns hier nur der Einfachheit halber für diese Reihenfolge entschieden.

5.4 Übersetzung von Funktionsspezifikationen via IF

Wir wollen nun angeben, wie eine Funktion $f \in F_{if}$, die via IF definiert ist, übersetzt wird. Betrachten wir dazu zuerst die Regel $rule \in \mathcal{R}$, die f definiert. Diese Regel sei

$$\begin{array}{rcl}
f(x_1, \ldots x_n) & = & rs_1, b_1 \\
 & = & rs_2, b_2 \\
 & & \vdots \\
 & = & rs_k, b_k
\end{array}$$

wobei $n \geq 0$ und $k \geq 1$ ist. Um herauszufinden, welche der gegebenen Alternativen $rs_1, \ldots, rs_k$ anzuwenden ist, müssen wir zuerst die Bedingungen $b_1, \ldots, b_k$ testen. Dazu teilen wir die Menge der Bedingungen $\{b_1, \ldots, b_k\}$ in die trivialen und die nicht–trivialen Bedingungen auf. Als triviale Bedingung wollen wir die Bedingung `OTHERWISE` bezeichnen. Wir wollen an dieser Stelle nochmals daran erinnern, daß eine Funktion, die via IF definiert ist, höchstens eine triviale Bedingung hat (vgl. Definition 3.16, Seite 35).

Um nun zu ermitteln, welche Alternative zutrifft, überprüfen wir zuerst die nicht–trivialen Bedingungen. In Abschnitt 3.1 haben wir bereits erläutert, daß die Reihenfolge, in der diese Bedingungen überprüft werden, nicht der angegebenen Reihenfolge entsprechen muß, sondern eine beliebige Permutation dieser Reihenfolge sein kann.

Dazu bilden wir das Tupel $((b'_1, rs'_1), \ldots, (b'_{k'}, rs'_{k'}))$, das eine beliebige Anordnung der Menge

$$\{(b_1, rs_1), \ldots, (b_k, rs_k)\} \backslash \{(\texttt{OTHERWISE}, rs) \mid rs \text{ beliebig}\}.$$

ist. Bei der Implementierung des Übersetzers sollte an dieser Stelle ein Zufallsgenerator Verwendung finden.

In der Variablen *other* merken wir uns nun noch, ob und wenn ja welche Alternative mit einer trivialen Bedingung versehen ist. Der Wert für *other* ergibt sich als

$$\{(b_1, rs_1), \ldots, (b_k, rs_k)\} \cap \{(\texttt{OTHERWISE}, rs) \mid rs \text{ beliebig}\}.$$

Das zu erzeugende Maschinenprogramm muß die einzelnen Bedingungen $b'_1, \ldots, b'_{k'}$ der Reihe nach überprüfen. Bei der ersten erfüllten Bedingung wird zu der Routine gesprungen, die den Wert der entsprechenden Alternative ermittelt. Sollte keine der Bedingungen $b'_1, \ldots, b'_{k'}$ zutreffen, so wird, falls vorhanden, die Alternative der trivialen Bedingung ausgeführt. In Abbildung 5.17 haben wir versucht, diese Vorgehensweise graphisch aufzubereiten.

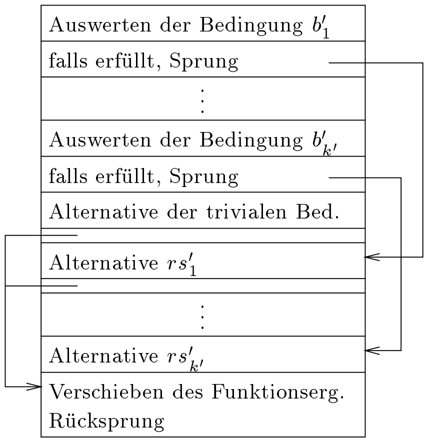


Abb. 5.17: Prinzipieller Aufbau des Maschinenprogramms einer Funktion f, die via IF definiert ist.

Es stellt sich jetzt noch die Frage, was zu tun ist, falls keine der nicht–trivialen Bedingungen erfüllt ist und die Funktion keine triviale Bedingung enthält. In diesem Fall wird die Ausführung des Maschinenprogramms mittels `stop` abgebrochen. Um aber kenntlich zu machen, daß die Ausführung nicht fehlerfrei durchgeführt werden konnte, setzen wir zuvor den Wert des Rückgabewert–Registers auf 1.

Die Maschinenbefehle zum Verschieben des Funktionsergebnisses und für den Rücksprung sind, wie wir bereits in Abschnitt 5.1 erläutert haben, cleanup bzw. ret.

Die formale Angabe der Übersetzungsfunktion für die Funktion f findet sich im folgenden angegeben.

$$
\begin{array}{ll}
\multicolumn{2}{l}{trans_{if}[f](adr) =} \\
 & trans_{RS}[b'_1](adr.1) \\
 & \textsf{jump\_on\_true}\ adr.1 \\
 & trans_{RS}[b'_2](adr.2) \\
 & \textsf{jump\_on\_true}\ adr.2 \\
 & \quad\vdots \\
 & trans_{RS}[b'_{k'}](adr.k') \\
 & \textsf{jump\_on\_true}\ adr.k' \\
 & trans_{other}[other](adr.(2k'+1), adr.0) \\
adr.1: & trans_{RS}[rs'_1](adr.(1+k')) \\
 & \textsf{jump}\ adr.0 \\
adr.2: & trans_{RS}[rs'_2](adr.(2+k')) \\
 & \textsf{jump}\ adr.0 \\
 & \quad\vdots \\
adr.(k'-1): & trans_{RS}[rs'_{k'-1}](adr.((k'-1)+k')) \\
 & \textsf{jump}\ adr.0 \\
adr.k': & trans_{RS}[rs'_{k'}](adr.(k'+k')) \\
adr.0: & \textsf{cleanup} \\
 & \textsf{ret}
\end{array}
$$

Die Übersetzungsfunktion $trans_{other}$ erzeugt die Maschinenbefehle des Feldes „Alternative der trivialen Bediengung“ aus Abbildung 5.17. Falls eine triviale Bedingung angegeben ist, d.h. $other = (\texttt{OTHERWISE}, rs)$, werden Maschinenbefehle für die Alternative rs erzeugt. Falls keine triviale Bedingung existiert, d.h. $other = \emptyset$, werden Befehle zum Setzen des Rückgabewert–Registers und zum Anhalten der Maschine erzeugt.

$$
\begin{array}{l}
trans_{other}[\emptyset](adr, retadr) = \\
\qquad \textsf{loadRC}\ 1 \\
\qquad \textsf{stop} \\
trans_{other}[\{(\texttt{OTHERWISE}, rs)\}](adr, retadr) = \\
\qquad trans_{RS}[rs](adr) \\
\qquad \textsf{jump}\ retadr
\end{array}
$$

Damit ist die Beschreibung der Übersetzungsschemata zur Bestimmung der anzuwendenden Alternative für eine Funktion, die via IF definiert ist, abgeschlossen. Im folgenden Abschnitt betrachten wir dieselbe Problematik für eine Funktion, die mittels Pattern Matching definiert ist.

5.5 Übersetzung von Funktionsspezifikationen via Pattern Matching

In diesem Abschnitt wollen wir uns mit den Übersetzungsfunktionen für eine Funktion $g \in F_{pat}$, also einer Funktion, die mittels Pattern Matching definiert ist, beschäftigen.

Auch hier stehen wir wieder vor der Aufgabe, Maschinenbefehle zu erzeugen, die es uns ermöglichen, die aktuell passende Alternative zu bestimmen. Die Regeln, die die Funktion g definieren, seien

$$\begin{aligned} g(p_{11}, \ldots, p_{1r}) &= rs_1 \\ g(p_{21}, \ldots, p_{2r}) &= rs_2 \\ &\vdots \\ g(p_{k1}, \ldots, p_{kr}) &= rs_k. \end{aligned}$$

Da jedes Pattern Matching überschneidungsfrei und erschöpfend sein muß, wissen wir, daß immer genau eine Alternative angewendet werden kann.

Eine naive Vorgehensweise zur Bestimmung der passenden Alternative wäre folglich, jede Gleichung der Reihe nach zu betrachten und zu überprüfen, ob die aktuellen Parameterwerte zu den angegebenen Mustern passen. Diese Vorgehensweise ist aber ineffizient und führt, wie das folgende Beispiel zeigt, im Falle der Auswertungsstrategie Call–by–Name zu einem unerwünschten Effekt.

Beispiel 5.10

Betrachten wir die in Abbildung 5.18 angegebenen Funktionsgleichungen einer Funktion *test*, die mittels Pattern Matching definiert ist. Die Auswertungsstrategie für `test` sei Call–by–Name.

```
test(x1,0) = 1
test(0,n+1) = 2
test(m+1,n+1) = 3
```

Abb. 5.18: Gleichungen der Funktion `test`.

Rufen wir diese Funktion mit der Anfrage `test(...,0)` auf, so ist klar, daß das Funktionsergebnis 1 sein muß. Auf ein Auswerten des ersten Parameters kann in diesem Fall verzichtet werden.

Nehmen wir nun an, daß die Anfrage `test(1 div 0,0)` lautet, so führt das Auswerten des ersten Parameters sogar zum Abbruch der Berechnung, obwohl der Wert des ersten Parameters zur Bestimmung des Funktionsergebnisses gar nicht benötigt wird. Im Falle der Auswertungsstrategie Call–by–Name ist dies freilich unerwünscht. □

Eine wesentlich elegantere Lösung ist die Verwendung von CASE–Ausdrücken. Ein CASE–Ausdruck repräsentiert die Fallunterscheidungen, die bei der Bestimmung der passenden Alternative anzuwenden sind. CASE–Ausdrücke sind induktiv aufgebaut: Entweder handelt es sich um einen Ausduck der Form CASE pos OF $p_1 \to case_1 \ldots p_k \to case_k$, wobei die $case_i$ wiederum CASE–Ausdrücke sind, oder um einen Ausdruck CASENIL rs, wobei rs

die rechte Seite einer Funktionsgleichung ist. In einem CASE–Ausdruck darf jede Parameterposition *pos* höchstens einmal vorkommen und die einzelnen Muster $p_1, \ldots, p_k$ müssen typmäßig zu der *pos*–ten Parameterposition passen und alle Möglichkeiten abdecken.

Beispiel 5.11 (Fortsetzung von Beispiel 5.10)

Der zur Funktion `test` gehörende CASE–Ausdruck lautet

$$
\begin{array}{l}
\text{CASE 2 OF} \\
\quad 0 \rightarrow \text{CASENIL 1} \\
\quad \geq 1 \rightarrow \text{CASE 1 OF} \\
\qquad\qquad\qquad 0 \rightarrow \text{CASENIL 2} \\
\qquad\qquad\qquad \geq 1 \rightarrow \text{CASENIL 3.}
\end{array}
$$

□

Hier stellt sich nun die Frage, ob ein solcher CASE–Ausdruck immer existiert, oder ob es Funktionsdefinitionen gibt, für die sich kein CASE–Ausdruck finden läßt. Der folgende Satz beantwortet diese Frage.

Satz 5.12 (Existenz eines CASE–Ausdrucks)

Ist das durch die Funktionsgleichungen von g induzierte Pattern Matching erschöpfend und überschneidungsfrei, so existert ein zu g gehörender CASE–Ausdruck. □

Beweis

Beobachten wir zuerst, daß ein CASE–Ausdruck eine vollständige Partitionierung aller möglichen Eingabevektoren von g vornimmt.

Nehmen wir nun an, daß es eine Funktion g gibt, die mittels eines erschöpfenden und überschneidungsfreien Pattern Matchings definiert ist und für die sich kein CASE–Ausdruck finden läßt.

Daraus folgt, daß sich keine vollständige Partitionierung aller möglichen Eingabevektoren von g finden läßt. Dies wiederum bedeutet, daß es wenigstens einen Funktionsaufruf $g(x_1, \ldots, x_n)$ gibt, der sich nicht eindeutig einer Funktionsgleichung zuordnen läßt.

Dies bedeutet aber, daß das von den Funktionsgleichungen von g induzierte Pattern Matching nicht überschneidungsfrei und erschöpfend gewesen sein kann.
⇒ Widerspruch. □

Ausgehend von einem CASE–Ausdruck ist es einfach, Maschinenbefehle für die Bestimmung der passenden Alternative zu erzeugen. Im folgenden Abschnitt befassen wir uns nun damit, einen CASE–Ausdruck automatisch aus den Funktionsgleichungen von g zu ermitteln. Das hier vorgestellte Verfahren basiert auf [Pey87]. In den Abschnitten 5.5.2 und 5.5.3 zeigen wir dann, wie mit Hilfe eines CASE–Ausdrucks Maschinenbefehle erzeugt werden.

5.5.1 Ermittlung der Auswertungsreihenfolge der Parameter

Im folgenden beschreiben wir die Erzeugung eines CASE–Ausdrucks aus einer Funktionsdefinition von g. Dazu wollen wir zuerst den Begriff CASE–Ausdruck formal einführen.

Definition 5.13 (CASE–Ausdruck)
Die *Menge der CASE–Ausdrücke*, bezeichnet mit $CASEEXPR(S, F, \Delta, Var)$ ist die kleinste Menge E, für die gilt

- für alle $e \in F\text{–}EXPR\,(S, F, \Delta, Var)$ liegt CASENIL e in E
- CASE $parPos$ OF

  $$\begin{array}{l} pattern_1 \rightarrow caseExpr_1 \\ pattern_2 \rightarrow caseExpr_2 \\ \quad\vdots \\ pattern_n \rightarrow caseExpr_n \end{array}$$

  ist ein Element von E, wobei $parPos \in \mathbb{N}$, $n \geq 1$ und für alle $i \in [n]$ gilt $pattern_i \in PAT\,(S, \Delta, Var)$ und $caseExpr_i \in E$. □

Bei der Generierung eines CASE–Ausdrucks zu g gehen wir folgerdermaßen vor:

- Wir bilden aus den Funktionsgleichungen für g einen Match–Ausdruck. Ein Match–Ausdruck beschreibt die Funktionsgleichungen von g, soweit sie zur Bestimmung der anzuwendenden Alternative notwendig sind.
- Diesen Match–Ausdruck formen wir mit Hilfe von vier Transformationsregeln schrittweise in einen CASE–Ausdruck um.

Die formale Definition eines Match–Ausdrucks haben wir im folgenden angegeben.

Definition 5.14 (Match–Ausdruck)
Ein *Match–Ausdruck* ist ein Term der Gestalt $match\ A$, wobei A ein Element der Menge

$$\mathbb{N}^* \times \mathbb{N}_0 \times [(PAT\,(S, \Delta, Var))^* \times F\text{–}EXPR\,(S, F, \Delta, Var)]^*$$

ist. □

Die Zuordnung von Funktionsgleichungen für g zu Match–Ausdruck ist intuitiv. Sind

$$\begin{array}{rcl} g(p_{11}, \ldots, p_{1r}) & = & rs_1 \\ g(p_{21}, \ldots, p_{2r}) & = & rs_2 \\ & \vdots & \\ g(p_{k1}, \ldots, p_{kr}) & = & rs_k \end{array}$$

alle Regeln in $\mathcal{R}$, die g definieren, so lautet der zugehörige Match–Ausdruck

$$\begin{array}{l} match\,((1, 2, \ldots, r),\ r, \\ \quad [((p_{11}, p_{12}, \ldots, p_{1r}), rs_1), \\ \quad\ ((p_{21}, p_{22}, \ldots, p_{2r}), rs_2), \\ \qquad \vdots \\ \quad\ ((p_{k1}, p_{k2}, \ldots, p_{kr}), rs_k)]). \end{array}$$

Der erste Parameter eines Match–Ausdrucks ist eine Liste von Zahlen, die Eingabeparameterpositionen bezeichnen. Der zweite Parameter gibt die Anzahl der bereits bekannten Funktionsparameter an. Diese Zahl herhöht sich, wenn durch ein Muster weitere Variablen eingeführt werden. Der dritte Parameter schließlich beinhaltet die linken und rechten Seiten der Funktionsgleichungen für g.

Beispiel 5.15 (Korrespondierender Match–Ausdruck)

Der zu den bereits bekannten Funktionsgleichungen der Funktion `test` aus Abbildung 5.18 gehörende Match–Ausdruck lautet

$$\begin{aligned}&\mathit{match}\,((1,2),\ 2,\\&\quad[((x_1,0),1),\\&\quad\ ((0,n+1),2),\\&\quad\ ((m+1,n+1),3)]).\end{aligned}\qquad\square$$

Im folgenden führen wir die verschiedenen Transformationsregeln ein. Wir wollen aber schon an dieser Stelle auf das Beispiel 5.19 verweisen, in dem wir den gesamten Transformationsvorgang für eine gegebene Funktion demonstrieren.

Die Auswahl der anzuwendenden Transformationsregel erfolgt aufgrund der jeweils ersten Muster im dritten Argument des Match–Ausdrucks, also aufgrund von p_{11}, p_{21}, $\ldots$, p_{k1}. Diese Muster nennen wir *Auswahlmuster*.

Variablen–Regel

Sind alle Auswahlmuster Variablen, so bedeutet dies, daß die Bestimmung der passenden Alternative nicht vom Wert dieses Parameters abhängt. Das Problem zur Bestimmung der passenden Alternative reduziert sich folglich auf die Betrachtung der weiteren Parameter.

Der Match–Ausdruck

$$\begin{aligned}&\mathit{match}\,((i_1,i_2,\ldots,i_r),\ \mathit{next},\\&\quad[((p_{11},p_{12},\ldots,p_{1r}),rs_1),\\&\quad\ ((p_{21},p_{22},\ldots,p_{2r}),rs_2),\\&\qquad\quad\vdots\\&\quad\ ((p_{k1},p_{k2},\ldots,p_{kr}),rs_k)])\end{aligned}$$

wobei für alle $i \in [k]$ gilt, daß $p_{i1} \in \mathit{Var}$, kann also zu dem Ausdruck

$$\begin{aligned}&\mathit{match}\,((i_2,\ldots,i_r),\ \mathit{next},\\&\quad[((p_{12},\ldots,p_{1r}),rs_1),\\&\quad\ ((p_{22},\ldots,p_{2r}),rs_2),\\&\qquad\quad\vdots\\&\quad\ ((p_{k2},\ldots,p_{kr}),rs_k)])\end{aligned}$$

umgeformt werden.

Konstruktor–Regel

Die nächste zu betrachtende Möglichkeit ist, daß alle Auswahlmuster nur Konstanten und gemischte Ausdrücke sind. In diesem Fall können wir aufgrund des Parameterwertes eine echte Einschränkung bezüglich der noch zur Auswahl stehenden Alternativen treffen. Ein Auswerten des Parameters ist hier also erforderlich. Wir erhalten folglich für den zu bildenden CASE–Ausdruck einen Knoten der Form CASE ... OF ... .

Beispiel 5.16 (Konstruktor–Regel)

Betrachten wir die in Abbildung 5.19 angegebenen Funktionsgleichungen der Funktion `test2`. Den zugehörigen Match–Ausdruck haben wir in Abbildung 5.20 wiedergegeben.

```
test2(0,x2) = 1
test2(n+1,0) = 2
test2(n+1,m+1) = 3
```

Abb. 5.19: Funktion `test2`.

$$\begin{aligned}&match\,((1,2),\ 2,\\&\quad [((0,x_2),1),\\&\quad\ ((n+1,0),2),\\&\quad\ ((n+1,m+1),3)]).\end{aligned}$$

Abb. 5.20: Match–Ausdruck für `test2`.

Wir können hier in Abhängigkeit eines Wertes für den ersten Parameter eine Vorentscheidung über die passende Alternative treffen.

Als CASE–Ausdruck erhalten wir also den in Abbildung 5.21 dargestellten Term.

$$\begin{aligned}&\text{CASE 1 OF}\\&\quad 0 \rightarrow match\,((2),\ 2,\ [((x_2),1)])\\&\quad 1 \rightarrow match\,((2),\ 3,\ [((0),2),((m+1),3)])\end{aligned}$$

Abb. 5.21: Resultierender CASE–Ausdruck. □

Um den Transformationsschritt eindeutig zu machen, müssen wir festlegen, in welcher Reihenfolge die Verzweigungspunkte angeordnet werden sollen. Dazu müssen wir eine Ordnung auf den möglichen Mustern definieren.

Definition 5.17 (Ordnungsrelation auf Mustern ($<_{pat}$, $=_{pat}$))

Seien p_1 und p_2 zwei Muster aus $PAT(S, \Delta, Var)$. Das Muster p_1 heißt *strukturell gleich* wie das Muster p_2, kurz $p_1 =_{pat} p_2$, wenn eine der folgenden Bedingungen erfüllt ist.

- $p_1 = p_2$,
- $p_1, p_2 \in Var$,
- $p_1 = v_1 + n$ und $p_2 = v_2 + n$ mit $v_1, v_2 \in Var$ und $n \in I\!N$, oder
- $p_1 = d(x_1, \ldots, x_n)$ und $p_2 = d(x'_1, \ldots, x'_n)$ mit $d \in \Delta$ und $x_i, x'_i \in Var$ für $1 \leq i \leq n$

Das Muster p_1 heißt *strukturell kleiner* als des Muster p_2, kurz $p_1 <_{pat} p_2$, wenn eine der folgenden Bedingungen erfüllt ist.

- $p_1 = \texttt{FALSE}$ und $p_2 = \texttt{TRUE}$,
- $p_1, p_2 \in \mathbb{N}_0$ und $p_1 < p_2$,
- $p_1 \in \mathbb{N}_0$, $p_2 = v + n$ mit $v \in \mathit{Var}$, $n \in \mathbb{N}$ und $p_1 < n$ oder
- $p_1 = d_1(x_1, \ldots, x_n)$ und $p_2 = d_2(x'_1, \ldots, x'_k)$ mit $n, k \in \mathbb{N}_0$ wobei $x_i, x'_j \in \mathit{Var}$ und entweder
  - ◇ $d_1 \in \Delta[\theta_1]_S$, $d_2 \in \Delta[\theta_2]_S$ und $\mathit{enum}_S(\theta_1) < \mathit{enum}_S(\theta_2)$ mit $\theta_1, \theta_2 \in S$, oder
  - ◇ $d_1 \in \Delta[\theta]_S$, $d_2 \in \Delta[\theta]_S$ und $\delta\_to\_n(d_1) < \delta\_to\_n(d_2)$ mit $\theta \in S$

Durch $=_{pat}$ und $<_{pat}$ wird eine Ordnungsrelation auf $\mathit{PAT}(S, \Delta, \mathit{Var})$ definiert. □

Die obige Überlegung führen uns zu folgender Transformationsregel: Seien im Ausdruck

$$\begin{array}{l} \mathit{match}\,((i_1, i_2, \ldots, i_r),\ \mathit{next}, \\ \quad [((p_{11}, p_{12}, \ldots, p_{1r}), rs_1), \\ \quad\ ((p_{21}, p_{22}, \ldots, p_{2r}), rs_2), \\ \qquad \vdots \\ \quad\ ((p_{k1}, p_{k2}, \ldots, p_{kr}), rs_k)]) \end{array}$$

alle p_{i1} für $i \in [k]$ Elemente der Menge $\mathit{PAT}(S, \Delta, \mathit{Var}) \setminus \mathit{Var}$.

In diesem Fall bestimme die Funktion $\mathit{part} : [k] \to [\tilde{k}]$, so daß für alle $1 \leq i, j \leq k$ gilt, daß

$$\mathit{part}(i) = \mathit{part}(j) \Leftrightarrow p_{i1} =_{pat} p_{j1},$$

$$\mathit{part}(i) < \mathit{part}(j) \Leftrightarrow p_{i1} <_{pat} p_{j1}$$

und $\tilde{k}$ minimal ist. Diese Funktion ist eindeutig.

Dann läßt sich obiger Match–Ausdruck zu dem folgenden Match–Ausdruck umformen.

$$\begin{array}{l} \text{CASE } i_1 \text{ OF} \\ \quad 1 \to \mathit{match}\,((i_2, \ldots, i_r),\ (\mathit{next} + \mathit{bind}(1)),\ [\mathit{patExpr}(1)]) \\ \quad 2 \to \mathit{match}\,((i_2, \ldots, i_r),\ (\mathit{next} + \mathit{bind}(2)),\ [\mathit{patExpr}(2)]) \\ \qquad\qquad \vdots \\ \quad \tilde{k} \to \mathit{match}\,((i_2, \ldots, i_r),\ (\mathit{next} + \mathit{bind}(\tilde{k})),\ [\mathit{patExpr}(\tilde{k})]) \end{array}$$

Mit $\mathit{patExpr}(n)$ wird dabei die Sequenz der Tupel

$$(\ ((p_{i2}, \ldots, p_{ir}), rs'_i) \mid i \in [k] \text{ und } \mathit{part}(i) = n)$$

bezeichnet. Die in p_{i1} vorkommenden Variablen seien der Reihe nach $v_1, \ldots, v_m$. Die rechte Seite einer Funktionsgleichung rs'_i entsteht dann aus rs_i, indem die Substitution $[v_1/x_{next+1}, \ldots, v_m/x_{next+m}]$ auf rs_i angewandt wird. Für die Funktion bind gilt, daß $\mathit{bind}(n) = m$, wobei $\mathit{part}(i) = n$ und in p_{i1} kommen die Variablen $v_1, \ldots, v_m$ vor.

An dieser Stelle wollen wir darauf hinweisen, daß jeder Match–Ausdruck, der durch Anwenden dieser Regel erzeugt wird, wiederum für ein überschneidungsfreies und erschöpfendes Pattern Matching steht. Der Beweis dieser Aussage sei dem interessieren Leser überlassen.

Leere Regel

Wir benötigen noch eine Regel, die angibt, wann der Transformationsprozeß terminiert. Keine weiteren Fallunterscheidungen sind zu treffen, wenn es keine Auswahlmuster mehr gibt. Formalisiert bedeutet dies, daß der Match–Ausdruck

$$\begin{array}{l} \mathit{match}\,((),\ \mathit{next}, \\ \quad [(\,(), rs)]) \end{array}$$

zu dem CASE–Ausdruck CASENIL rs vereinfacht werden kann.

Rotations–Regel

Falls keiner der vorherigen Fälle trifft, so repräsentiert der Match–Ausdruck ein Pattern Matching, bei dem das Auswahlmuster in manchen Gleichungen eine Variable ist und in manchen Gleichungen nicht. Es ist also nicht sicher, ob der Parameterwert in jedem Fall zur Entscheidungsfindung bezüglich der anzuwendenden Alternative überhaupt benötigt wird. Die Bearbeitung dieser Parameterposition wird hintenangestellt. Aus dem Ausdruck

$$\begin{array}{l} \mathit{match}\,((i_1, i_2, \ldots, i_r),\ \mathit{next}, \\ \quad [((p_{11}, p_{12}, \ldots, p_{1r}), rs_1), \\ \quad\ ((p_{21}, p_{22}, \ldots, p_{2r}), rs_2), \\ \qquad\quad \vdots \\ \quad\ ((p_{k1}, p_{k2}, \ldots, p_{kr}), rs_k)]) \end{array}$$

wird der Ausdruck

$$\begin{array}{l} \mathit{match}\,((i_2, \ldots, i_r, i_1),\ \mathit{next}, \\ \quad [\ ((p_{12}, \ldots, p_{1r}, p_{11}), rs_1), \\ \quad\ \ ((p_{22}, \ldots, p_{2r}, p_{21}), rs_2), \\ \qquad\quad \vdots \\ \quad\ \ ((p_{k2}, \ldots, p_{kr}, p_{k1}), rs_k)]). \end{array}$$

Zur Umwandlung eines Match–Ausdrucks M in einen CASE–Ausdruck wird der gegebene Ausdruck M solange anhand der vorgestellten Regeln umgeformt, bis im resultierenden Ausdruck keine Match–Ausdrücke mehr enthalten sind. Der so entstandene Ausdruck ist dann ein CASE–Ausdruck.

Die eben vorgestellte Rotations–Regel läßt natürlich die Frage aufkommen, ob es nicht passieren kann, daß diese Regel beliebig oft angewandt wird und der Transformationsprozeß damit nicht terminiert.

Satz 5.18 (Terminierung des Umwandlungsvorgangs von Match–Ausdruck nach CASE–Ausdruck)

Ist das Pattern Matching, das von den Funktionsgleichungen einer Funktion g induziert wird, erschöpfend und überschneidungsfrei, so terminiert der Transformationsvorgang von Match–Ausdruck nach CASE–Ausdruck. □

Beweis

Nehmen wir an, daß der Transformationsvorgang für g nicht terminiert. Dies bedeutet, daß wir im Verlauf der Transformation auf einen Match–Ausdruck

$$\begin{array}{l} match\,((i_1, i_2, \ldots, i_r),\ next, \\ \quad [((p_{11}, p_{12}, \ldots, p_{1r}), rs_1), \\ \quad\ ((p_{21}, p_{22}, \ldots, p_{2r}), rs_2), \\ \qquad \vdots \\ \quad\ ((p_{k1}, p_{k2}, \ldots, p_{kr}), rs_k)]) \end{array}$$

gestoßen sind, für den folgende Aussagen gelten: $r \geq 1$, $k \geq 2$ und für alle $j \in [r]$ gilt: In $p_{1j}, \ldots, p_{kj}$ findet sich mindestens ein Element aus Var und mindestens ein Element aus $PAT(S, \Delta, Var) \setminus Var$.

Folglich gibt es einen Eingabevektor von g, für den sich nicht eindeutig eine Zeile bestimmen läßt, so daß alle Elemente des Eingabevektors zu den angegebenen Mustern passen. Das diesem Match–Ausdruck entsprechende Pattern Matching kann also nicht überschneidungsfrei und erschöpfend sein.

Da aber jeder Match–Ausdruck, der sich im Verlauf der Anwendung der Transformationsregeln ergibt, wieder einem überschneidungsfreien und erschöpfenden Pattern Matching entsprechen muß, folgt daraus, daß das zugrundeliegende Pattern Matching nicht überschneidungsfrei und erschöpfend gewesen sein kann.
$\Rightarrow$ Widerspruch □

Betrachten wir zur Verdeutlichung des eben vorgestellten Verfahrens noch ein Beispiel, in dem alle eingeführten Regeln zur Anwendung kommen.

Beispiel 5.19 (Ermittlung des korrespondierenden CASE–Ausdrucks)

In Abbildung 5.22 finden sich die Funktionsgleichungen der Funktion `bsp`, die mittels Pattern Matchung definiert ist. Die darin verwendeten Konstruktoren gehören zum algebraischen Datentyp `Baum`, der bereits aus Abbildung 3.1 bekannt ist. Dieses Pattern Matching ist überschneidungsfrei und erschöpfend. Der zu diesen Funktionsgleichungen gehörende Match–Ausdruck findet sich in Abbildung 5.23.

```
bsp(Knoten(a,b,c),TRUE,n+1)  = (1,n)
bsp(Blatt(d),x2,0)           = (2,d)
bsp(Blatt(d),x2,n+1)         = (3,x2)
bsp(Knoten(a,b,c),FALSE,n+1) = (4,a)
bsp(Knoten(a,b,c),x2,0)      = (5,x2)
```

Abb. 5.22: Definitionsgleichungen der Funktion `bsp`.

In den Funktionsgleichungen steckt ein kleiner Fehler: Die Typen der verschiedenen Ergebnisse in der zweiten Ergebniskomponente passen nicht zueinander. Hätten wir typkorrekte Ausdrücke verwendet, wären die rechten Seiten wesentlich länger geworden, was der Lesbarkeit abträglich gewesen wäre. Für die Konstruktor–Funktion δ_to_n, die wir im weiteren Verlauf des Beispiels benötigen, gelte, daß δ_to_n(`Knoten`) $= 1$ und δ_to_n(`Blatt`) $= 2$.

$$
\begin{array}{l}
match\,((1,2,3),\ 3,\\
\quad [((\texttt{Knoten}(a,b,c), \texttt{TRUE}, & n+1), (1,n)),\\
\quad\ ((\texttt{Blatt}(d), x_2, & 0), (2,d)),\\
\quad\ ((\texttt{Blatt}(d), x_2, & n+1), (3,x_2)),\\
\quad\ ((\texttt{Knoten}(a,b,c), \texttt{FALSE}, & n+1), (4,a)),\\
\quad\ ((\texttt{Knoten}(a,b,c), x_2, & 0), (5,x_2))])
\end{array}
$$

Abb. 5.23: Zu den Funktionsgleichungen von `bsp` gehörender Match–Ausdruck.

Da alle Auswahlmuster gemische Ausdrücke sind, findet die Konstruktor–Regel Anwendung. Dabei benötigen wir die Funktion *part*. Mit $\texttt{Knoten}(a,b,c) <_p \texttt{Blatt}(d)$ erhalten wir $part(1) = part(4) = part(5) = 1$ und $part(2) = part(3) = 2$. Die Anwendung der Konstruktor–Regel ergibt folglich den Ausdruck, der in Abbildung 5.24 wiedergegeben ist. Um die anzuwendende Alternative zu bestimmen, muß also auf jeden Fall der erste Parameter ausgewertet werden.

$$
\begin{array}{lll}
\text{CASE 1 OF} & & \\
\quad \texttt{Knoten}(a,b,c) & \rightarrow & match\,((2,3),\ 6,\\
 & & \quad [((\texttt{TRUE},\ n+1),\ (1,n)),\\
 & & \quad\ ((\texttt{FALSE}, n+1),\ (4,x_4)),\\
 & & \quad\ ((x_2,\quad 0),\quad (5,x_2))])\\
\quad \texttt{Blatt}(d) & \rightarrow & match\,((2,3),\ 4,\\
 & & \quad [((x_2, 0),\quad (2,x_4)),\\
 & & \quad\ ((x_2, n+1),\ (3,x_2))])
\end{array}
$$

Abb. 5.24: Ausdruck nach Anwendung der Konstruktor–Regel.

Die Zahlen 6 bzw. 4 in den erzeugten Match–Ausdrücken stammen von den Variablen, die in den bei der Fallunterscheidung betrachteten Mustern $\texttt{Knoten}(a,b,c)$ bzw. $\texttt{Blatt}(d)$ neu eingeführt wurden. Im Zuge der Neueinführung wurden diese Variablen auch auf den rechten Seiten beachtet. Aus der rechten Seite $(2,d)$ ist durch Umbenennen der neu bekannt gewordenen Variablen die rechte Seite $(2,x_4)$ entstanden.

Betrachten wir uns nun die beiden Match–Ausdrücke, so erkennen wir, daß auf den oberen Match–Ausdruck nur die Rotations–Regeln angewandt werden kann. Auf den unteren Match–Ausdruck hingegen müssen wir die Variablen–Regel anwenden. Wir erhalten durch das simultane Anwenden beider Regeln den in Abbildung 5.25 angegebenen Ausdruck.

$$
\begin{array}{lll}
\text{CASE 1 OF} & & \\
\quad \texttt{Knoten}(a,b,c) & \rightarrow & match\,((3,2),\ 6,\\
 & & \quad [((n+1,\ \texttt{TRUE}),\ (1,n)),\\
 & & \quad\ ((n+1,\ \texttt{FALSE}), (4,x_4)),\\
 & & \quad\ ((0,\quad x_2),\quad (5,x_2))])\\
\quad \texttt{Blatt}(d) & \rightarrow & match\,((3),\ 4,\\
 & & \quad [((0),\quad (2,x_4)),\\
 & & \quad\ ((n+1),\ (3,x_2))])
\end{array}
$$

Abb. 5.25: Ausdruck nach simultaner Anwendung der Rotations– und Variablen–Regel.

Nun können wir auf beide Match–Ausdrücke die Konstruktor–Regel anwenden. Der nun entstandene Ausdruck ist in Abbildung 5.26 dargestellt.

$$
\begin{array}{llll}
\text{CASE 1 OF} & & & \\
\quad \texttt{Knoten}(a,b,c) & \to \text{CASE 3 OF} & & \\
 & \quad = 0 & \to & \mathit{match}\,((2),\ 6, [((x_2),(5,x_2))]) \\
 & \quad \geq 1 & \to & \mathit{match}\,((2),\ 7, \\
 & & & \quad [((\texttt{TRUE}),\ \ (1,x_7)), \\
 & & & \quad ((\texttt{FALSE}),(4,x_4))]), \\
\quad \texttt{Blatt}(d) & \to \text{CASE 3 OF} & & \\
 & \quad = 0 & \to & \mathit{match}\,((),\ 4,\ [((),(2,x_4))]) \\
 & \quad \geq 1 & \to & \mathit{match}\,((),\ 5,\ [((),(3,x_2))])
\end{array}
$$

Abb. 5.26: Match–Ausdruck nach Anwendung der Konstruktor–Regel.

Die oberen beiden Match–Ausdrücke transformieren wir nun mit der Variablen–Regel bzw. Konstruktor–Regel. Auf die beiden unteren Match–Ausdrücke kann nur die leere Regel angewandt werden. Der nach Ausführung dieser Regeln entstandene Ausdruck enthält drei leere Match–Ausdrücke. Werden auch diese noch ersetzt, so erhalten wir den CASE–Ausdruck, der in Abbildung 5.27 wiedergegeben ist.

$$
\begin{array}{llll}
\text{CASE 1 OF} & & & \\
\quad \texttt{Knoten}(a,b,c) & \to \text{CASE 3 OF} & & \\
 & \quad = 0 & \to & \text{CASENIL } (5,x_2) \\
 & \quad \geq 1 & \to & \text{CASE 2 OF} \\
 & & & \quad \texttt{TRUE} \ \ \to \text{CASENIL } (1,x_7) \\
 & & & \quad \texttt{FALSE} \to \text{CASENIL } (4,x_4) \\
\quad \texttt{Blatt}(d) & \to \text{CASE 3 OF} & & \\
 & \quad = 0 & \to & \text{CASENIL } (2,x_4) \\
 & \quad \geq 1 & \to & \text{CASENIL } (3,x_2)
\end{array}
$$

Abb. 5.27: Resultierender CASE–Ausdruck.

Die Transformation des ursprünglichen Match–Ausdrucks in eine CASE–Ausdruck ist damit abgeschlossen. Aus dem resultierenden CASE–Ausdruck können wir nun die nacheinander anzuwendenden Fallunterscheidungen genau ablesen. □

5.5.2 Codeerzeugung für Funktionen mit Pattern Matching

Im vorigen Abschnitt haben wir aus den Funktionsgleichungen einer Funktion $g \in F_{pat}$ einen CASE–Ausdruck konstruiert. Diesen CASE–Ausdruck werden wir nun verwenden, um die Funktion g zu übersetzen. Dabei bilden wir einfach die durch den CASE–Ausdruck vorgegebene Struktur durch Maschinenbefehle nach. Einen CASE–Ausdruck der Form

$$\text{CASE } \mathit{pos} \text{ OF Fall}_1 \to \mathit{case}_1 \text{ Fall}_2 \to \mathit{case}_2 \ldots \text{Fall}_k \to \mathit{case}_k$$

übersetzen wir in der in Abbildung 5.28 dargestellten Weise. Stoßen wir im Laufe der Übersetzung auf einen CASE–Ausdruck der Gestalt CASENIL rs, so ist rs die rechte Seite einer Funktionsgleichung und wird entsprechend übersetzt. Anschließend folgt ein Sprungbefehl an das Ende der Routine für die Funktion g, an der die Ergebnisübergabe und der Rücksprung durchgeführt wird.

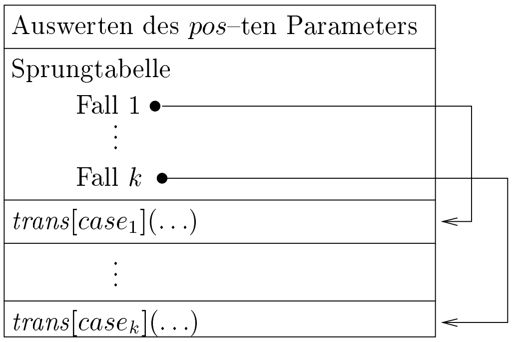


Abb. 5.28: Prinzipielle Übersetzung von CASE *pos* OF … .

Sei $g \in F_{pat}$ eine Funktion, die via Pattern Matching definiert ist. Den Funktionsgleichungen von g entspricht, wie wir in Abschnitt 5.5.1 gesehen haben, ein Match–Ausdruck. Diesen Match–Ausdruck kann man mit dem eben vorgestellten Verfahren in einen CASE–Ausdruck $case(g)$ umwandeln.

Die Übersetzung von g ist dann die Übersetzung des CASE–Ausdrucks $case(g)$.

$trans_{pat}[g](adr) =$
    $trans_{case}[case(g)](g, adr.1, adr.0)$
$adr.0$: cleanup
    ret

Die Übersetzung des CASE–Ausdrucks erfolgt über der Struktur dieses Ausdrucks.

$trans_{case}[\text{CASE } pos \text{ OF } 1 \rightarrow case_1 \ 2 \rightarrow case_2 \ \ldots k \rightarrow case_k](g, adr, retadr) =$
    $trans_{pateval}(pos, g, adr.1, k)$
    jump_idx adr.2.0
$adr.2.0$: jump adr.2.1
    jump adr.2.2
    ⋮
    jump adr.2.k
$adr.2.1$: $trans_{case}[case_1](g, adr.2.1, retadr)$
$adr.2.2$: $trans_{case}[case_2](g, adr.2.2, retadr)$
    ⋮
$adr.2.k$: $trans_{case}[case_k](g, adr.2.k, retadr)$

$trans_{case}[rs](g, adr, retadr) =$
    $trans_{RS}(rs, adr)$
    jump $retadr$

Kommen wir nun zu den Maschinenbefehlen, die erzeugt werden müssen, um aus dem Parameterwert die für den indizierten Sprung benötigte Zahl auf den Berechnungsstack zu erzeugen.

5.5.3 Codeerzeugung für die Auswertung eines Musters

In diesem Abschnitt wollen wir die Übersetzungsfunktion $trans_{pateval}$ behandeln, die für eine gegebene Parameterpostion sowohl die neu eingeführten Variablen bestimmen soll, als auch einen numerischen Wert liefern. Dieser numerische Wert wird, wie wir im vorigen Abschnitt gesehen haben, für einen nachfolgenden indizierten Sprung benötigt.

Betrachten wir zuerst das Problem der neu eingeführten Variablen. Durch den Wert $next$, der in den Match–Ausdrücken aus Abschnitt 5.5.1 vorkam, und die Variablenumbenennung in der Konstruktor–Regel, wurden für die neu eingeführten Variablen ebenfalls Namen der Form x_i vergeben. Der Index i gibt dabei an, an welche Parameterposition des Kellerrahmens der Wert dieser Variablen gehört.

Zum Zeitpunkt der Ausführung der von $trans_{pateval}$ erzeugten Befehle liegen nur der Kellerrahmen sowie gegebenenfalls weitere, bereits eingeführte Variablenwerte auf dem Berechnungsstack. Die durch das aktuell betrachtete Muster neu eingeführten Variablenwerte müssen also oben auf dem Berechnungsstack abgelegt werden.

Die für den indizierten Sprung nötigen Zahl erhalten wir durch eine numerische Kodierung des ursprünglichen Parameterwertes. Diese Kodierung ist natürlich abhängig vom Typ des aktuell betrachteten Parameters.

Sei $g \in F_{pat}^{(s_1 \ldots s_n, \sigma_1 \ldots \sigma_m)}$ mit $n, m \geq 1$ und $s_i, \sigma_j \in S$ sowie $pos \in \mathbb{N}$. Falls $s_{pos} = bool$, also an der aktuell betrachteten Parameterposition ein Wahrheitswert steht, dann gilt

$trans_{pateval}[pos](g, adr, max) =$

```
          eval pos
          jump_on_true adr.1
          push_const (Y_num : 1)
          jump adr.2
adr.1 :   push_const (Y_num : 2)
adr.2 :
```

Für `FALSE` wird die Zahl 1 aufgelegt, für `TRUE` die Zahl 2. Neu eingeführte Variablen kann es in diesem Fall nicht geben.

Falls $s_{pos} = card$ ist, der aktuell betrachtete Parameter also eine Zahl ist, dann gilt

$trans_{pateval}[pos](g, adr, max) =$

```
          eval pos
          dup
          push_const (Y_num : max)
          les
          jump_on_true adr.1
          push_const (Y_num : max)
          sub
          push_const (Y_num : max)
adr.1 :   push_const (Y_num : 1)
          add
```

Ist der ermittelte Parameterwert x kleiner als max, so trifft ein Fall mit einer Konstante als Muster zu. In diesem Fall werden keine neuen Variablen eingeführt und nur der Wert x

als Sprungindex auf dem Berechnungsstack zurückgelassen. Gilt aber $x \geq max$, so muß ein neuer Funktionsparameter mit dem Wert $x - max$ angelegt werden. Als Sprungindex muß zudem der Wert max oben auf dem Berechnungsstack abgelegt werden. Die anschließende Inkrementierung ist nötig, da die möglichen Fälle bei 0 beginnen, ein Sprungindex aber mindestens den Wert 1 haben muß.

Die letzte zu betrachtende Möglichkeit liegt vor, falls der aktuelle Parameterwert ein Baum ist, d.h. $s_{pos} \in \hat{S}$. In diesem Fall können wir die erforderlichen Aufgaben elegant lösen. Der Maschinenbefehl deref_node wandelt einen Verweis auf einen Knoten *node* in einem Baum in Verweise auf die Nachfolger von *node* sowie dessen Beschriftung um. Diese Beschriftung wandeln wir mit Hilfe von cons_to_num in eine Zahl um.

$trans_{pateval}[pos](g, adr, max) =$
    eval pos
    deref_node
    cons_to_num

Damit haben wir die Beschreibung der Übersetzungsschemata zur Bestimmung der passenden Alternative einer Funktion, die mittels Pattern Matching definiert ist, abgeschlossen.

5.6 Übersetzung rechter Seiten von Funktionsgleichungen

Die Anforderungen an die Routine, die die Ermittlung des aktuellen Funktionsergebnisses durchführt, wurden bereits in den vorangegangenen Abschnitten sowie in Kapitel 4 genannt:

Zu Beginn der Ausführung liegt auf dem Berechnungsstack ein Kellerrahmen, der neben zwei organisatorischen Zellen alle benötigten Parameterwerte enthält. In der unteren organisatorischen Zelle steht zudem, welche Komponente des Ergebnisses berechnet werden soll. Nach Ausführung der Routine ist der Kellerrahmen entfernt und statt dessen liegt die errechnete Ergebniskomponente auf dem Berechnungsstack.

Kommen wir zuerst zur Behandlung der verschiedene Ergebniskomponenten. Für jede Ergebniskomponente erzeugen wir Maschinenbefehle, die diese jeweils berechnen. Die Auswahl einer dieser verschiedene Routinen erfolgt nun mittels eines indizierten Sprunges. Den Sprungindex erhalten wir aus der unteren organisatorischen Zelle. Hat eine Funktion nur eine Ergebniskomponente, so entfällt die Sprungtabelle.

Seien also $(e_1, \ldots, e_\mu)$ und e aus $RS\_F(S, F, \Delta, Var)$ zu übersetzende rechte Seiten. Die Übersetzung dieser rechten Seiten ist dann

$trans_{RS}[(e_1, \ldots, e_\mu)](adr) =$
    push CSTACK[AFP]
    jump_idx $adr.1$
$adr.1:$ $trans_{EL}[(e_1, \ldots, e_\mu)](adr.1, 1)$
$adr.0:$

im Falle eines Tupels von Ausdrücken und

$$
\begin{array}{l}
\mathit{trans}_{RS}[e](\mathit{adr}) = \\
\qquad \mathit{trans}_{Expr}[e](\mathit{adr}.1, \mathit{direct}, \mathit{out}, 1)
\end{array}
$$

im Falle eines einzelnen Ausdrucks.

Für eine Tupelliste muß nun eine Sprungtabelle erzeugt werden, die das Anspringen der gewünschten Tupelkomponente ermöglicht.

Sei $e_1 \in F\text{–}EXPR\,(S, F, \Delta, Var)^{(\sigma_1 \ldots \sigma_k)}$ ein Funktionsausdruck und sei $m \geq 2$.

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{trans}_{EL}[(e_1, \ldots, e_m)](\mathit{adr}.i, n) =} \\
& \left.\begin{array}{l} \mathsf{jump}\ \mathit{adr}.i \\ \vdots \\ \mathsf{jump}\ \mathit{adr}.i \end{array}\right\} k \text{ mal} \\
& \mathit{trans}_{EL}[(e_2, \ldots, e_m)](\mathit{adr}.(i+1), n+k) \\
& \mathsf{jump}\ \mathit{adr}.0 \\
\mathit{adr}.i: & \mathit{trans}_{Expr}[e_1](\mathit{adr}.i.1, \mathit{direct}, \mathit{out}, n) \\
\multicolumn{2}{l}{\mathit{trans}_{EL}[(e_1)](\mathit{adr}.i, n) =} \\
& \left.\begin{array}{l} \mathsf{jump}\ \mathit{adr}.i \\ \vdots \\ \mathsf{jump}\ \mathit{adr}.i \end{array}\right\} k-1 \text{ mal} \\
\mathit{adr}.i: & \mathit{trans}_{Expr}[e_1](\mathit{adr}.i.1, \mathit{direct}, \mathit{out}, n)
\end{array}
$$

In der Übersetzungsfunktion trans_{EL} wird ein Parameter n verwendet. Die Bedeutung dieses Parameters wollen wir anhand des nachfolgenden Beispiels erläutern.

Beispiel 5.20

Gegeben seien die Funktionsdeklarationen, die in Abbildung 5.29 wiedergegeben sind.

```
f1 : INTEGER -> INTEGER x BOOLEAN;
f2 : INTEGER -> REAL;
f3 : INTEGER -> REAL x CHARACTER.
```

Abb. 5.29: Deklaration der Funktionen `f1`, `f2` und `f3`.

Nehmen wir nun an, wir wollen von einer Funktion mit rechter Seite `[f1(x),f2(x),f3(x)]` die 5. Komponente berechnen lassen. Dazu müssen wir die Funktion `f3` aufrufen und von dieser die 2. Komponente ermitteln lassen. Um die korrekte Ermittlung der Komponentennummer zu gewährleisten, muß bei der Übersetzung des Ausdrucks `f3(x)` bekannt sein, daß vor diesem Ausdruck andere Funktionsaufrufe stehen, die insgesamt drei Ergebniskomponenten liefern. Bei der Übersetzung muß also die Umgebung mit berücksichtigt werden. □

Die Angabe dieser Umgebung wird hier durch den Parameter n übernommen. Er gibt an, wieviel Ergebniskomponenten die links von dem aktuell betrachteten Ausdruck stehenden Ausdrücke liefern. Kommen wir nun zur Übersetzung eines einzelnen Ausdrucks.

Wie bereits in Abschnitt 5.1 erläutert, muß bei Parameterzugriffen darauf geachtet werden, ob diese direkt bei der Funktionsausführung oder zu einem späteren Zeitpunkt erfolgen.

Dieser zweite Fall tritt ein, wenn der Parameterzugriff in eine Routine eingebunden ist, die durch ein Funktionselement referenziert wird. Diesem Punkt wird in den folgenden Übersetzungsfunktionen mit dem Parameter $modus \in \{direct, closure\}$ Rechnung getragen. Falls $modus = direct$, so kann für einen Parameterzugriff der Befehl eval erzeugt werden. Andernfalls findet der eval_param–Befehl Verwendung.

Der Parameter $loc \in \{in, out\}$ spielt im Zusammenhang mit den Ergebniskomponenten eines Ausdrucks eine Rolle und gibt die Bedeutung des Parameters n an. Ist $loc = out$, so gibt n an, wieviel Ergebniskomponenten die links von dem aktuell betrachteten Ausdruck stehenden Ausdrücke liefern. Ist $loc = in$, so gibt n direkt an, die wievielte Komponente des aktuellen Ausdrucks bestimmt werden soll. Dieser Fall ist notwendig, wenn die Parameter eines Funktionsaufrufs bestimmt werden müssen.

Für einen einzelnen Ausdruck gibt es nun mehrere Alternativen bezüglich seines Aufbaus. An dieser Stelle sei nochmals an die Definition 3.8 (Seite 29) erinnert, in der die Ausdrücke definiert wurden.

$trans_{Expr}[(e_1\ op\ e_2)](adr, modus, loc, n) =$
  $trans_{Expr}[e_1](adr.1, modus, loc, n)$
  $trans_{Expr}[e_2](adr.2, modus, loc, n)$
  $trans_{op}[op](typ(e_1))$

$trans_{Expr}[const](adr, modus, loc, n) =$
  if $typ(const) \in \{bool, int, card, char\}$
    push_const $(Y_{typ(const)} : const)$
  otherwise
    push_const $(Y_{Cons} : const, \delta\_to\_n(const))$
    make_node 0

Erinnern wir uns an dieser Stelle, daß die Variablen so durchnumeriert sind, daß der Index des Variablennamens genau seiner Parameterposition entspricht. Diese Eigenschaft haben wir auch bei der Neueinführung von Parametern durch Muster berücksichtigt. Die für einen Parameterzugriff benötigte Parameternummer, die angibt, an der wievielten Stelle im aktuellen Kellerrahmen der gewünschte Wert steht, ergibt sich folglich direkt aus dem Index des Variablennamens.

$trans_{Expr}[x_i](adr, modus, loc, n) =$
  if $modus = direct$
    eval i
  otherwise
    eval_param i

$trans_{Expr}[a\ (e_1, \ldots, e_m)](adr, modus, loc, n) =$
  if $a \in \Delta^{s_1 \ldots s_k, \sigma}$
    push_const $(Y_{Cons} : a, \delta\_to\_n(a))$
    $trans_{ApplDir}[(e_1, \ldots, e_m)](adr, modus)$
    make_node k
  if $a \in F^{(s_1 \ldots s_k, \sigma_1 \ldots \sigma_l)}$
    if $loc = out$
      push CSTACK[AFP]

$sub\_if\_necessary(n)$
otherwise
push_const $(Y_{num} : n)$
push AFP
if $eval(f) = CbV$
$trans_{ApplDir}[(e_1, \ldots, e_m)](adr, modus)$
otherwise
$trans_{ApplClos}[(e_1, \ldots, e_m)](adr, modus)$
set_AFP k
call $adr\_fkt(a)$

Die Hilfsfunktion *sub_if_necessary* dient dazu, im Falle einer Tupelliste ggf. eine Korrektur an der Angabe, welche Ergebniskomponente der angerufenen Routine ausgewertet werden soll, vorzunehmen.

$sub\_if\_necessary(n) =$
if $(n \geq 2)$
push_const $(Y_{num} : (n - 1))$
sub

Mit den Funktionen $trans_{ApplDir}$ und $trans_{ApplCons}$ werden die Parameter eines Konstruktors oder einer Funktion ermittelt. Im Falle von $trans_{ApplDir}$ werden die Werte direkt ausgerechnet.

$trans_{ApplDir}[(e_1, \ldots, e_m)](adr, modus) =$
$trans_{Appl'}[e_1](adr.1, modus)$
$trans_{Appl'}[e_2](adr.2, modus)$
$\vdots$
$trans_{Appl'}[e_m](adr.m, modus)$

Die Funktion $trans_{ApplClos}$ bewirkt, für einen Parameter ein Funktionselement angelegt wird und die Routine zur Ermittlung des Parameterwerts erzeugt wird. Die Übersetzung der Parameterausdrücke erfolgt dann mit $modus = closure$.

$trans_{ApplClos}[(e_1, \ldots, e_m)](adr, modus) =$

	push_adr $(F : adr.1, ort)$
	jump $adr.(m+1)$
$adr.1$:	$trans_{Appl'}[e_1](adr.1, closure)$
	ret_param
$adr.(m+1)$:	push_adr $(F : adr.2, ort)$
	jump $adr.(m+2)$
$adr.2$:	$trans_{Appl'}[e_2](adr.2, closure)$
	ret_param
	$\vdots$
$adr.(m+m-1)$:	push_adr $(F : adr.m, ort)$
	jump $adr.(m+m)$

$adr.m:$ $\quad trans_{Appl'}[e_m](adr.m, closure)$
ret_param
$adr.(m+m):$

wobei $ort = (\textbf{if } modus = direct \textbf{ then } \mathsf{AFP} \textbf{ else } \mathsf{ENV})$

Die Übersetzungsfunktion $trans_{Appl'}$ verwenden wir, um die einzelnen Komponenten eines Parameterwertes zu bestimmen. Sei $e \in F\text{–}EXPR\,(S, F, \Delta, Var)^{(s_1 \ldots s_n)}$ mit $n \geq 1$ und $s_1, \ldots, s_n \in S$ der betrachtete Parameter.

$$
\begin{aligned}
&trans_{Appl'}[e](adr, modus) = \\
&\qquad trans_{Expr}[e](adr.1, modus, in, 1) \\
&\qquad trans_{Expr}[e](adr.2, modus, in, 2) \\
&\qquad\quad \vdots \\
&\qquad trans_{Expr}[e](adr.n, modus, in, n)
\end{aligned}
$$

Hiermit haben wir die Beschreibung der Übersetzung von NoName*–Programmen mit gewöhnlichen Funktionen abgeschlossen. In den folgenden beiden Abschnitten wenden wir uns der Übersetzung von tree transducern zu.

5.7 Übersetzung von tree transducern

In den vorhergegangenen Abschnitten haben wir uns im wesentlichen mit der Übersetzung von gewöhnlichen Funktionen beschäftigt. In diesem Abschnitt wollen wir nun die Übersetzung von macro attributed tree transducern (mat) betrachten.

Den Aufbau und die Arbeitsweise eines mats haben wir bereits in frühreren Abschnitten diskutiert. Auch zur Umsetzung dieses Konzepts im Rahmen unserer Implementierung haben wir bereits einiges gesagt. Im folgenden wollen wir diese Punkte nur kurz wiederholen und ansonsten auf die entsprechenden Abschnitte verweisen.

- In Abschnitt 3.1 haben wir den strukturellen Aufbau eines mats sowie dessen Arbeitsweise erläutert. Der wesentliche Punkt dabei ist, daß mit Hilfe von Pfadangaben Bewegungen auf dem um das Wurzelsymbol erweiterten Eingabebaum durchgeführt werden.
- Diese Pfadangaben werden in der Maschinenumsetzung in Pfadverweiselementen verwaltet. Durch eigene Maschinenbefehle können diese Pfadangaben verändert werden (vgl. Abschnitt 4.3 auf Seite 46 und Abschnitt 4.2.1 auf Seite 61).

Aus diesen Punkten können wir die Vorgehensweise bei der Übersetzung von mat Funktionen ableiten. Im wesentlichen werden mat Funktionen genau so übersetzt wie auch gewöhnliche Funktionen. Für jeden Funktionsaufruf wird ein Kellerrahmen auf dem Berechnungsstack angelegt und nach Abarbeitung des Funktionsaufrufs wieder entfernt. Der erste Parameter — das Rekursionsargument — ist immer ein Pfadverweiselement. Die Wahl der Auswertungsstrategie wirkt sich auf diesen Parameter nicht aus. Die weiteren Parameter werden genau so behandelt, wie auch bei gewöhnlichen Funktionen.

Ein Problem ergibt sich nun noch: Wird eine mat Funktion f_{tr} von einer gewöhnlichen Funktion f aufgerufen, so befindet sich im Kellerrahmen an der ersten Parameterposition ein Verweiselement auf den Eingabenbaum. Erfolgt ein Aufruf von f_{tr} aber von einer anderen mat Funktion desselben Spezifikationsblocks, so ist der erste Parameter ein Pfadverweiselement für den erweiterten Eingabebaum. Diese Problematik findet ihren Ursprung in der Sprachbeschreibung [EGH+94], die in diesem Punkt ungenau ist. Sie läßt eine solche polymorphe Verwendung dieser ersten Parameterposition zu.

Wir lösen dieses Problem nun, indem wir vor die eigentliche Funktionsroutine für f_{tr} einige weitere Maschinenbefehle setzen, die die Umwandlung des Verweises auf den Eingabebaum in ein Pfadverweiselement für den erweiterten Eingabebaum durchführen.

Wir erhalten also für eine mat Funktion den in Abbildung 5.30 skizzierten Aufbau des zugehörigen Maschinenprogramms.

Erweitern des Eingabebaums um das Wurzelsymbol. Umwandlung von Verweis in Pfadangabe.	$\leftarrow$ Einsprungpunkt bei Aufruf durch eine externe Funktion.
Eigentliche mat Funktion	$\leftarrow$ Einsprungpunkt bei Aufruf durch eine Funktion desselben mat.

Abb. 5.30: Aufbau des Maschinenprogramms für eine mat Funktion.

Betrachten wir nun die Übersetzungsfunktion für eine mat Funktion $f_s \in F_{syn}$, also einer synthetischen Funktion eines tree transducers. Sämtliche Regeln in $\mathcal{R}$, die f_s definieren, seien

$$\begin{aligned} f_s(\langle z : d_1\rangle, x_1, \ldots, x_n) &= rs_1 \\ f_s(\langle z : d_2\rangle, x_1, \ldots, x_n) &= rs_2 \\ &\vdots \\ f_s(\langle z : d_k\rangle, x_1, \ldots, x_n) &= rs_k \end{aligned}$$

wobei für alle $1 \leq i < j \leq k$ die Aussage $\delta\_to\_n(d_i) < \delta\_to\_n(d_j)$ gilt.

Die Übersetzungsfunktionen für f_s sind im folgenden angebenen. Die Funktion $trans_{syn}$ erzeugt die Befehle, die nötig sind, aus einem Verweis auf den Eingabebaum ein Pfadverweiselement für den erweiterten Eingabebaum zu erzeugen.

$trans_{syn}[f_s](adr) =$

push_const $(Y_{cons} : root, 0)$
eval 1
make_node 1
make_stack
write_back 1
$adr.1 :$ $trans_{syn'}[f_s](adr.1)$

Anschließen erfolgt die Übersetzung der eigentlichen Funktion f_s. Auch hier steht vor der Berechnung des Funktionsergebnisses die Bestimmung der anzuwendenen Alternative.

Diese Bestimmung ist aber leicht durchführbar: Wir betrachten einfach den Konstruktor des Knotens im erweiterten Eingabebaum, auf den die aktuelle Pfadangabe gerade zeigt. Diesen Konstruktor wandeln wir in eine Zahl um und führen einen indizierten Sprung zur passenden Alternative durch.

$trans_{syn'}[f_s](adr) =$

```
                eval 1
                heap_top
                extract 0
                cons_to_num
                push_const (Y_num : 1)
                add
                jump_idx adr.(k+1)
   adr.(k+1) :  jump adr.1
                jump adr.2
                   ⋮
                jump adr.k
   adr.1 :      trans_tRS[rs_1](adr.1)
                jump adr.0
   adr.2 :      trans_tRS[rs_2](adr.2)
                jump adr.0
                   ⋮
   adr.k :      trans_tRS[rs_k](adr.k)
   adr.0 :      cleanup
                ret
```

Kommen wir nun zur Übersetzung einer inheriten Funktion $f_h \in F_{inh}$. Sämtliche Regeln in $\mathcal{R}$, die f_h definieren, seien

$$\begin{aligned}
&f_h(\langle z : d_1\rangle.1, x_1, \ldots, x_n) = rs_{11}\\
&\quad\vdots\\
&f_h(\langle z : d_1\rangle.n_1, x_1, \ldots, x_n) = rs_{1n_1}\\
&f_h(\langle z : d_2\rangle.1, x_1, \ldots, x_n) = rs_{21}\\
&\quad\vdots\\
&f_h(\langle z : d_2\rangle.n_2, x_1, \ldots, x_n) = rs_{2n_2}\\
&\quad\vdots\\
&f_h(\langle z : d_k\rangle.1, x_1, \ldots, x_n) = rs_{k1}\\
&\quad\vdots\\
&f_h(\langle z : d_k\rangle.n_k, x_1, \ldots, x_n) = rs_{kn_n}
\end{aligned}$$

wobei für alle $1 \leq i < j \leq k$ die Aussage $\delta\_to\_n(d_i) < \delta\_to\_n(d_j)$ gilt.

Die Übersetzungsfunktionen für f_h sind im folgenden angegeben. Eine vorgeschaltete Routine zur Konvertierung von Verweis in Pfadangabe ist hier nicht nötig. Dies liegt daran, daß inherite Funktionen nur von anderen mat Funktionen desselben Spezifikationsblocks aufgerufen werden können und damit der erste Parameter immer ein Pfadverweiselement ist. Aus diesem Grund erzeugt die Übersetzungsfunktion $trans_{inh}$ auch keine eigenen Maschinenbefehle.

$trans_{inh}[f_h](adr) =$
$adr.1:$ $trans_{inh'}[f_h](adr.1)$

Die Bestimmung des anzuwendenden Falls, die auch hier notwendig ist, gestaltet sich etwas aufwendiger. Zuerst müssen wir die aktuelle Pfadangabe z' zu z verkürzen und uns dabei merken, der wievielte Nachfolger der Knoten von z' in Bezug auf den Knoten von z war. Dann ermitteln wir die Konstruktornummer des Knotens von z. Mit der Knotennummer führen wir einen ersten indizierten Sprung durch. Mit Hilfe der gespeicherten Nachfolgernummer wird dann ein zweiter indizierter Sprung zur passenden Alternative durchgeführt.

$trans_{inh'}[h](adr) =$

```
                 eval 1
                 heap_top
                 eval 1
                 heap_pop
                 eval 1
                 heap_top
                 get_son_nr
                 eval 1
                 heap_top
                 extract 0
                 cons_to_num
                 push_const (Y_num : 1)
                 add
                 jump_idx adr.(k+1)
adr.(k+1) :      jump adr.1
                 jump adr.2
                   ⋮
                 jump adr.k
adr.1 :          jump_idx adr.1.0
adr.1.0 :        jump adr.1.1
                 jump adr.1.2
                   ⋮
                 jump adr.1.n_1
adr.1.1 :        trans_tRS[rs_11](adr.1.1)
                 jump adr.0
adr.1.2 :        trans_tRS[rs_12](adr.1.2)
                 jump adr.0
                   ⋮
adr.1.n_1 :      trans_tRS[rs_1n_1](adr.1.n_1)
                 jump adr.0
```

$$
\begin{array}{ll}
 & \quad\vdots \\
adr.k: & \mathsf{jump\_idx}\ adr.k.0 \\
adr.k.0: & \mathsf{jump}\ adr.k.1 \\
 & \mathsf{jump}\ adr.k.2 \\
 & \quad\vdots \\
 & \mathsf{jump}\ adr.k.n_k \\
adr.k.1: & trans_{tRS}[rs_{k1}](adr.k.1) \\
 & \mathsf{jump}\ adr.0 \\
adr.k.2: & trans_{tRS}[rs_{k2}](adr.k.2) \\
 & \mathsf{jump}\ adr.0 \\
 & \quad\vdots \\
adr.k.n_k: & trans_{tRS}[rs_{kn_k}](adr.k.n_k) \\
adr.0: & \mathsf{cleanup} \\
 & \mathsf{ret}
\end{array}
$$

Im nächsten Abschnitt befassen wir uns mit der Bestimmung des Funktionsergebnisses einer mat Funktion.

5.8 Übersetzung rechter Seiten von tree transducer Funktionsgleichungen

Für die Übersetzung von Funktionsresultaten von tree transducern gilt im Prinzip genau dasselbe, wie auch bei der Übersetzung von allgemeinen Funktionsresultaten. Die einzigen Unterschiede ergeben sich darin, daß zum einen auch Ausdrücke der Form $id\langle z\rangle.k$ übersetzt werden müssen, und zum anderen, daß bei Aufrufen weiterer tree transducer Funktionen die Bewegungen im erweiterten Eingabebaum durch Verändern der Pfadangabe durchzuführen sind.

Die Behandlung von Ergebnistupeln gestaltet sich genau so, wie wir es bereits bei gewöhnlichen Funktionen kennengelernt haben.

$$
\begin{array}{ll}
\multicolumn{2}{l}{trans_{tRS}[(e_1,\ldots,e_\mu)](adr) =} \\
 & \mathsf{push\ CSTACK[AFP]} \\
 & \mathsf{jump\_idx}\ adr.1 \\
adr.1: & trans_{tEL}[(e_1,\ldots,e_\mu)](adr.1,1) \\
adr.0: &
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{trans_{tRS}[e](adr) =} \\
 & trans_{tExpr}[e](adr.1, direct, out, 1)
\end{array}
$$

Sei im folgenden e_1 ein tree transducer Ausdruck mit der Sorte $(s_1 \ldots s_k)$ und $m \geq 2$.

$$
\begin{array}{l}
trans_{tEL}[(e_1,\ldots,e_m)](adr.i,n) = \\
\qquad \left.\begin{array}{l}
\mathsf{jump}\ adr.i \\
\vdots \\
\mathsf{jump}\ adr.i
\end{array}\right\}\ k \text{ mal}
\end{array}
$$

$$
\begin{array}{rl}
 & trans_{tEL}[(e_2,\ldots,e_m)](adr.(i+1),n+k) \\
 & \mathsf{jump}\ adr.0 \\
adr.i: & trans_{tExpr}[e_1](adr.i.1,direct,out,n)
\end{array}
$$

$$
\begin{array}{rl}
\multicolumn{2}{l}{trans_{tEL}[(e_1)](adr.i,n) =} \\
 & \left.\begin{array}{l}\mathsf{jump}\ adr.i \\ \vdots \\ \mathsf{jump}\ adr.i\end{array}\right\} k-1 \text{ mal} \\
adr.i: & trans_{tExpr}[e_1](adr.i.1,direct,out,n)
\end{array}
$$

An der Bedeutung der Parameter *modus* und *loc* ergibt sich ebenfalls keine Änderung. Die Übersetzung von Ausdrücken erfolgt also in der gewohnten Art und Weise.

$trans_{tExpr}[(e_1\ op\ e_2)](adr,modus,loc,n) =$
- $trans_{tExpr}[e_1](adr.1,modus,loc,n)$
- $trans_{tExpr}[e_2](adr.2,modus,loc,n)$
- $trans_{op}[op](typ(e_1))$

$trans_{tExpr}[const](adr,modus,loc,n) =$

if $typ(const) \in \{bool,int,card,char\}$
  push_const $(Y_{typ(const)} : const)$
otherwise
  push_const $(Y_{Cons} : const, \delta\_to\_n(const))$
  make_node 0

$trans_{tExpr}[x_i](adr,modus,loc,n) =$

if $modus = direct$
  eval i
otherwise
  eval_param i

Bei der Übersetzung von Funktionsaufrufen müssen wir noch eine weitere Möglichkeit beachten: Rufen wir eine mat Funktion f_{tr} auf, so müssen wir unterscheiden, ob wir diese mit $\langle z\rangle$ bzw. $\langle z\rangle.k$ als Rekursionsargument aufrufen oder nicht. Je nachdem wählen wir bei dem zu erzeugenden call–Befehl die passende Programmadresse.

Falls das Rekursionsargument $\langle z\rangle$ bzw. $\langle z\rangle.k$ ist, muß außerdem die aktuelle Pfadangabe kopiert und gegebenenfalls modifiziert werden, bevor diese als erster Parameterwert im neu anzulegenden Kellerrahmen verwendet werden kann.

$trans_{tExpr}[a\ (e_1,\ldots,e_m)](adr,modus,loc,n) =$

if $a \in \Delta^{s_1\ldots s_k,\sigma}$
  push_const $(Y_{Cons} : a, \delta\_to\_n(a))$
  $trans_{tApplDir}[(e_1,\ldots,e_m)](adr,modus)$
  make_node k
if $a \in F^{(s_1\ldots s_k,\sigma_1\ldots\sigma_l)}$
  if $loc = out$
    push CSTACK[AFP]
    $sub\_if\_necessary(n)$

otherwise
  push_const $(Y_{num} : n)$
push CSTACK[AFP]
$sub\_if\_necessary(n)$
push AFP
if $e_1 = \langle z \rangle$
  if $modus = direct$
    eval i
  otherwise
    eval_param i
  make_copy
  setzte μ auf 2
if $e_1 = \langle z \rangle.son\_nr$
  if $modus = direct$
    eval i
  otherwise
    eval_param i
  make_copy
  dup
  heap_top
  extract son_nr
  heap_push
  setzte μ auf 2
otherwise
  setze μ auf 1
if $eval(f) = CbV$
  $trans_{tApplDir}[(e_\mu, \ldots, e_m)](adr, modus)$
otherwise
  $trans_{tApplClos}'[(e_\mu, \ldots, e_m)](adr, modus)$
set_AFP k
if $e_1 = \langle z \rangle.son\_nr$ **or** $e_1 = \langle z \rangle$
  call $adresse(a).1$
otherwise
  call $adresse(a)$

Die Behandlung von Parametern der Form $id(\langle z \rangle)$ bzw. $id(\langle z \rangle.k)$ ist einfach. Wir müssen dazu nur den obersten Eintrag des aktuellen Pfadverweislements betrachten. Dieser Eintrag ist ein Verweis auf den gewünschten (Teil–)baum. Im Falle von $id(\langle z \rangle.k)$ müssen wir zudem noch den k–ten Nachfolger auswählen. Diese Aufgabe kann aber mit dem Maschinenbefehl extract k einfach gelöst werden.

$trans_{tExpr}[id(\langle z \rangle)](adr, modus, loc, n) =$
if $modus = direct$
  eval 1
otherwise
  eval_param 1
heap_top

$$
\begin{array}{l}
trans_{tExpr}[id(\langle z\rangle.k)](adr, modus, loc, n) = \\
\qquad \textbf{if } modus = direct \\
\qquad\quad \mathsf{eval}\ 1 \\
\qquad \textbf{otherwise} \\
\qquad\quad \mathsf{eval\_param}\ 1 \\
\qquad \mathsf{heap\_top} \\
\qquad \mathsf{extract}\ k
\end{array}
$$

Die Übersetzung der Parameterlisten erfolgt genau wie bei gewöhnlichen Funktionen.

$$
\begin{array}{l}
trans_{tApplDir}[(e_1, \ldots, e_m)](adr, modus) = \\
\qquad trans_{tAppl'}[e_1](adr.1, modus) \\
\qquad trans_{tAppl'}[e_2](adr.2, modus) \\
\qquad\quad \vdots \\
\qquad trans_{tAppl'}[e_m](adr.m, modus)
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{trans_{tApplClos}[(e_1, \ldots, e_m)](adr, modus) =} \\
 & \mathsf{push\_adr}\ (F : adr.1, ort) \\
 & \mathsf{jump}\ adr.(m+1) \\
adr.1 : & trans_{tAppl'}[e_1](adr.1, closure) \\
 & \mathsf{ret\_param} \\
adr.(m+1) : & \mathsf{push\_adr}\ (F : adr.2, ort) \\
 & \mathsf{jump}\ adr.(m+2) \\
adr.2 : & trans_{tAppl'}[e_2](adr.2, closure) \\
 & \mathsf{ret\_param} \\
 & \quad \vdots \\
adr.(m+m-1) : & \mathsf{push\_adr}\ (F : adr.m, ort) \\
 & \mathsf{jump}\ adr.(m+m) \\
adr.m : & trans_{tAppl'}[e_m](adr.m, closure) \\
 & \mathsf{ret\_param} \\
adr.(m+m) : &
\end{array}
$$

wobei $ort = (\textbf{if } modus = direct \textbf{ then } \mathsf{AFP} \textbf{ else } \mathsf{ENV})$
Sei $e \in TR\text{–}EXPR(S, F, \Delta, Var)^{(s_1 \ldots s_n)}$ mit $n \geq 1$ und $s_1, \ldots, s_n \in S$.

$$
\begin{array}{l}
trans_{tAppl'}[e](adr, modus) = \\
\qquad trans_{tExpr}[e](adr.1, modus, in, 1) \\
\qquad trans_{tExpr}[e](adr.2, modus, in, 2) \\
\qquad\quad \vdots \\
\qquad trans_{tExpr}[e](adr.n, modus, in, n)
\end{array}
$$

Damit haben wir auch die Beschreibung der Übersetzung von tree transducer Funktionen abgeschlossen.

5.9 Übersetzung der Anfrage

Betrachten wir nun noch die Übersetzung einer Anfrage *anfr*. Im Prinzip kann man eine solche Anfrage als rechte Seite einer Funktion *ziel*, die keine Parameter hat, auffassen. Dieser Ansatz führt dann zu richtigen Ergebnissen, wenn das Ergebnis der Anfrage aus nur einer Komponente besteht. Besteht das Ergebnis der Anfrage aber aus mehreren Komponenten, so ist es notwendig, alle Ergebniskomponenten von *ziel* zu ermitteln. Ein einfacher Funktionsaufruf ermittelt aber immer nur die Ergebniskomponente, die durch den Wert der unteren organisatorischen Zelle des zugehörigen Kellerrahmens festgelegt ist. Durch Anwenden des folgenden Pseudoalgorithmus können aber alle Ergebniskomponenten ermittelt werden. Wir nehmen dabei an, daß $result(anfr) = \sigma_1 \dots \sigma_n$ gilt.

$k \leftarrow 0$;
while $k < n$ **do**
    $k \leftarrow k + 1$;
    Lege Funktionsrahmen mit k in der unteren organisatorischen Zelle auf
    call *ziel_adr*
end_while

Auf diese Art und Weise wird für jede Ergebniskomponente von *ziel* die Auswertung angestoßen. An der Stelle *ziel_adr* befinden sich dabei die Maschinenbefehle, die für *anfr* als Ausdruck mit Hilfe von $trans_{RS}$ erzeugt wurden.

Die Übersetzung einer Anfrage $anfr \in RS\_F(S, F \backslash F_{inh}, \Delta, Var)^{(\sigma_1 \dots \sigma_n)}$ geschieht also in der folgenden Weise.

$trans_{Anfr}[anfr](adr) =$

	push_const $(Y_{num} : 0)$
$adr.1$:	eval -1
	push_const $(Y_{num} : 1)$
	add
	write_back -1
	eval -1
	push_const $(Y_{num} : n)$
	grt
	jump_on_true $adr.2$
	eval -1
	push_const $(Y_{num} : 1)$
	set_AFP 0
	call $adr.3$
	jump $adr.1$
$adr.3$:	$trans_{RS}[anfr](adr.3)$
	cleanup
	ret
$adr.2$:	

Mit der Übersetzung der Anfrage schließen wir die Beschreibung der Übersetzungsfunktionen ab. Im Ausblick (Kapitel 7) geben wir noch einige Ideen, wie die hier vorgestellte Übersetzung verbessert werden kann.

Kapitel 6

Implementierung

In diesem Kapitel beschreiben wir die Umsetzung der vorgestellten Implementierung auf einen realen Rechner. Für diese Umsetzung verwenden wir in diesem Kapitel, der gängigen Sprechweise entsprechend, ebenfalls den Begriff *Implementierung*.

Die Implementierung besteht aus zwei Teilen: dem eigentlichen Übersetzer und dem Maschinensimulator, der eine abstrakte NoName–Maschine simuliert. Der Übersetzer ist in der funktionalen Programmiersprache *Gofer* implementiert. Diese Sprache hat große Ähnlichkeit mit Miranda[1] oder Haskell. Der Maschinensimulator ist in C implementiert.

In Abschnitt 6.1 werden wir die eigentliche Implementierung beschreiben. In Abschnitt 6.2 folgen Hinweise, wie man Übersetzer und Maschinensimulator auf dem eigenen Rechner installiert. Im Abschnitt 6.3 schließlich zeigen wir die Bedienung anhand einer Beispielsitzung.

6.1 Beschreibung der Implementierung

6.1.1 Implementierung des Übersetzers

Der eigentliche Übersetzer, der aus NoName*–Programmen Maschinenprogramme erzeugt, wurde in der funktionalen Programmiersprache Gofer implementiert. Eine erste Fassung des NoName*–Übersetzers wurde in der Sprache Miranda erstellt. Da Miranda jedoch nicht für alle Rechner verfügbar und zudem ein kommerzielles Produkt ist, wurde diese Fassung bald nach Gofer portiert. Gofer ist Public Domain, darf also frei weitergegeben werden und ist zudem auf den meisten Rechnern lauffähig, für die ein C–Compiler existiert. Auf den Disketten, die zu dieser Diplomarbeit gehören, befindet sich neben den Quelltexten des Übersetzers und des Maschinensimulators das gesamte Gofer–System.

Mit dem Beispiel 6.1 wollen wir die Sprache Gofer kurz vorstellen, um dem Leser das bessere Nachvollziehen des implementierten Übersetzers zu ermöglichen. Weitere und wesentlich ausführlichere Informationen zu Gofer finden sich in [Jon91].

[1] Miranda ist eine funktionale Programmiersprache, die von David Turner [Tur85] entwickelt wurde. Das Miranda–System ist ein Warenzeichen der Firma Software Research Ltd., Canterbury, Großbritannien.

Beispiel 6.1 (Programmbeispiel in Gofer)

Betrachten wir das Gofer–Programm aus Abbildung 6.1.

```
data Baum = Knoten Baum Baum Bool | Blatt Int

count :: Baum -> Int
count (Knoten t1 t2 b) = count t1 + count t2
count (Blatt n) = 1

fib :: Int -> Int
fib n = 1, if (n == 0) || (n == 1)
      = fib (n-1) + fib (n-2), otherwise

fak :: Int -> Int
fak 0 = 1
fak n = n * fak_rest, if (n>1)
      = error "Fakultaet nicht fuer negative Zahlen."
        where fak_rest = fak (n-1)
```

Abb. 6.1: Beispiel für ein Gofer–Programm.

Mit der Angabe `data Baum =...` wird ein algebraischer Datentyp definiert, der die Konstruktorsymbole `Knoten` und `Blatt` besitzt (vgl. Beispiel 3.4, Seite 28).

Die erste Zeile jeder Funktionsspezifikation ist die Deklaration. So ist beispielsweise `fib` eine Funktion, die als Eingabe eine ganze Zahl (`Int`) nimmt und als Ausgabe wiederum eine ganze Zahl liefert. Die Angabe der Deklaration ist aber, anders als in NoName, nicht zwingend notwendig.

Die Funktion `count` ist mittels Pattern Matching definiert. Auffällig sind hierbei vor allem die fehlenden Klammern beim Funktionaufruf. Dies liegt daran, daß die Notation der Funktionen in Gofer auf dem λ–Kalkül ([Chu32]) beruht, in dem eine Funktionsanwendung einer Funktion f auf die Parameter x und y in der Form $f\ x\ y$ notiert wird. In der Spezifikation der Funktion `fib` liegt ein Beispiel für eine Funktionsdefinition via IF vor.

Die Definition der Funktion `fak` zeigt eine Reihe weiterer Eigenschaften von Gofer: Pattern Matching muß hier nicht erschöpfend und überschneidungsfrei sein. Die einzelnen Fälle werden der Reihe nach durchlaufen. Aus diesem Grund muß der speziellste Fall am Anfang stehen, der allgemeinste Fall am Ende der Aufschreibungsreihenfolge. Ein Mischen von Pattern Matching und IF ist erlaubt. Mit dem Konstrukt `where` können lokale Variablen eingeführt werden. Mit der Angabe `error "..."` ist es möglich, Fehlermeldungen zu produzieren und damit partielle Funktionen zu realisieren. □

Um den Übersetzer auch sinnvoll einsetzen zu können, arbeitet die implementierte Fassung nicht auf formalisierten NoName*–Programmen sondern auf Programmen in der syntaktischen Version von NoName*. Im anderen Falle hätten wir für die formalisierten NoName*–Programme eine eigene Syntax entwerfen müssen sowie ein Umsetzungsprogramm zwischen der NoName*–Syntax und der Syntax der formalisierten NoName*–Programme.

Der Übersetzer ist, gemäß der Aufgabenstellung, so implementiert, daß er nur korrekte Programme korrekt übersetzt. Es werden nur einige wenige Fehler entdeckt. Folglich werden viele fehlerhafte Programme auch übersetzt, jedoch ist dann eine sinnvolle Abarbeitung des erzeugten Maschinenprogramms nicht zu erwarten.

Die Forderung aus Abschnitt 5.1, daß die Variablen nach dem Schema `x1`, `x2` ... benannt sein müssen, wurde aufgegeben, da eine Umbenennung, die diese Eigenschaft sicherstellt, leicht automatisch durchgeführt werden kann und deswegen mit implementiert ist.

Durch die Teilmengeneigenschaft von NoName* in Bezug auf NoName konnten wir für NoName* eine einfachere Syntax als die ursprüngliche Syntax, die in [EGH+94] beschrieben ist, entwerfen. Im Gegenzug haben wir die Anfrage, die ja ein Teil eines NoName*–Programms ist, mit in den Programmtext aufgenommen. Vor dem eigentlichen Programm steht das Schlüsselwort `GOAL` gefolgt vom Ausdruck der Anfrage. Die so modifizierte kontextfreie Syntax von NoName* ist im Anhang A abgedruckt.

Die Übersetzung selbst wird anhand eines abstrakten Syntaxbaums durchgeführt. Um diesen aufzubauen, ist es notwendig, den Eingabetext gemäß der kontextfreien Syntax zu parsen. Der dazu notwendige Parser läßt sich mit Hilfe des Programmpaketes *Ratatosk* (vgl. [Mog93]), leicht erzeugen. Der erzeugte Parser liegt dann in der Programmiersprache Gofer vor, so daß er sich nahtlos in den eigentlichen Übersetzer integrieren läßt.

Die Parserdefinition in Ratatosk setzt sich aus den Angaben der kontextfreien Grammatik und Erkennungsregeln zusammen. Die Erkennungsregeln — eingefaßt in Dollarsymbole — geben an, welches Ergebnis zurückzuliefern ist, nachdem die entsprechende Regel der kontextfreien Grammatik erkannt wurde. Dabei kann mit `x1`, `x2`, ... auf Resultate zugegriffen werden, die bereits durch vorangegangenes Parsing berechnet wurden.

Beispiel 6.2

Betrachten wir zuerst einen Ausschnitt der Scannerdefinition, die sich aus der Angabe von regulären Ausdrücken zusammensetzt.

```
kommaToken     = ",";
klToken        = "(";
krToken        = ")";
infixToken     = "-"|"*"|"/"|">="|"<="|"<>"|"and"|"or"|"div";
numtoken       = (@|"-") ziffer ziffer*;
idToken        = (["a"-"z"]|["A"-"Z"])*;
where
ziffer         = ["0"-"9"];
```

Abb. 6.2: Scannerdefinition in Ratatosk.

Eine Zahl (`numToken`) ist also eine nichtleere Folge von Ziffern. Das Zeichen `@` steht für das leere Wort ε. Vor einer Zahl kann sich also ein Minus als Vorzeichen befinden.

Im folgenden findet sich eine vereinfachte Version der Parserbeschreibung für rechte Seiten von Funktionsgleichungen.

```
ExprList     -> Expr kommaToken ExprList     $ (EL1 x1 x3) $
             |  Expr                         $ (EL2 x1) $
Expr         -> SecExpr                      $ (E1 x1) $
             |  SecExpr infixToken SecExpr   $ (E2 x1 x2 x3) $
SecExpr      -> idToken                      $ (SE1 x1) $
```

```
             |  numToken                    $ (SE2 x1) $
             |  klToken Expr krToken        $ (SE3 x2) $
             |  IdToken Application         $ (SE4 x1 x2) $
Application  -> klToken krToken             $ (A1) $
             |  klToken ExprList krToken    $ (A2 x2) $
```

Abb. 6.3: Parserdefinition in Ratatosk.

Die erste Zeile beispielsweise besagt: Falls eine Liste von Ausdrücken (`ExprList`) erkannt werden soll, und ein Ausdruck (`Expr`) gefolgt von einem Komma (`kommaToken`) und einer Liste von Ausdrücken (`ExprList`) erkannt wurde, dann wird der Ausdruck `EL1`(t_1,t_2) zurückgegeben. Dabei ist t_1 der Wert, der beim Erkennen des Ausdrucks errechnet wurde und t_2 der Wert, der beim Erkennen der Liste von Ausdrücken bestimmt wurde. □

Aus dem abstrakten Syntaxbaum wird zuerst eine Umgebung `rho` abgeleitet. Eine Umgebung ist eine Liste von Umgebungseinträgen, wobei für einen solchen Umgebungseintrag folgende Alternativen möglich sind:

- `AlgDef` *id def*
  Der Bezeichner *id* ist der Name eines algebraischen Datentyps mit Definition *def*.
- `CbN` *id*
  Die Funktion *id* ist mit der Auswertungsstrategie Call–by–Name auszuwerten.
- `CbV` *id*
  Die Funktion *id* ist mit der Auswertungsstrategie Call–by–Value auszuwerten.
- `FuncDecl` *id def*
  Der Bezeichner *id* ist der Name einer Funktion mit Typ *def*.
- `FktAdresse` *id adr*
  Realisierung der Adressfunktion: Der Funktion *id* wird die Adresse *adr* zugewiesen.
- `Konstr` id_1 id_2 *nr typ*
  Der Bezeichner id_1 ist ein Konstruktor des algebraischen Datentyps id_2 und hat die Konstruktornummer *nr*. Die Definition der Nachfolger von id_1 ist in *typ* angegeben.
- `Wurzelsymbol` *id*
  Der Bezeichner *id* wird als Wurzelsymbol verwendet.
- `MatDecl` *id art* typ_1 typ_2
  Der Bezeichner *id* ist der Name einer tree transducer Funktion. Mit *art* wird festegelegt, ob es sich dabei um eine synthetische (`SYN_FKT`) oder inherite (`INH_FKT`) Funktion handelt. Die Angaben typ_1 und typ_2 bestimmen die Ein– bzw. Ausgabetypen der Funktion.
- `DefaultEval` *ev*
  Ist für eine Funktion keine Auswertungsstrategie angegeben, so soll sie mit der Strategie *ev* (`CallByName` oder `CallByValue`) ausgewertet werden.

Die baumstrukturierten Adressen werden in Form von Zahlenlisten verwaltet.

Die eigentliche Übersetzung erfolgt dann ebenfalls anhand des abstrakten Syntaxbaums. Allerdings enthält der abstrakte Syntaxbaum die zur Übersetzung benötigten Informationen in viel kleineren Einheiten als die formalisierte Version von NoName*–Programmen.

So wird beispielsweise in der formalisierten Version von NoName* eine Liste von Ausdrücken als $(e_1, \ldots, e_n)$ notiert. Im abstrakten Syntaxbaum hingegen liegt diese Information, wie aus Beispiel 6.2 ersichtlich, in Baumform vor. Um nun beispielsweise die Länge der Liste — hier also n — zu bestimmen, ist es notwendig, den Baum rekursiv bis zu den Blättern zu durchlaufen.

Da also die benötigten Informationen im abstrakten Syntaxbaum stärker verteilt sind, ist es nötig, eine Reihe von Hilfsfunktionen einzusetzen.

Die erzeugten Maschinenbefehle sind Zeichenfolgen, das erzeugte Maschinenprogramm ist eine Liste von Zeichenfolgen. Die Marken werden wie normale Maschinenbefehle in das erzeugte Maschinenprogramm mit aufgenommen. Im wesentlichen sind die Übersetzungsfunktionen aus Kapitel 5 unverändert übernommen worden, wie die folgenden Abbildungen 6.4 und 6.5 anschaulich anhand der Übersetzungsfunktion $trans_{RS}$ zeigen.

$$
\begin{array}{ll}
\multicolumn{2}{l}{trans_{RS}[(e_1, \ldots, e_\mu)](adr, adr\_fkt) =} \\
 & \mathsf{push\ CSTACK[AFP]} \\
 & \mathsf{jump\_idx}\ adr.1 \\
adr.1: & trans_{EL}[(e_1, \ldots, e_\mu)] \\
 & \quad (adr.2, adr\_fkt, 1) \\
adr.0: & \\
\multicolumn{2}{l}{trans_{RS}[e](adr, adr\_fkt) =} \\
\multicolumn{2}{l}{\quad trans_{Expr}[e](adr.1, adr\_fkt,} \\
\multicolumn{2}{l}{\qquad direct, out, 1)}
\end{array}
$$

Abb. 6.4: Formale Version von $trans_{RS}$

```
trans_RS ( (RS1 el), adr, rho )
  = [PUSH CSTACK [AFP]"] ++
    ["JUMP_IDX "++(adrshow(adr++[1]))]++
    [(adrshow(adr++[1]))++":"]++
    (trans_EL(el,(adr++[2]),rho,1))++
    [(adrshow(adr++[0]))++":"]
trans_RS ( (RS2 e), adr, rho)
  = trans_E(e,(adr++[1]),rho,
        Direct,Out,1)
```

Abb. 6.5: Gofer–Version von $trans_{RS}$

Zu Beginn dieses Abschnitts haben wir erwähnt, daß wir die Forderung nach Variablennamen der Form `x1`, `x2`, … aufgegeben haben. Bevor nun eine Funktionsdefinition übersetzt wird, führen wir immer eine Variablenumbenennung durch, die dann zu Variablennamen der obigen Form führt und somit die gewünschte Eigenschaft sicherstellt.

Ein Problem stellt sich in der Realisierung von Zufallszahlen, die wir eigentlich zur Bestimmung einer Reihenfolge der Fälle einer via IF definierten Funktion brauchen. Da es in Gofer keine Möglichkeit gibt, einen hinreichend zufälligen Startwert (wie z.B. den Stand der internen Uhr des Rechners) zu ermitteln, kann nur ein Zufallsgenerator mit festem Startwert verwendet werden. Dies hat zur Folge, daß ein und dieselbe IF–Fallunterscheidung immer genau gleich übersetzt wird, obwohl dies der ursprünglichen Intention der Sprachdefinition (vgl. [EGH$^+$94], Seite 18) entgegenläuft.

Die Aufteilung des Übersetzers in die einzelnen Dateien sieht wie folgt aus.

`nnm.def`	Enthält die Definition der benötigten algebraischen Datentypen (z.B. für den abstrakten Syntaxbaum und die Umgebung).
`nnm.tok`	Scanner–Definition für NoName*–Programme in Ratatosk–Syntax.
`nnm.sca`	Aus `nnm.tok` erzeugter Scanner in der Programmiersprache Gofer.
`nnm.gra`	Parser–Definition für NoName*–Programme in Ratatosk–Syntax.

`nnm.par` Aus `nnm.gra` erzeugter Parser in der Programmiersprache Gofer.

`parseerr.gs` Beinhaltet Funktionen zur Behandlung von Fehlern beim Parsen.

`nnm.f` Enthält Hilfsfunktionen, die es ermöglichen, von Gofer aus auf Dateien lesend und schreibend zugreifen zu können.

`nnm.tr` Beinhaltet den eigentlichen Übersetzer sowie die Funktionen zur Bestimmung der Umgebung.

`nnm.ub` Enthält Funktionen, mit denen der Benutzer den Übersetzer aktivieren kann.

`nnm.gp` Projektdatei, die das Einladen der benötigten Dateien in der richtigen Reihenfolge veranlaßt.

Um den Übersetzer nutzen zu können, muß zuerst mit

`gofer + nnm.gp -h200000`

der Übersetzer aktiviert werden. Der Benutzer kann dann den Übersetzer mit zwei verschieden Funktionen ansprechen. Der Aufruf

`testcompile` ⟨*Eingabe*⟩ ⟨*Auswertungsstrategie*⟩

stößt den Übersetzungsvorgang für das Programm, das in der Datei ⟨*Eingabe*⟩ abgespeichert ist, an. Der Dateiname ⟨*Eingabe*⟩ muß in Hochkommata eingeschlossen sein. Mit der Angabe ⟨*Auswertungsstrategie*⟩ wird festgelegt, mit welcher Strategie die Funktionen ausgewertet werden sollen, für die keine Auswertungsstrategie explizit angegeben worden ist. Dabei können die Einstellungen `CallByValue` und `CallByName` verwendet werden. Eine Angabe für ⟨*Auswertungsstrategie*⟩ ist zwingend erforderlich.

Die Ausgabe dieses Funktionsaufrufs besteht aus dem eingelesenen Programmtext, dem erzeugten abstrakten Syntaxbaum, der berechneten Umgebung und dem erzeugten Code.

Der Aufruf

`compile` ⟨*Eingabe*⟩ ⟨*Ausgabe*⟩ ⟨*Auswertungsstrategie*⟩

veranlaßt das Einlesen der Datei ⟨*Eingabe*⟩, stößt den Übersetzungsvorgang an und schreibt das Ergebnis der Übersetzung in die Datei ⟨*Ausgabe*⟩. Die Bedeutung des Parameters ⟨*Auswertungsstrategie*⟩ ist dieselbe wie beim Aufruf von `testcompile`. Die Datei, die durch diesen Aufruf erzeugt wird, kann direkt als Eingabe für den Maschinensimulator verwendet werden.

6.1.2 Implementierung des Maschinensimulators

Der Maschinensimulator wurde in C unter Zuhilfenahme der Utilities LEX und YACC implementiert. LEX erzeugt aus regulären Ausdrücken einen Scanner und mit YACC kann aus einer kontextfreien Grammatik ein Parser kreiert werden. Die Syntax ist dabei Ratatosk sehr ähnlich (siehe Beispiel 6.2). Nähere Informationen zu LEX und YACC finden sich beispielsweise in [Her92].

Der prinzipielle Aufbau des Programms `nomach`, das eine NoMach simuliert, ist im folgenden Pseudoalgorithmus (Abbildung 6.6) wiedergegeben.

(1) Einlesen und Linearisieren des auszuführenden Maschinenprogramms
(2) **repeat**
(3) $b \leftarrow$ Benutzereingabe
(4) Werte b aus und führe gewünschte Aktion aus
(5) **until** $b =$ Beenden

Abb. 6.6: Vorgehensweise des Maschinensimulators.

Nach dem vorbereitenden Schritt 1 wird dem Benutzer ein Befehlsprompt gegeben, an dem er verschiedene Kommandos eingeben kann, die den Ablauf der NoMach steuern. So kann beispielsweise ein einzelner Maschinenbefehl ausgeführt werden, oder es werden die Komponenten der NoMach angezeigt.

Betrachten wir nun die einzelnen Schritte ein wenig genauer. Um das Einlesen von Maschinenprogrammen möglichst einfach zu gestalten, haben wir eine kontextfreie Grammatik für Maschinenprogramme entworfen, aus der wir mit Hilfe von LEX und YACC ein Scanner und Parser erzeugt.

Der Parser liefert der Reihe nach die einzelnen Maschinenbefehle und die Marken, die in dem einzulesenden Maschinenprogramm P enthalten sind. Sollte dabei ein Fehler auftreten, wird dieser vom Parser gemeldet und die Programmausführung wird abgebrochen.

Da das durch den Übersetzer erzeugte Maschinenprogramm P_M aber baumstrukturierte Adressen enthält, müssen diese zuerst in lineare Adressen umgesetzt werden. Um dies zu erreichen, werden zuerst die einzelnen Maschinenbefehle in dem Feld `pstore` der Reihe nach abgelegt. Der Index in `pstore` entspricht dann auch der linearen Adresse, die dem eingelesenen Maschinenbefehl zugeordnet ist. Die Marken werden in dem Feld `adrlist` abgelegt, wobei mit jeder Marke auch die lineare Adresse gespeichert wird, die mit dieser Marke verbunden ist.

Beispiel 6.3

Das zu lesende Maschinenprogramm P_M sei

1.2:	jump_on_true 1.1
	push_const (Y_{bool} : `TRUE`)
	jump 1.2
1.1:	ret

Im Feld `pstore` werden nun die einzelnen Maschinenbefehle hintereinander abgelegt. Der Maschinenbefehl jump_on_true 1.1 wird in `pstore[1]` gespeichert, push_const (Y_{bool} : `TRUE`) in `pstore[2]`, usw. Die Marke 1.2 wird zusammen mit der linearen Adresse 1, die Marke 1.1 zusammen mit der linearen Adresse 4 in `adrlist` gespeichert. □

Wird versucht, dieselbe Marke zweimal zu verwenden, so erscheint eine Meldung, die den Benutzer auf diesen Umstand aufmerksam macht, da in diesem Fall ein Fehler in der Eingabe naheliegend ist.

Im Anschluß an das Einlesen erfolgt dann das eigentliche Linearisieren. Dazu wird jede baumstrukturierte Adresse a, die in einem Befehl verwendet wird, durch die lineare Adresse ersetzt, die zusammen mit a in `adrlist` gespeichert ist.

Beispiel 6.4 (Fortsetzung von Beispiel 6.3)

Der Maschinenbefehl jump_on_true 1.1 wird also zu jump_on_true 4, und der Maschinenbefehl jump 1.2 wird zu jump 1. □

Sollte beim Linearisieren der Fall eintreten, daß eine baumstrukturierte Adresse a in einem Befehl auftritt, nicht jedoch als Marke definiert wurde, so hält das Programm mit einer Fehlermeldung an.

Wurde die Linearisierung erfolgreich durchgeführt, so wird das gesamte eingelesene Maschinenprogramm in der linearisierten Form von der Prozedur `prg_anzeige` ausgegeben.

Nach diesem vorbereitenden Schritt werden die einzelnen Komponenten der abstrakten Maschine durch die Prozedur `init` initialisiert. An dieser Stelle wollen wir auch die Implementierung der einzelnen Komponenten etwas näher betrachten.

Die beiden Keller, der Berechnungsstack und das Maschinenprogramm sind als Felder realisiert. Auf das oberste Keller– bzw. Berechnungsstack–Element zeigt jeweils ein Stackpointer. Die beiden Keller sind Felder über ganzen Zahlen. Beim Berechnungsstack findet als Datenstruktur ein sog. Variantenrecord Verwendung. In Abhängigkeit der Recordkomponente `feldtyp` enthält die Komponente `block` verschiedene Variablen. Für `feldtyp` stehen dabei die folgenden Werte zur Verfügung: `YBlock_Num`, `YBlock_Char`, `YBlock_Cons`, `YBlock_Bool` für die verschiedenen Arten von Wertelementen, `FBlock` für ein Funktionselement und `PBlock` für einen Verweisblock.

In Abbildung 6.7 ist die Definition dieses Variantenrecords angegeben.

```
typedef struct {
  enum FeldTyp     feldtyp;
  union
  { float         num_wert;
    char          char_wert;
    yblock_cons   cons_wert;
    int           bool_wert;
    fblocktyp     f_wert;
    p_heap_elem   ptr;
  } block;
} c_elem;
```

Abb. 6.7: Definition des Typs eines Berechnungsstack–Elements.

Die Datentypen `yblock_cons` und `fblocktyp` bezeichnen wiederum Records, in denen die Komponenten von Konstruktor–Wertelementen (Konstruktorname und Konstruktornummer) bzw. Funktionselementen (Adresse und Verweis) angegeben sind. Der Typ `p_heap_elem` bezeichnet einen Zeiger auf ein Haldenelement.

Haldenelemente sind ebenfalls als Variantenrecords implementiert. Die Nachfolgerliste und der Keller aus Verweisen im Falle von Knotenelement bzw. Pfadverweiselement ist dabei ebenfalls als Feld fester Länge realisiert.

Ein einzelner Maschinenbefehl des Maschinenprogramms wird in einem Record abgelegt. Die verschiedenen Komponenten dieses Records sind nötig, da es verschiedene Arten von Parametern gibt, die in Recordkomponenten unterschiedlichen Typs gespeichert werden.

Dem Leser wird auffallen, daß alle Teile der Maschine, die eine unterschiedliche Größe haben können, wie z.B. Keller oder Maschinenprogramm, als Felder realisiert sind. Diese Designentscheidung begründen wir damit, daß die gesamte Implementierung als Experimentiersystem gedacht ist. Außerdem steht von konzeptioneller Seite her einer Verwendung von dynamischen Datenstrukturen nichts im Wege.

Die einzelnen Komponenten sind als globale Variablen implementiert.

Nach der Initialisierung tritt das Programm in die **repeat** — **until** Schleife (Zeile 2 bis 5 in Abbildung 6.6) ein. Dem Benutzer wird ein Befehlspromt gezeigt, worauf dieser einen Befehl eingeben kann. Dabei stehen die folgenden Befehle zur Verfügung:

`q` Der Maschinensimulator wird beendet (*quit*).

`h` Es wird eine Liste aller möglichen Befehle angezeigt (*help*).

`s` Mittels der Funktion `execute` wird der aktuelle Maschinenbefehl ausgeführt (*step*). Falls kein Maschinenbefehl mehr ausgeführt werden kann, erscheint eine Warnung.

`s` xx Dieser Befehl ist eine Kurzform für die xx–malige Verwendung des Befehls `s`.

`g` xx Es wird so oft die Ausführung eines Befehls mittels `execute` angestoßen, bis der Programmzähler auf die Adresse xx zeigt, oder aber die Abarbeitung beendet ist (*go*).

`r` Das Maschinenprogramm wird vollständig abgearbeitet (*run*).

`v` Alle Komponenten der Maschinen werden mit Hilfe der Prozedur `state` angezeigt (*view*).

`i` xx Das xx–te Element des Berechnungsstacks wird mit Hilfe von `inspect` angezeigt. Falls es sich dabei um ein Verweiselement handeln sollte, so verwendet `inspect` die Prozedur `zeige_rekursiv`, um die dahinterliegende Struktur vollständig anzeigen zu lassen (*inspect*).

`c` Die abstrakte Maschine wird zurückgesetzt (*clear*).

`p` xx Die Maschinenbefehle an den Adressen xx bis $xx + 10$ werden, soweit existent, angezeigt (*program view*).

Die eben erwähnte Prozedur `execute` stellt das Herzstück des Maschinensimulators dar. Sie realisiert die Befehlssemantik der abstrakten Maschine. Dazu ist in einer großen Fallunterscheidung für jeden Maschinenbefehl dessen Befehlssemantik implementiert. Anhand des Maschinenbefehl `ret` wollen wir dies demonstrieren: In Abbildung 6.8 haben wir den C–Code für den `ret`–Befehl angegeben. Daneben haben wir die in Abschnitt 4.2 definierte Befehlssemantik desselben Befehls wiedergegeben.

Die Variable `jsp` gibt an, wieviele Elemente des Sprungkellers belegt sind. Mit der Bedingung `if (jsp >= 1)` wird also getestet, ob sich auf dem Sprungkeller mindestens ein Element befindet. Wenn ja, dann wird der Programmzähler (`pc`) auf den Wert des obersten Sprungkeller–Elements gesetzt und das oberste Sprungkeller–Element durch Verkleinern

```
case RET :
   if (jsp >= 1) { pc = jpd[jsp];
                   jsp--; }
   else          { pc = 0;
                   rc = -3; }
   break;
```

Abb. 6.8: Implementierung von ret.

$$
\begin{array}{l}
\mathcal{C}[\![\,\mathsf{ret}\,]\!](pc, afp, rc, cst, j.1 : \ldots : j.k, envpd, hp) = \\
\quad \textbf{if } k \geq 1 \\
\qquad (\underline{j.1}, afp, rc, cst, \underline{j.2 : \ldots : j.k}, envpd, hp) \\
\quad \textbf{otherwise} \\
\qquad (\underline{0}, afp, \underline{-3}, cst, j.1 : \ldots : j.k, envpd, hp)
\end{array}
$$

Abb. 6.9: Befehlssematik von ret.

von `jsp` entfernt. Falls die Bedingung nicht erfüllt war, so wird der Programmzähler auf 0 gesetzt und in das Rückgabewert–Register (`rc`) der Fehlercode -3 geschrieben.

Nach Abarbeitung dieser Routine hat sich die Maschinenkonfiguration in derselben Weise verändert, wie in der formalen Definition (Abbildung 6.9) beschrieben.

Die Implementierung des Maschinensimulators umfaßt zwei Dateien: In `nomach.l` sind die regulären Ausdrücke für die möglichen Maschinenbefehle angegeben. Aus dieser Datei wird mit Hilfe von LEX ein Scanner erzeugt. Die Datei `nomach.y` enthält die Beschreibung der kontextfreien Grammatik für Maschinenprogramme sowie alle beschriebenen Prozeduren. Mit Hilfe von YACC und einem C–Compiler wird daraus der lauffähige Maschinensimulator erzeugt.

6.2 Installation

Der Übersetzer sowie der Maschinensimulator können auf jedem Rechner installiert werden, der unter dem Betriebssystem Unix[2] betrieben wird. Als Software müssen auf dem Rechner ein C–Compiler sowie die Utilities LEX und YACC verfügbar sein.

Vorbereitende Schritte

Die im folgenden vorgeschlagenen Verzeichnisse sind keineswegs zwingend. Der erfahrene Unix–Benutzer kann selbstverständlich andere Verzeichnisse verwenden.

- Kopieren der Dateien der beigefügten Disketten auf den Rechner:

  Dateien der Diskette 1 in das Verzeichnis `/usr/local/Gofer`
  Dateien der Diskette 2 in das Verzeichnis `/usr/local/NoMach`
- Wechseln in das Verzeichnis `/usr/local/Gofer`
- Dekomprimieren der Datei `gofer.tgz` mit `gunzip gofer.tgz`
- Entpacken der Datei `gofer230a.tar` mit `tar -xf gofer230a.tar`
- Entpacken der Datei `ratatosk.tar` mit `tar -xf ratatosk.tar`

[2] Unix ist ein eingetragenes Warenzeichen der AT&T Bell Laboratories.

Erzeugen des Gofer–Systems

Um das Gofer–System zu erzeugen, muß zuerst in `/usr/local/Gofer/src/prelude.h` eine Änderung vorgenommen werden: Am Anfang dieser Datei steht eine Liste von `define`–Angaben für unterschiedliche Unix–Betriebssysteme. Für das gewünschte Zielsystem muß die entsprechende `define`–Angabe auf 1 gesetzt werden. Alle anderen `define`–Angaben müssen auf 0 gesetzt sein.

Durch den Aufruf von `make` im Verzeichnis `/usr/local/Gofer/src` wird dann das Gofer–System erzeugt.

Durch Aufruf des Skripts `/usr/local/Gofer/scripts/gofer` wird das Gofer–System gestartet. Dieses Skript sollte also im Suchpfad des Systems zu finden sein.

Sollten bei der Erzeugung des Gofer–Systems Probleme auftreten, so ist in der Datei `/usr/local/Gofer/src/Readme` die Erzeugung des Gofer–Systems nochmals genau beschrieben.

Erzeugen des Maschinensimulators

Die Erzeugung des Maschinensimulators wird durch den Aufruf von `make` im Verzeichnis `/usr/local/NoMach` angestoßen. Der Maschinensimulator selbst wird dann mit der Eingabe `nomach` gestartet. Deshalb sollte auch diese Datei im Suchpfad des Systems zu finden sein.

6.3 Anwendungsbeispiel

In diesem Abschnitt wollen wir die Bedienung des Übersetzers und des Maschinensimulators anhand zweier kleiner Beispiele demonstrieren. Dabei zeigen wir immer wieder die Ausgaben, die die jeweiligen Programme erzeugen.

Als erstes Anwendungsbeispiel wollen wir das in Abbildung 6.10 dargestellte Programm zur Berechnung der Fakultätsfunktion übersetzen und ausführen lassen.

```
GOAL fak(3)
PROGRAM Fakultaet;
  FUNC_SPEC
    CALL_BY_NAME fak;
    fak : CARDINAL -> CARDINAL.
    fak (n) = 1, IF n = 1,
            = n * fak(n - 1), OTHERWISE;
  END_FUNC_SPEC
END_PROGRAM Fakultaet.
```

Abb. 6.10: NoName*–Programm zur Berechnung der Fakultät von 3.

Die Datei, in der das Fakultätsprogramm gespeichert ist, sei `fak.nnm`. Der Übersetzer wird durch den Aufruf

```
gofer + nnm.gp -h200000
```

im Verzeichnis `/usr/local/NoMach` aktiviert.

Nach der Einladephase meldet sich das System mit der Ausgabe

```
Gofer session for: (project: nnm.gp)
/usr/local/Gofer/standard.prelude
nnm.def
parseerr.gs
nnm.f
nnm.sca
nnm.par
nnm.tr
nnm.ub
Type :? for help
?
```

Durch den Aufruf von

```
testcompile "fak.nnm" CallByValue
```

wird der Übersetzungsvorgang angestoßen. Die Angabe der Standardauswertungsstrategie (hier `CallByValue`) ist hier zwar ohne Bedeutung, da für die einzige Funktion in `fak.nnm` ohnehin eine eigene Auswertungsstrategie angegeben ist, kann aber dennoch nicht weggelassen werden.

Nacheinander werden das eingelesene Programm `fak.nnm`, der zugehörige abstrakte Syntaxbaum, die errechnete Umgebung sowie das erzeugte Maschinenprogramm ausgegeben.

Mit dem Aufruf

```
compile "fak.nnm" "fak.nomach" CallByValue
```

wird dann die Erzeugung einer Datei `fak.nomach` veranlaßt, in der sich das erzeugte Maschinenprogramm befindet.

Mit `:q` können wir das Gofer–System verlassen.

Versuchen wir nun, das erzeugte Maschinenprogramm auszuführen. Dazu starten wir zuerst den Maschinensimulator mit

`nomach fak.nomach`,

worauf dieser das eingelesene Maschinenprogramm mit linearisierten Adressen anzeigt und mit der Meldung

```
Zustand: AFP = 1, RC = 0, PC = 1 [PUSH_CONST Y_NUM 0.0000]
Befehl (h = Hilfe):
```

eine Benutzereingabe erwartet. Der in eckige Klammern eingefaßte Maschinenbefehl hinter der Angabe des Programmzählers gibt den nächsten auszuführenden Befehl an.

Mit der Eingabe `r` veranlassen wir die komplette Abarbeitung des eingelesenen Maschinenprogramms und erhalten die Meldung

```
Maschinenlauf beendet.
Letzter ausgefuehrter Befehl war : 25 [STOP]
Meldung:
Fehlerfreie Ausfuehrung.

Erg = 6.000000

Zustand: AFP = 1, RC = 0, PC = 0 [*** Undefiniert ***]
Befehl (h = Hilfe):
```

Die Ausführung erfolgte also ohne Fehler und das Ergebnis der Berechnung ist `6.000000`. Wir wollen die Berechnung nun erneut durchführen lassen, wobei diese aber schrittweise erfolgen soll. Dazu müssen wir zuerst mit `c` die abstrakte Maschine zurücksetzen. Mit `g 15` veranlassen wir, daß die Maschine solange Befehle abarbeitet, bis der Programmzähler auf 15 steht oder aber die Programmabarbeitung beendet ist. In unserem Fall meldet sich der Simulator mit der Meldung

```
Zustand: AFP = 2, RC = 0, PC = 15 [PUSH CSTACK [AFP]]
Befehl (h = Hilfe):
```

was bedeutet, daß die Abarbeitung noch nicht beendet ist. Mit `v` können wir uns nun die aktuelle Konfiguration der NoMach ansehen. Auf dem Berechnungsstack liegen mittlerweile 3 Wertelemente und auf dem Sprungkeller liegt eine Rücksprungadresse. Mit `p 15` können wir uns die Befehle an den Adresse 15 bis 25 ansehen, um festzustellen, welche Befehle als nächstes ausgeführt werden. Mit `g 22` führen wir nun die Befehle zum Bilden eines neuen Kellerrahmens aus. Auf dem Berechnungsstack liegen dann (Ausschnitt der Ausgabe des Befehls `v`)

```
6: F: 19, 2           }
5: Y_Num: 2.000000    } aktueller Kellerrahmen
4: Y_Num: 1.000000    }
3: Y_Num: 1.000000
2: Y_Num: 1.000000
1: Y_Num: 1.000000.
```

Mit `s` wird der aktuell zur Bearbeitung anstehende Maschinenbefehl ausgeführt, in diesem Fall der Aufruf des Unterprogramms an der Adresse 26. Mit den Befehlen `s` und `v` führen wir den Maschinenbefehl eval 1 aus und betrachten die veränderte Maschinenkonfiguration. Da der 1. Parameter im aktuellen Funktionsrahmen ein Funktionselement ($F:\ 19,\ 2$) war, wurde die Adresse 19 angesprungen und der Verweis 2 auf den Umgebungskeller gelegt. Mit dem Befehl `q` wollen wir nun den Maschinensimulator verlassen.

An dieser Stelle wollen wir dieses Beispiel beenden und statt dessen noch kurz die Funktionsweise des Befehls i demonstrieren.

Dazu übersetzen wir zuerst in der oben beschriebenen Weise das Programm `baum.nnm`, das in Abbildung 6.11 wiedergegeben ist, und erhalten die Datei `baum.nomach`. Dann starten wir mit `nomach baum.nomach` den Maschinensimulator. Mit `r` lassen wir das Maschinenprogramm abarbeiten. Betrachten wir danach mit `v` die Maschinenkonfiguration,

```
GOAL Knoten(Blatt(3),Blatt(7),TRUE)
PROGRAM BaumDemonstration;
  ALG_DEF
    Baum = Knoten(Baum,Baum,BOOLEAN)
         | Blatt(INTEGER);
  END_ALG_DEF
END_PROGRAM BaumDemonstration.
```

Abb. 6.11: NoName*–Programm `baum.nnm`.

so erkennen wir auf dem Berechnungsstack ein Verweiselement. Um nun auch das dahinterstehende Haldenelement betrachten zu können, verwenden wir den Befehl `i 2`. Dies führt zur Ausgabe

```
CSTACK[2]: P: 525232 (Verweis auf Knoten-Element [3])
Konstruktor: Y_Cons: Knoten (1)
- P: 519072 (Verweis auf Knoten-Element [1])
  Konstruktor: Y_Cons: Blatt (2)
  - Y_Num: 3.000000
- P: 522152 (Verweis auf Knoten-Element [1])
  Konstruktor: Y_Cons: Blatt (2)
  - Y_Num: 7.000000
- Y_Bool: TRUE
```

wobei selbstverständlich die angegebenen Pointerwerte von Ausführung zu Ausführung variieren können. Mit `q` verlassen wir den Maschinensimulator.

Die Beschreibung der Implementierung ist hiermit abgeschlossen. Beim Arbeiten mit Übersetzer und Maschinensimulator sollte man sich jedoch vor Augen halten, daß diese Programme mehr als Experimentiersystem denn als benutzerfreundliche Anwendungsprogramme gedacht sind. Eine ansprechendere Implementierung wäre sicherlich wünschenswert, war jedoch kein Ziel dieser Diplomarbeit.

Kapitel 7

Ausblick

Mit der Vorstellung der Rechner–Implementierung in Kapitel 6 ist die eigentliche Bearbeitung des Diplomarbeitsthemas abgeschlossen. In diesem Kapitel wollen wir nun noch einige Erweiterungsmöglichkeiten und weitergehende Gedanken vorstellen.

7.1 Optimierung

Die Optimierung, die wir hier besprechen wollen, betrifft eine Verkürzung des erzeugten Maschinenprogramms durch Entfernen unnötiger Maschinenbefehle. Dadurch ermöglichen wir eine schnellere Ausführung der Maschinenprogramme.

Für diese Optimierung gibt es drei Ansatzpunkte, nämlich

- vor dem eigentlichen Übersetzungsvorgang, also noch auf Ebene des zu übersetzenden NoName*–Programms,
- während der eigentlichen Übersetzung, also durch Modifikationen der Übersetzungsfunktionen und
- nach der Übersetzung, also auf Ebene des erzeugten Maschinenprogramms.

Wir wollen im folgenden diese Punkte anhand einiger Beispiele verdeutlichen.

7.1.1 Optimierung vor der Übersetzung

Die Optimierung vor der Übersetzung wird auf Ebene des zu übersetzenden Programms P durchgeführt. Ein Ansatzpunkt wäre hier das Entfernen von offensichtlich unerfüllbaren Bedingungen in Funktionen, die via IF definiert sind. Lassen wir beispielsweise in der Definition der Funktion `fkt` aus Abbildung 7.1 die Zeile

```
= 2, IF (n<1) and (n>1)
```

weg, so erhalten wir eine neue Funktion, die für alle möglichen Eingaben genau dieselben Funktionswerte wie `fkt` berechnet.

```
FUNC_SPEC
   fkt :  INTEGER -> INTEGER.
   fkt(n) = 1, IF n = 1,
          = 2, IF (n<1) and (n>1),
          = 0, OTHERWISE;
END_FUNC_SPEC
```

Abb. 7.1: Funktion mit unerfüllbarer Bedingung.

Ein weiterer Ansatzpunkt zur Optimierung ist die sogenannte Konstantenfaltung (vgl. [Wal74]). Dazu wird in P' nach konstanten numerischen Ausdrücken gesucht. Die Maschinenbefehle, die als Übersetzungsergebnis dieses Ausdrucks erzeugt wurden, liefern bei der Ausführung natürlich immer dasselbe Ergebnis. Folglich können diese Berechnungen bereits vor der Übersetzung durchgeführt werden. Die Befehlsfolge zur Berechnung des Ausdrucks kann durch einen Maschinenbefehl ersetzt werden, der das konstante Ergebnis der Berechnung auf den Berechnungsstack schreibt.

Dieser Punkt kann sogar noch ein wenig ausgeweitet werden, wenn bei der Suche nach konstanten numerischen Ausdrücken auch noch algebraische Umformungen, wie beispielsweise das Distributivgesetz, Anwendung finden. So sind die Ausdrücke `3*(x+2)+6` und `(3*x)+(3*2)+6` bezüglich ihres Berechnungsergebnisses äquivalent. Der zweite Ausdruck läßt sich aber durch Bestimmung konstanter Ausdrücke zu `(3*x)+12` vereinfachen.

Daß sich auf diese Weise kürzere Maschinenprogramme ergeben können, ist offensichtlich. So ergibt beispielsweise die Übersetzung von `3*(x+2)+6` sieben Maschinenbefehle während die Übersetzung von `(3*x)+12` nur 5 Maschinenbefehle erzeugt.

7.1.2 Optimierung der Übersetzungsfunktionen

Im Maschinenprogramm, das von den Übersetzungsfunktionen erzeugt wird, finden sich des öfteren Sprungtabellen wieder. So wird beispielsweise die Auswahl der anzuwendenden Alternative einer Funktion, die mittels Pattern Matching definiert ist, durch eine Sprungtabelle realisiert (siehe Abschnitt 5.5).

In Abbildung 7.2 ist das Schema solch einer Sprungtabelle wiedergegeben. Dieses Schema kann jedoch verbessert werden, indem man die letzte Sprunganweisung jump $adr.n$ in der Sprungtabelle wegläßt und statt deren direkt die Maschinenbefehle, die zu diesem Fall gehören, einsetzt. In Abbildung 7.3 findet sich das so verbesserte Schema wieder.

	jump_idx adr
adr :	jump $adr.1$
	$\vdots$
	jump $adr.(n-1)$
	jump $adr.n$
$adr.1$:	Fall 1
	$\vdots$
$adr.(n-1)$:	Fall $(n-1)$
$adr.n$:	Fall n

Abb. 7.2: Ursprüngliche Sprungtabelle.

	jump_idx adr
adr :	jump $adr.1$
	$\vdots$
	jump $adr.(n-1)$
	Fall n
$adr.1$:	Fall 1
	$\vdots$
$adr.(n-1)$:	Fall $(n-1)$

Abb. 7.3: Verbesserte Sprungtabelle.

Ein weiterer Ansatzpunkt zur Verbesserung der vorgestellten Übersetzungsfunktionen liegt in der Tatsache, daß bei der Übersetzung Spezialfälle nicht beachtet werden. Vielmehr wird auf strukturell gleiche Konstrukte dieselbe Übersetzungsfunktion angewandt, ohne zu beachten, daß sich in Einzelfällen bessere Maschinenbefehlssequenzen finden ließen.

Betrachten wir dazu das Programmfragment in Abbildung 7.4

```
ALG_DEF
  Tupel = Paar(REAL,REAL);
END_ALG_DEF

FUNC_SPEC
  Proj : Tupel -> REAL.
  Proj(Paar(x,y)) = x;
END_FUNC_SPEC
```

Abb. 7.4: Pattern Matching mit einem Konstruktor.

Bei der Übersetzung der Funktion `Proj` wird durch die Übersetzungsfunktionen, wie sie in Abschnitt 5.5 beschrieben wurden, eine Sprungtabelle mit einem Eintrag erzeugt. Eine solche Tabelle ist freilich unnötig, da immer dieser eine Eintrag angesprungen wird. Durch die Verwendung von spezielleren Übersetzungsfunktionen könnte hierbei eine Optimierung erzielt werden.

Der Befehlssatz der vorgestellten abstrakten NoName–Maschine ist so konzipiert, daß er alle Befehle enthält, die im Zuge der vorgestellten Implementierung notwendig sind. Erweitert man den Befehlssatz aber ein wenig, so sind weitere Optimierungen denkbar. Ein in vielen Maschinen vorhandener Maschinenbefehl ist beispielsweise inc c, der bewirkt, daß der Wert des obersten Elements des Berechnungsstacks um c erhöht wird.

Mit einem so erweiterten Befehlssatz lassen sich für bestimmte Ausdrücke effizientere Maschinenbefehlssequenzen finden. Der Ausdruck $(a+c)$, wobei a ein beliebiger Ausdruck und c eine numerische Konstante ist, läßt sich dann mit dem Schema

$trans_{Expr}[a](\ldots)$
inc c

anstelle des Übersetzungsschemas

$trans_{Expr}[a](\ldots)$
$trans_{Expr}[c](\ldots)$
add

übersetzen. Ein ähnliches Vorgehen wäre beispielsweise für die Multiplikation mit einer Konstanten und dem neuen Maschinenbefehl rep c denkbar.

Die Möglichkeit der Optimierung durch einen erweiterten Befehlssatz sollte vor allem auch in Hinblick auf Abschnitt 7.2 betrachtet werden. Dort werden wir anregen, als Zielarchitektur nicht die abstrakte NoName–Maschine, sondern eine realen Maschine zu verwenden. Und die meisten realen Maschinen besitzen einen umfangreichen Befehlssatz, der solche effizienteren Befehle enthält.

7.1.3 Optimierung des erzeugten Maschinenprogramms

Ein dritter möglicher Ansatzpunkt für die Optimierung findet sich in den Maschinenprogrammen, die durch die Übersetzungsfunktionen erzeugt wurden. Diese Art von Optimierung wird meist mit Hilfe eines sogenannten *Peephole–Optimierers* [WM92] durchgeführt. Dazu wird eine Folge von einigen wenigen Maschinenbefehlen betrachtet und versucht, dafür eine bessere, d.h. kürzere, Sequenz von Maschinenbefehlen zu finden.

In den erzeugten Maschinenprogrammen gibt es vor allem einen Punkt, für den sich eine solche Vorgehensweise anbieten würde: Die Eliminierung von leeren Sprunganweisungen. Eine Sprunganweisung mit Programmadresse adr heißt leer, wenn die durch diesen Befehl angesprungene Programmadresse $(adr + 1)$ ist. In Abbildung 7.5 ist ein Maschinenprogramm dargestellt, das an der Adresse 3 eine leere Sprunganweisung hat.

$$\begin{array}{rl} 1: & \mathsf{jump\_on\_true}\ 4 \\ 2: & \mathsf{push\_const}\ Y_{num} : 2 \\ 3: & \mathsf{jump}\ 4 \\ 4: & \mathsf{cleanup} \end{array}$$

Abb. 7.5: Maschinenprogramm mit leerer Sprunganweisung.

Diese leeren Sprungbefehle können natürlich entfernt werden. Hierbei ist natürlich zu beachten, daß die nachfolgenden Befehle eine um jeweils 1 verminderte Programmadresse haben. Es müssen also alle Sprung– und Unterprogrammbefehle, die sich auf Adressen größer adr beziehen, entsprechend geändert werden. Bezogen auf das Maschinenprogramm in Abbildung 7.5 würde sich das folgende kürzere Maschinenprogramm ergeben.

$$\begin{array}{rl} 1: & \mathsf{jump\_on\_true}\ 3 \\ 2: & \mathsf{push\_const}\ Y_{num} : 2 \\ 3: & \mathsf{cleanup} \end{array}$$

Abb. 7.6: Verkürztes Maschinenprogramm.

Wird diese Art der Optimierung auf Ebene von Maschinenprogrammen mit baumstrukturierten Adressen durchgeführt, so genügt es sogar, nur die leeren Sprunganweisungen zu entfernen. Die eben beschriebene Änderung von Sprungadressen ist unnötig. Sie ergibt sich beim Linearisieren von selbst.

Erweitert man den Befehlssatz der NoMach, wie bereits in Abschnitt 7.1.2 angesprochen, so lassen sich noch weitere Optimierungen durchführen. So läßt sich beispielsweise die Befehlsfolge $\mathsf{push\_const}\ (Y_{num} : c)\ \ \mathsf{add}$ zu $\mathsf{inc}\ c$ vereinfachen. Der Platzhalter c steht dabei für eine beliebige Zahl.

Mit diesem Punkt wollen wir hier die Überlegungen zur Optimierung des erzeugten Maschinenprogramms beenden. Weiterführende Literatur zum Thema Optimierung sowie eine umfangreiche Literaturliste findet sich in [ASU86].

7.2 Codeerzeugung für reale Maschinen

Die Maschinenprogramme, die wir in dieser Arbeit vorgestellt haben, sind für eine abstrakte NoName–Maschine bestimmt. Da eine solche Maschine, wie bereits der Name andeutet, real nicht existiert, benötigen wir einen Interpreter, um diese Maschinenprogramme ausführen zu können. Dies ist freilich unbefriedigend und zwar nicht zuletzt deswegen, da der Umweg über einen Interpreter einen Zeitverlust bei der Ausführung des Maschinenprogramms mit sich bringt. Es wäre also wünschenswert, Maschinenprogramme für real existierende Maschinen zu erzeugen.

Allerdings unterscheiden sich reale Maschinen in einigen Punkten von der vorgestellten abstrakten NoName–Maschine, so daß Änderungen in den zugrundeliegenden Überlegungen gemacht werden müssen. Im folgenden betrachten wir die wesentlichen Charakteristika der vorgestellten abstrakten NoName–Maschine und des Befehlssatzes und skizzieren, ob und in wieweit jeweils Änderungen nötig sind.

- Die Kontrollstrukturen, die in übersetzten NoName*–Programmen Verwendung finden, nämlich Sprünge und Unterprogramme, finden sich auch in realen Maschinen wieder. Hier sind also keine Änderungen vorzunehmen.
- Halde, Stack und Keller sind auf realen Maschinen einfach zu realisieren. Oftmals existieren Maschinenbefehle (z.B. `push` und `pop`), die solche Strukturen direkt unterstützen. Für die Halde muß eine Haldenverwaltung implementiert werden, die sich um die Freigabe von nicht mehr benötigten Haldenobjekten kümmert (*grabage collection*, siehe [FH88]).
- Die meisten realen Maschinen führen Berechnungen nicht auf einem Runtime–Stack, sondern mittels eines Registersatzes durch. In [WM92] oder [ASU86] finden sich Beschreibungen, wie Ausdrücke übersetzt werden können, damit sie auf Registermaschinen arbeiten.
- Die NoMach kennt verschiedene Arten von Berechnungsstack–Elementen. In Abschnitt 4.1 haben wir aber erwähnt, daß sich für diese Elemente auch eine numerische Repräsentation finden läßt. Diese kann auf realen Maschinen problemlos verwendet werden.
- Viele Maschinenbefehle der NoMach sind genau auf die beabsichtigten Operationen zugeschnitten. Es lassen sich also keine passenden Gegenstücke im Befehlssatz realer Maschinen finden. Für diese komplexen NoName–Maschinenbefehle lassen sich aber Befehlssequenzen für die reale Maschinen finden, die dieselbe Wirkung haben. Diese Befehlssequenzen können dann in Form von Unterprogrammen anstelle der ursprünglichen NoMach–Befehle aufgerufen werden.

Bei diesem Ansatz haben wir dann die Sprache *LNoMach* als Zwischensprache im Übersetzungsvorgang verwendet. In vielen Texten zum Übersetzerbau wird der Schritt über eine Zwischenrepräsentation gemacht (siehe z.B. [Wir86, ASU86]). Ein weiterer, möglicherweise sogar effizienterer Ansatz ist, nicht den Umweg über eine Zwischenrepräsentation zu wählen, sondern direkt ein Maschinenprogramm für eine reale Maschine zu erzeugen. In [Glo93] wurde ein solcher Ansatz verfolgt.

7.3 Erweiterung des Sprachumfangs

Ein Ausdrucksmittel, das in der Sprache NoName keine direkte Unterstützung auf syntaktischer Ebene besitzt, jedoch sehr angenehm bei der Programmerstellung ist (siehe

[May95]), sind Listen. Mittels algebraischer Datentypen können zwar Listen erzeugt werden, jedoch ist der dann nötige Umgang mit Konstruktorausdrücken sehr aufwendig. Insbesondere bei Zeichenketten (*strings*), also Listen von Zeichen, ist die Konstruktordarstellung hinderlich. Eine kompaktere Darstellung wäre also wünschenswert.

Für die Übersetzung ergibt sich durch die Verwendung von eigenen Sprachkonstrukten, die die Erstellung und Manipulation von Listen ermöglichen, jedoch nichts neues, da sich diese kompaktere Darstellung automatisch in die bisher verwendete Konstruktordarstellung überführen läßt.

Ein weiterer Ansatzpunkt zur Erweiterung des Sprachumfangs liegt in den zur Verfügung stehenden Auswertungsstrategien. Die unterstützte Auswertungsstrategie Call–by–Name hat den bekannten Nachteil der Mehrfachauswertung eines mehrfach benötigten Parameters. Dieses Problem läßt sich leicht beheben, indem nach der Auswertung eines Parameters der berechnete Wert an die betreffende Stelle im Kellerrahmen zurückgeschrieben wird. Dieser Ansatz führt zu der Auswertungsstrategie *Call–by–Need*.

Nun gibt es noch eine Klasse von Funktionen, die mit den bisherigen Auswertungsstrategien nicht berechnet werden können. Diese Funktionen arbeiten mit unendlichen Datentypen. In Abbildung 7.7 ist ein Beispiel für solche Funktionen gegeben (vgl. [Loo90]).

```
ALG_DEF
  NumList = Cons(CARDINAL,NumList)
          | Nil;
END_ALG_DEF
FUNC_SPEC
  fiblist : CARDINAL x CARDINAL -> NumList;
  getfib : CARDINAL x NumList -> CARDINAL.
  fiblist(x,y) = Cons(x,fiblist(y,x+y));
  getfib(0,Cons(x,fl)) = x
  getfib(n+1,Cons(x,fl)) = getfib(n,fl)
  getfib(n,Nil) = 0;
END_FUNC_SPEC
```

Abb. 7.7: Programm mit unendlichen Datentypen.

Der Aufruf `getfib(5,fiblist(1,1))` sollte die 5. Fibonaccizahl, also 8, liefern. Mit den Auswertungsstrategien Call–by–Value und Call–by–Name wird jedoch von der Funktion `getfib` ein Auswerten des 2. Parameters, also des Aufrufs `fiblist(1,1)`, erzwungen. Wie aus der Definition dieser Funktion leicht zu ersehen ist, terminiert diese Berechnung nicht.

Das gewünschte Ergebnis erhält man nur, wenn die von `getfib` angestoßene Berechnung nicht vollständig durchgeführt wird, sondern nach der Bestimmung des ersten Konstruktors suspendiert wird. Es ergibt sich also eine verzögerte Auswertung (*lazy evaluation*).

Um diese Strategie bei der Implementierung umsetzen zu können, ist es notwendig, ein weiteres Berechnungsstack–Element einzuführen, das eine suspendierte Berechnung repräsentiert. Dieses Element muß neben einer Programmadresse, die angibt, an welcher Stelle diese Berechnung wieder aufgenommen werden kann, auch noch die zu dieser Berechnung gehörenden Parameter enthalten.

In wieweit die Umsetzung dieser Strategie auf einer Runtime–Stack Maschine aber noch sinnvoll ist, soll hier nicht näher betrachtet werden.

7.4 „Embedded“ NoName

Der Ansatz der funktionalen Programmierung hat gegenüber dem Ansatz der imperativen Programmierung viele Vorteile. In der Einleitung von [Loo90] und in fast jedem Buch über funktionale Programmierung finden sich dafür Beispiele. Die wesentlichen Vorteile liegen in der deklarativen Beschreibung des zu modellierenden Problems und in der referenziellen Transparenz. Die referenzielle Transparenz ist die Eigenschaft, daß derselbe Funktionsaufruf mit denselben Parameterwerten immer dasselbe Ergebnis liefert. Die Berechnung ist also frei von Seiteneffekten.

Möchte man aber auf Dateien zugreifen oder in sonst einer Art und Weise die Ein– und Ausgabefähigkeiten des Rechners nutzen, so ergeben sich Probleme, da in diesem Fall die referenzielle Transparenz nicht unbedingt gewahrt bleibt. Allerdings sind interaktive Programme ohne Benutzereingabe kaum denkbar. Imperative Sprachen freilich sind prädistiniert für interaktive, zustandsgesteuerte Programme.

Aus diesen Gründen wäre es wünschenswert, den Ansatz der funktionalen und der imperativen Programmierung zu vermischen. Ein möglicher Ansatzpunkt in dieser Richtung sind Monaden [Wad95]. Eine andere Variante wird in [Kla89] angesprochen, nämlich die Einbettung von funktionalen Elementen in imperative Programme: In eigenen Dateien werden in einer funktionalen Programmiersprache algebraische Datentypen und darauf arbeitende Funktionen definiert. Aus diesen Dateien werden ausführbare Maschinenprogramme und Modula–2 Definitionsmodule erzeugt. Von Modula–2 Programmen aus können dann die neu definierten Datentypen und Funktionen benutzt werden.

Ein ähnlicher Ansatz würde sich auch für NoName* und eine imperative „Wirtssprache“, beispielsweise C, verwirklichen lassen. Im Gegensatz zu [Kla89] haben wir jedoch an eine direkte Einbettung von funktionalen Sprachkonstrukten in C–Programme gedacht. Durch spezielle Zeichen (z.B. `#`) werden, wie auch bei *Embedded SQL* (siehe beispielweise [GW94]), die funktionalen Sprachkonstrukte explizit gekennzeichnet. In Abbildung 7.8 findet sich ein Beispiel dafür, wie solch eine Einbettung aussehen könnte.

Ein solches C–Programm P_C mit eingebetteten NoName–Sprachkonstrukten kann dann von einem sogenannten Präprozessor in ein reines C–Programm P'_C umgewandelt werden. Hierbei stellt sich natürlich die Frage, wie die NoName–Funktionen nach C umgesetzt werden können. Ein möglicher Ansatz besteht darin, den hier vorgestellten Übersetzer mitzuverwenden.

Zuerst werden die funktionalen Sprachelemente aus dem gemischtsprachigen Ausgangsprogramm P_C extrahiert. Dann werden die NoName–Funktionen in der in Kapitel 5 beschriebenen Art und Weise übersetzt. Für jede NoName–Funktion wird zudem eine C–Funktion erzeugt, die den Aufruf des Maschinenprogramms und die Übergabe der Parameter übernimmt.

Um die erzeugten Maschinenprogramme abarbeiten zu können, ist zudem noch eine Implementierung der abstrakten NoName–Maschine notwendig, die natürlich ebenfalls Bestandteil des zu erzeugenden Programms P'_C sein muß.

```
void fak_interaktiv() {
   int n=0;
   while (n>=0) {
      printf("Bitte geben Sie eine Zahl ein: \n");
      scanf("%d",&n);
      if (n>=0)
         {printf("Fakultaet von %d ist %d \n", n, fak(n));}
   } /* of while */
/* of fak_interaktiv */
# FUNC_SPEC
#    fak : CARDINAL -> CARDINAL.
#    fak(0) = 1
#    fak(k+1) = (k+1) * fak(k);
# END_FUNC_SPEC
void main() {
    fak_interaktiv;
} /* of main */
```

Abb. 7.8: C–Programm mit eingebetteter NoName–Funktion.

Das erzeugte Maschinenprogramm, die neuen C–Funktionen sowie die Implementierung der abstrakten Maschine werden anstelle der entfernten NoName–Funktionen in das ursprüngliche Programm eingesetzt. In Abbildung 7.9 haben wir versucht, den Vorgang nochmals bildlich darzustellen.

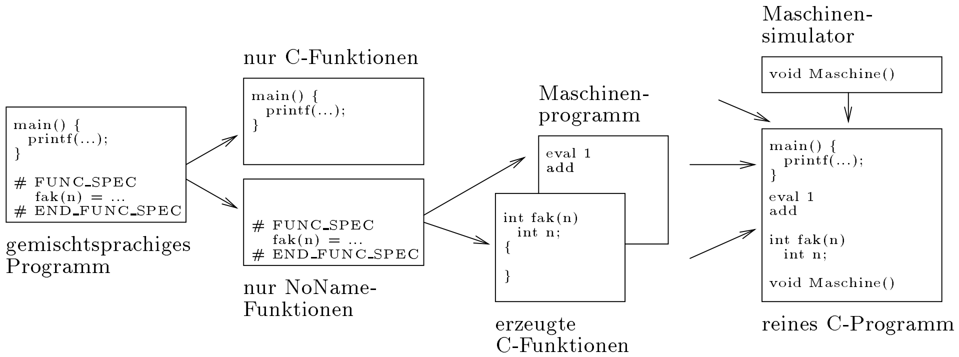


Abb. 7.9: Arbeit des Präprozessors.

Mit dieser skizzenhaften Beschreibung einer möglichen Kombination von funktionalen und imperativen Sprachelementen wollen wir dieses Kapitel beenden.

Anhang A

Kontextfreie Grammatik von NoName*

In diesem Kapitel geben wir die kontextfreie Grammatik von NoName* an. Nichtterminale werden durch spitze Klammern gekennzeichnet. Terminale sind in Hochkommata eingefaßt. Die verwendeten regulären Ausdrücke für Zeichen, Bezeichner, Infixoperatoren und Zahlen sind im folgenden wiedergegeben.

⟨CharToken⟩	=	(["a"–"z"] | ["A"–"Z"] | ["0"–"9"] | "_")
⟨IdToken⟩	=	(["a"–"z"] | ["A"–"Z"]) ⟨CharToken⟩∗
⟨InfixToken⟩	=	"-" | "*" | "/" | ">=" | "<=" | "<>" | "and" | "or" | "div"
⟨NumToken⟩	=	"0" | ((ε |"-") ["1"–"9"] ["0"–"9"]*) |
		((ε |"-") ["0"–"9"] ["0"–"9"]* "." ["0"–"9"] ["0"–"9"]*)

Die Produktionen (1) bis (3) bilden das Grobgerüst eines jeden NoName*–Programms.

(1)	⟨AnfrPrg⟩	→	“GOAL“ ⟨Expr⟩ ⟨Program⟩
(2)	⟨Program⟩	→	“PROGRAM“ ⟨Id⟩ “;“ ⟨SpecList⟩ “END_PROGRAM“ ⟨Id⟩ “.“
(3)	⟨SpecList⟩	→	⟨Spec⟩ ⟨SpecList⟩
		|	⟨Spec⟩

Für eine einzelne Spezifikation gibt es, wie die Produktion (4) zeigt, verschiedene Alternativen: es kann sich dabei um die Spezifikation einer gewöhnlichen Funktion, um die Spezifikation eines tree transducers oder um die Spezifikation eines algebraischen Datentyps handeln.

(4)	⟨Spec⟩	→	⟨FuncSpec⟩
		|	“ALG_DEF“ ⟨AlgDefList⟩ “END_ALG_DEF“
		|	⟨MatSpec⟩

Die Produktionen (5) bis (9) legen fest, wie die Spezifikation eines oder mehrere algebraischer Datentypen aufgebaut ist.

(5)	⟨AlgDefList⟩	→	⟨AlgDef⟩ “`;`“
		|	⟨AlgDef⟩ “`;`“ ⟨AlgDefList⟩
(6)	⟨AlgDef⟩	→	⟨Id⟩ “`=`“ ⟨AlgDefRSList⟩
(7)	⟨AlgDefRSList⟩	→	⟨AlgDefRS⟩
		|	⟨AlgDefRS⟩ “`|`“ ⟨AlgDefRSList⟩
(8)	⟨AlgDefRS⟩	→	⟨Id⟩
		|	⟨Id⟩ “`(`“ ⟨BasicTypeKL⟩ “`)`“
(9)	⟨BasicTypeKL⟩	→	⟨BasicType⟩
		|	⟨BasicType⟩ “`,`“ ⟨BasicTypeKL⟩

Mit den Produktionen (10) und (11) wird festgelegt, wie die Auswertungsstrategien für Funktionen angegeben werden können.

(10)	⟨EvalStratList⟩	→	ε
		|	⟨EvalStrat⟩ ⟨EvalStratList⟩
(11)	⟨EvalStrat⟩	→	“`CALL_BY_NAME`“ ⟨IdList⟩ “`;`“
		|	“`CALL_BY_VALUE`“ ⟨IdList⟩ “`;`“

Die Produktionen (11) bis (29) beschreiben den Aufbau einer Funktionsspezifikation, wobei die Produktionen (14) bis (17) den Aufbau der Deklaration und die Produktionen (18) bis (29) den Aufbau der Definition übernehmen.

(12)	⟨FuncSpec⟩	→	“`FUNC_SPEC`“ ⟨EvalStratList⟩ ⟨FuncDeclList⟩ ⟨FuncDefList⟩ “`END_FUNC_SPEC`“
(13)	⟨FuncDeclList⟩	→	⟨FuncDecl⟩ “`.`“
		|	⟨FuncDecl⟩ “`;`“ ⟨FuncDeclList⟩
(14)	⟨FuncDecl⟩	→	⟨Id⟩ “`:`“ ⟨FuncType⟩
(15)	⟨FuncType⟩	→	⟨TypeList⟩ “`->`“ ⟨TypeList⟩
(16)	⟨TypeList⟩	→	⟨BasicType⟩ “`x`“ ⟨TypeList⟩
		|	⟨BasicType⟩
(17)	⟨BasicType⟩	→	“`CARDINAL`“
		|	“`INTEGER`“
		|	“`BOOLEAN`“
		|	“`REAL`“
		|	“`CHARACTER`“
		|	“`VOID`“
		|	⟨Id⟩
(18)	⟨FuncDefList⟩	→	⟨FuncDef⟩ “`;`“
		|	⟨FuncDef⟩ “`;`“ ⟨FuncDefList⟩
(19)	⟨FuncDef⟩	→	⟨PMFuncDef⟩
		|	⟨IfFuncDef⟩
(20)	⟨PMFuncDef⟩	→	⟨PMEquation⟩
		|	⟨PMEquation⟩ ⟨PMFuncDef⟩
(21)	⟨PMEquation⟩	→	⟨LinkeSeite⟩ “`=`“ ⟨RechteSeite⟩
(22)	⟨IfFuncDef⟩	→	⟨LinkeSeite⟩ ⟨IfDefList⟩

(23)	⟨IfDefList⟩	→	“=“ ⟨RechteSeite⟩ “,“ ⟨IfCond⟩
		|	“=“ ⟨RechteSeite⟩ “,“ ⟨IfCond⟩ “,“ ⟨IfDefList⟩
(24)	⟨IfCond⟩	→	“IF“ ⟨Expr⟩
		|	“OTHERWISE“
(25)	⟨LinkeSeite⟩	→	⟨Id⟩ “(“ ⟨OptParList⟩ “)“
(26)	⟨OptParList⟩	→	ε
		|	⟨ParList⟩
(27)	⟨ParList⟩	→	⟨Param⟩
		|	⟨Param⟩ “,“ ⟨ParList⟩
(28)	⟨Param⟩	→	⟨Constant⟩
		|	⟨Id⟩
		|	⟨Id⟩ “(“ ⟨IdList⟩ “)“
		|	⟨Id⟩ “+“ ⟨NumToken⟩
(29)	⟨IdList⟩	→	⟨Id⟩
		|	⟨Id⟩ “,“ ⟨IdList⟩

Die folgenden Produktionen (30) bis (34) legen den Aufbau einer rechten Seite einer Funktionsdefinition fest.

(30)	⟨RechteSeite⟩	→	“(“ ⟨Expr⟩ “,“ ⟨ExprList⟩ “)“
		|	⟨Expr⟩
(31)	⟨ExprList⟩	→	⟨Expr⟩ “,“ ⟨ExprList⟩
		|	⟨Expr⟩
(32)	⟨Expr⟩	→	⟨SecExpr⟩
		|	⟨SecExpr⟩ ⟨Infix⟩ ⟨SecExpr⟩
(33)	⟨SecExpr⟩	→	⟨Id⟩
		|	⟨Constant⟩
		|	“(“ ⟨Expr⟩ “)“
		|	⟨Id⟩ ⟨Application⟩
(34)	⟨Application⟩	→	“()“
		|	“(“ ⟨ExprList⟩ “)“

In den Produktionen (35) bis (38) ist der Aufbau von allgemein verwendeten Objekten wie Konstante oder Zeichen angegeben.

(35)	⟨Infix⟩	→	“=“
		|	⟨InfixToken⟩
		|	“+“
		|	“<“
		|	“>“
(36)	⟨Id⟩	→	⟨IdToken⟩
		|	⟨Char⟩
(37)	⟨Char⟩	→	“x“
		|	“z“
		|	⟨CharToken⟩

(38)	⟨Constant⟩	→	⟨NumToken⟩
		|	“`TRUE`“
		|	“`FALSE`“
		|	“`'`“ ⟨Char⟩ “`'`“

Die Produktionen (39) bis (55) schließlich legen den Aufbau einer tree transducer Spezifikation fest. Die Strukturierung erfolgt dabei in weiten Zügen analog zum Aufbau von gewöhnlichen Funktionsspezifikationen.

(39)	⟨MatSpec⟩	→	“`BEGIN_MAT`“ ⟨EvalStratList⟩ ⟨MatDeclList⟩ ⟨RootDecl⟩ ⟨MatDefList⟩ “`END_MAT`“
(40)	⟨MatDeclList⟩	→	⟨MatDecl⟩ “`.`“
		|	⟨MatDecl⟩ “`;`“ ⟨MatDeclList⟩
(41)	⟨MatDecl⟩	→	⟨Id⟩ “`: SYN WALK`“ ⟨TypeList⟩ “`->`“ ⟨TypeList⟩
		|	⟨Id⟩ “`: INH WALK`“ ⟨TypeList⟩ “`->`“ ⟨TypeList⟩
(42)	⟨MatDefList⟩	→	⟨MatDef⟩ “`;`“
		|	⟨MatDef⟩ “`;`“ ⟨MatDefList⟩
(43)	⟨MatDef⟩	→	⟨MatLS⟩ “`=`“ ⟨MatRS⟩
(44)	⟨MatLS⟩	→	⟨Id⟩ “`(`“ ⟨LSSynNode⟩ ⟨IdKList⟩ “`)`“
		|	⟨Id⟩ “`(`“ ⟨LSInhNode⟩ ⟨IdKList⟩ “`)`“
(45)	⟨MatRS⟩	→	“`(`“ ⟨MatExpr⟩ “`,`“ ⟨MatExprList⟩ “`)`“
		|	⟨MatExpr⟩
(46)	⟨MatExprList⟩	→	⟨MatExpr⟩ “`,`“ ⟨MatExprList⟩
		|	⟨MatExpr⟩
(47)	⟨MatExpr⟩	→	⟨MatSecExpr⟩
		|	⟨MatSecExpr⟩ ⟨Infix⟩ ⟨MatSecExpr⟩
(48)	⟨MatSecExpr⟩	→	⟨Id⟩
		|	Constant
		|	“`(`“ ⟨MatExpr⟩ “`)`“
		|	⟨Id⟩ ⟨MatAppl⟩
		|	⟨RSSynNode⟩
		|	⟨RSInhNode⟩
(49)	⟨MatAppl⟩	→	“`()`“
		|	“`(`“ ⟨MatExprList⟩ “`)`“
(50)	⟨LSSynNode⟩	→	“`<z:`“ Id “`>`“
(51)	⟨LSInhNode⟩	→	“`<z:`“ Id “`>.`“ ⟨NumToken⟩
(52)	⟨RSSynNode⟩	→	“`<z>.`“ ⟨NumToken⟩
(53)	⟨RSInhNode⟩	→	“`<z>`“
(54)	⟨RootDecl⟩	→	“`ROOT :`“ ⟨Id⟩ “`;`“
(55)	⟨IdKList⟩	→	ε
		|	“`,`“ ⟨Id⟩ ⟨IdKList⟩

Die Angabe der kontextfreien Grammatik findet sich als Eingabe für den Parsergenerator Ratatosk in den Dateien `nnm.tok` und `nnm.gra`, die auf den beigefügten Disketten enthalten sind.

Anhang B

Programm–Listings

In diesem Kapitel befinden sich in der eigentlichen Arbeit alle Listings des Übersetzers und des Maschinensimulators. In dieser — eigens für das WWW erstellten — Version haben wir diese weggelassen. Sollte sich der Leser dennoch für diese Listings interessieren, so möge er sich via e–mail an

`fhoudek@bach.informatik.uni-ulm.de`

wenden.

Literaturverzeichnis

[ASU86] A.V. Aho, R. Sethi und J.D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison Wesley, 1986.

[CF82] B. Courcelle und P. Franchi–Zannettacci. *Attribute grammars and recursive program schemes.* Theoretical Computer Science, 17:163–191, 235–257, 1982.

[Chu32] A. Church. *A set of postulates for the foundation of logic.* American Mathematical Society 2 (1932, 1933), Seiten 33–34, 346–366, 839–864, 1932.

[EGH+94] D. Ernst, D. Gluche, F. Houdek, A. Kühnemann, A. Mößle, E. Müller, S. Sablatnög, H. Vogler und E. Wieser. *NoName — Eine funktionale Sprache mit Unterstützung der primitiven Rekursion, Version 1.0.* Interner Bericht, Universität Ulm, 1994.

[EHM94] D. Ernst, F. Houdek und A. Mößle. *Technische Dokumentation zum NoName–Editor, Version 1.0.* Interner Bericht, Universität Ulm, 1994.

[Eng80] J. Engelfriet. *Some open questions and recent results on tree transducers and tree languages.* In: R.V. Book (Herausgeber): *Formal language theory; perspectives and open problems*, Seiten 241–286. New York, Academic Press, 1980.

[Ern95] D. Ernst. *Entwicklung und Implementierung eines syntaxgesteuerten Editors für die Sprache NoName.* Diplomarbeit, Universität Ulm, 1995.

[EV85] J. Engelfriet und H. Vogler. *Macro tree transducers.* Journal Computer System Science, 31:71–146, 1985.

[EV91] J. Engelfriet und H. Vogler. *Modular tree transducers.* Theoretical Compututer Science, 78:267–303, 1991.

[FGV91] H. Faßbender, K. Gladitz und H. Vogler. *Compiler–Based Implementation of Syntax–Directed Functional Programming.* Technischer Bericht 91–02, Universität Ulm, 1991.

[FH88] A.J. Field und P.G. Harrison. *Functional Programming.* Addison–Wesley, 1988.

[Fül81] Z. Fülöp. *On attributed tree transducers.* Acta Cybernetica, 5:261–279, 1981.

[Glo93] Michael Gloger. *Implementierung funktionaler Programmiersprachen: Codegenerierung, Speicherverwaltung und Testsysteme für Sprachen mit verzögerter Auswertung.* Deutscher Universitätsverlag, 1993.

[GW94] J.R. Groff und P.N. Weinberg. *LAN times guide to SQL.* McGraw–Hill, 1994.

[Hen80] Peter Henderson. *Functional Programming — Application and Implementation.* Prentice–Hall International, 1980.

[Her92] H. Herold. *LEX und YACC: lexikalische und syntaktische Analyse.* Addison–Wesley (Deutschland), 1992.

[HW88] P. Hudak und P. Wadler. *Report on the functional programming language Haskell.* Technischer Bericht YALEU/DCS/RR656, Yale University, 1988.

[Jon91] M.P. Jones. *GOFER — An Introduction to Gofer*, 1991.

[Kla89] H. Klaeren. *Embedding functional described abstract data types into Modula–2 programs.* In: *First International Modula–2 Conference*, 1989.

[KV94] A. Kühnemann und H. Vogler. *Synthesized and inherited functions — a new computational model for syntax–directed semantics.* Acta Informatica, 31:431–477, 1994.

[Lan64] P.J. Landin. *The mechanical evaluation of expressions.* Computer Journal, 6(4):308–320, 1964.

[Loo90] R. Loogen. *Parallele Implementierung funktionaler Programmiersprachen*, Band 232 der Reihe *Informatik Fachberichte.* Springer–Verlag, 1990.

[May95] R. Mayer. *Testbericht über den NoName–Editor.* Interner Bericht, Universität Ulm, 1995.

[Mil84] R. Milner. *A proposal for Standard ML.* In: *ACM Conference on LISP and Functional Programming*, ACM, Seiten 184–197, 1984.

[Mog93] T. Mogensen. *Ratatosk — A Parser Generator and Scanner Generator for Gofer*, 1993.

[Pey87] Simon L. Peyton Jones. *The implementation of functional programming languages.* Prentice–Hall International, 1987.

[Rou70] W.C. Rounds. *Mappings and grammars on trees.* Mathematical System Theory, 4:257–287, 1970.

[Tha70] J.W. Thatcher. *Generalized^2 sequential machine maps.* Journal of Computer System Science, 4:339–367, 1970.

[Tur85] D.A. Turner. *Miranda: A non–strict functional language with polymorphic types.* In: J.-P. Jouannaud (Herausgeber): *Functional Programming Languages and Computer Architecture*, Nummer 201 in *Lecture Notes in Computer Science*, Seiten 1–16. Springer Verlag, September 1985.

[Wad95] Philip Wadler. *Monads in functional programming.* In: *Advanced Functional Programming: First international Spring School on advanced functional programming techniques*, Band 925 der Reihe *Incs*, Seiten 24–52. Springer Verlag, Mai 1995.

[Wal74] H. Waldschmidt. *Optimierungsfragen im Compilerbau.* Hanser Verlag, 1974.

[Wir86] N. Wirth. *Compilerbau*, Band 36 der Reihe *Leitfäden der angewandten Mathematik und Mechanik.* Teubner Verlag, 4 Auflage, 1986.

[WM92] R. Wilhelm und D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung.* Springer Verlag, 1992.