Kleine Vortragsreihe mit Demonstrationen zu

NoName - Eine funktionale Programmiersprache und ihr syntaxgesteuerter Editor

D. Ernst, F. Houdek, A. Mößle, E. Müller
Abt. Programmiermethodik und Compilerbau
Universität Ulm

H. Vogler
Grundlagen der Programmierung, Institut für Softwaretechnik I
Technische Universität Dresden

Dresden, den 28.09.1995

Vorwort

Dieses Heftchen dokumentiert die kleine Vortragsreihe mit Demonstrationen zum Thema „NoName - Eine funktionale Programmiersprache und ihr syntaxgesteuerter Editor“, welche am 28.09.1995 an der TU Dresden durchgeführt wurde.

In diesem Vorwort möchten wir kurz auf die Geschichte der vorgetragenen Forschungsarbeiten eingehen und somit die einzelnen Vorträge in einen Kontext stellen.

Ausgangspunkt der Forschungsarbeiten war das DFG-Projekt mit dem Thema

Integration der iterativen Auswertungsstrategie für primitiv-rekursive Funktionen in die Implementierung von funktionalen Programmiersprachen.

Dieses Projekt wurde am 23.12.1992 (und der Fortsetzungsantrag [1] am 09.05.1995) mit einem Umfang von insgesamt

- einem wissenschaftlichen Mitarbeiter BAT IIa für 4 Jahre und
- einer studentischen Hilfskraft zu 19h/Woche für 3 Jahre (aufteilbar)

bewilligt. Die Arbeit begann am 01.04.1993 an der Universität Ulm unter Leitung von Herrn Vogler und wird voraussichtlich am 31.03.1997 enden. Die Stellen wurden durch Frau Dipl.-Math. Andrea Mößle (Abt. Programmiermethodik und Compilerbau der Universität Ulm) bzw. (im wesentlichen) durch die Herren Dietmar Ernst, Frank Houdek und Erich Müller (alle Universität Ulm) besetzt. Die genannten Herren sind Studenten im Diplomstudiengang Informatik und fertigen zur Zeit ihre Diplomarbeiten unter Betreuung von Frau Mößle und Herrn Vogler an.

Das Hauptproblem, welches im o.a. DFG-Projekt gestellt wurde, ist folgendes:

> Gegeben ist eine funktionale Programmiersprache. Finde Programmtransformationen, welche die primitive Rekursion in Iteration (ohne Benutzung eines Kellers) umwandeln, so daß die entstehenden Programme effizienter laufen als die ursprünglichen Programme.

Für die Fibonacci-Funktion ist die Lösung dieses Problems seit langem bekannt. Wenn etwa diese Funktion in der Form

$$fib(suc(x)) = fib(x) + h(x)$$

$$h(suc(x)) = fib(x)$$

$$fib(nul) = 0$$

$$h(nul) = 1$$

durch simultane Rekursion gegeben ist (wobei die natürliche Zahl n durch den Term $\underbrace{suc(\ldots suc(}_{n\,\mathrm{mal}} nul \underbrace{)\ldots)}_{n\,\mathrm{mal}}$ gegeben ist), dann liefert die Transformation in die iterative Form das folgende Programm:

$$fib(x) = fibiter(x, 0, 1)$$

[1] gemeinsamer Antrag mit Prof. Dr. H. Partsch, Universität Ulm

$$fibiter(suc(x), y, z) = fibiter(x, y + z, y)$$

$$fibiter(nul, y, z) = y$$

Die Vorzüge der iterativen Form sind allgemein bekannt: Während die Laufzeit und der Platzbedarf bei der rekursiven Form exponentiell bzw. linear mit der Größe der Eingabe wächst, ist die Komplexität der iterativen Form nur linear bzw. konstant.

Die Aufgabe des DFG-Projekts bestand nun darin, diese Transformation auch für primitiv-rekursiv definierte Funktionen (über algebraischen Datentypen) zu formulieren, welche nach einem allgemeineren Schema der primitiven Rekursion definiert sind. Hierauf geht der Vortrag von Frau A. Mößle mit dem Titel „Effiziente Auswertung von primitiv-rekursiven Programmschemata“ ein.

Natürlich bietet es sich bei dieser Themenstellung an, ein Softwaretool zu entwickeln, mit dessen Hilfe die Programmtransformationen getestet werden können. Dazu wurde zunächst eine kleine funktionale Programmiersprache namens NoName definiert (siehe Vorträge von F. Houdek: „Die funktionale Programmiersprache NoName“ und von F. Houdek: „Implementierung von first-level NoName auf einer Runtime-Stack-Maschine“). NoName-Programme zeichnen sich durch eine Syntax aus, die sehr nahe an der in der Mathematik üblichen Notation für Funktionen liegt.

Das Softwaretool besteht aus einem syntaxgesteuerten Editor für NoName-Programme und einem Informationssystem NoMIS (siehe Vortrag von E. Müller: „Entwicklung eines Modul-Informationssystems für den NoName-Editor“), welches als Datenbank für NoName-Programm(teile) fungiert. Der Editor wurde von einem gekauften Tool erzeugt, welches den Namen „SGen -synthesizer generator“ trägt (siehe Vortrag von D. Ernst und E. Müller: „Der SGen und der NoName-Editor“). Der SGen nimmt als Eingabe die Spezifikation der kontextfreien und kontextsensitiven Syntax einer Programmiersprache (in Form einer Attributgrammatik) und liefert als Ausgabe ein C-Programm, welches den syntaxgesteuerten Editor der spezifizierten Programmiersprache darstellt. Dieser Editor ist nichts anderes als ein inkrementeller Attributauswerter (siehe Vortrag von H. Vogler: „Attributgrammatiken und inkrementelle Auswertung“).

Die Entwicklung des Softwaretools und die Implementierung von NoName wurden unter Anleitung von Frau A. Mößle durch die Herren D. Ernst, F. Houdek und E. Müller im Rahmen ihrer Diplomarbeiten an der Universität Ulm durchgeführt. Die hier dokumentierte Vortragsreihe gibt einen Einblick in diese Entwicklungsarbeiten.

Programm

10.00 - 10.45	Heiko Vogler	Begrüßung Attributgrammatiken und inkrementelle Auswertung
10.45 - 11.15	Frank Houdek	Die funktionale Programmiersprache NoName
11.15 - 12.00	Dietmar Ernst und Erich Müller	Der SGen und der NoName-Editor

MITTAGSPAUSE

13.30 - 14.15	Dietmar Ernst	Implementierung und Erweiterung des NoName-Editors
14.15 - 15.00	Erich Müller	Entwicklung eines Modul-Informationssystems für den NoName-Editor

KAFFEEPAUSE

15.30 - 16.15	Frank Houdek	Implementierung von first-level NoName auf einer Runtime-Stack-Maschine
16.15 - 17.00	Andrea Mößle	Effiziente Auswertung von primitiv-rekursiven Programmschemata

ENDE

Die Zusammenfassungen der Vorträge 2 bis 6 haben ein gemeinsames Literaturverzeichnis, welches sich auf den Seiten 30 und 31 befindet.

Dresden, 28.09.1995
D. Ernst, F. Houdek, A. Mößle, E. Müller, H. Vogler

Attributgrammatiken und inkrementelle Attributauswerter

H. Vogler
Grundlagen der Programmierung, Fakultät Informatik
Technische Universität Dresden
Dresden, Germany

4. September 1995

In diesem Beitrag soll kurz das Konzept der Attributgrammatik und der inkrementellen Attributauswertung erläutert werden.

Attributgrammatiken wurden von D. Knuth [Knu68] entwickelt und in zahlreichen Arbeiten studiert (siehe [DJ90, DJL88, AM91] für Übersichten, Einführungen und Literatur). Eine Attributgrammatik dient dazu - für eine gegebene kontextfreie Grammatik G und einen Ableitungsbaum t - einen Informationstransport über t zu bewirken. Wenn z.B. G den kontextfreien Teil der Programmiersprache NoName generiert, dann könnte eine Attributgrammatik für ein NoName-Programm P die Information über die Deklariertheit von Programmvariablen in den Anweisungsteil von P transportieren. Dort kann dann die kontextsensitive Eigenschaft „declared before used" überprüft werden.

Formal gesehen ist ein Attributgrammatik eine kontextfreie Grammatik, in der jedes Nichtterminalsymbol mit einer Menge von Attributen versehen ist. Die Attribute sind eingeteilt nach *synthetisch* und *inherit*; die synthetischen Attribute bewirken einen Informationstransport von den Blättern eines Ableitungsbaums in Richtung Wurzel (und damit synthetisieren sie Information), die inheriten Attribute bewirken einen Informationstransport von der Wurzel in Richtung Blätter (und damit vererben sie Information). Der Informationstransport wird lokal zu den Produktionen festgelegt. D.h. jede Produktion ist mit einer Menge von semantischen Regeln versehen, mir deren Hilfe die Attribute der in der Produktion auftretenden Nichtterminale berechnet werden können. Werden die Produktionen nun - so wie in einem Ableitungsbaum - zusammengefügt, so entsteht aus der lokalen Spezifikation des Informationstransports ein globaler Informationstransport über den gesamten Ableitungsbaum.

Es gibt verschiedene Verfahren, nach denen für einen vorgelegten Ableitungsbaum die Werte der Attribute ausgerechnet werden können. Programme, die diese Rechnungen vornehmen, heißen *Attributauswerter*. Wir verweisen den Leser auf [Eng94] für eine Übersicht über Attributauswerter. In diesem kurzen Beitrag möchten wir die sogenannten *plangesteuerten Attributauswerter* für geordnete Attributgrammatiken [Kas80] näher erläutern.

Ein plangesteuerter Attributauswerter $plAA$ beruht auf folgender Vorbereitung. Jeder Produktion p mitsamt ihrer semantischen Regeln wird ein Plan $plan(p)$ zugeordnet. Dieser Plan ist eine Sequenz $s_1, \ldots, s_k$, wobei jedes s_i eine der drei folgenden Befehle ist:

- berechne Attribut $\langle \alpha, j \rangle$,
- besuche den j-ten Nachfolger des aktuellen Knotens ($visit(j)$) und
- gehe zum Vorgänger des aktuellen Knotens zurück ($suspend$).

Wird der $plAA$ nun für einen vorgelegten Ableitungsbaum t gestartet, so führt er einen Baumlauf über t aus. Wenn $plAA$ zum ersten mal den Knoten n besucht, an dem die Produktion p angewendet ist, dann führt $plAA$ die Befehle des Plans $plan(p)$ schrittweise von links nach rechts aus. Trifft er dabei auf einen Befehl der ersten Sorte, so berechnet er den Wert des angegebenen Attributs. Trifft er auf einen $visit(j)$-Befehl, so unterbricht er die Arbeit in $plan(p)$ an dieser Stelle (nennen wir sie einmal ra (return address)) und besucht den j-ten Nachfolger von n. Dort verfährt $plAA$ genauso wie am Knoten n. Trifft $plAA$ auf den Befehl *suspend*, so unterbricht er seine Arbeit in $plan(p)$ ebenfalls an dieser Stelle (auch ra genannt) und geht zum Vorgänger von n, dort verfährt $plAA$ ebenfalls genauso wie am Knoten n. Kommt nun $plAA$ zum Knoten n zurück, so nimmt er die Abarbeitung von $plan(p)$ an der Stelle ra wieder auf.

Syntaxgesteuerte Editoren sind Softwaretools, die eine Überprüfung der kontextsensitiven Syntax bei der inkrementellen Erstellung eines Programms vornehmen. Jeder Editor bezieht sich auf eine Programmiersprache, deren kontextfreie und kontextsensitive Syntax durch eine Attributgrammatik gegeben ist. Vom Standpunkt der Attributgrammatik aus ist der syntaxgesteuerte Editor ein *inkrementeller Attributauswerter*.

Bei der Arbeit mit einem syntaxgesteuerten Editor wird fortwährend das Programm (und damit der dazugehörende Ableitungsbaum) erweitert (und manchmal auch verkleinert). Bei der inkrementellen Attributauswertung kommt es nun darauf an, sowenig Attributwerte nach einer Veränderung des Ableitungsbaums wie möglich noch einmal zu berechnen. Das für einen neu eingefügten Teil alle Attributwerte berechnet werden müssen, ist klar. Aber alle anderen Attribute haben bereits einen Wert, welcher manchmal beibehalten werden kann.

Ein plangesteuerter Attributauswerter kann durch folgende Maßnahmen zu einem inkrementellen Attributauswerter erweitert werden. Wenn ein Ableitungsbaum t am Knoten n einen neuen Teilbaum erhält, so wird der $plAA$ am Vorgänger n' von n gestartet; dort führt er den Plan $plan(p')$ vom Anfang an aus, wobei p' die an n' angewendete Produktion ist. Tritt nun ein $visit(i)$-Befehl auf und es hat sich keiner der Werte der inheriten Attribute am i-ten Nachfolger von n' verändert, so besteht natürlich auch kein Bedarf, den i-ten Teilbaum zu besuchen: dort wird sich kein Attributwert ändern. Ähnliches gilt für den *suspend*-Befehl: wenn sich keiner der Werte von synthetischen Attributen am Vorgänger von n' geändert hat, so braucht $plAA$ den Vorgänger auch nicht zu besuchen. Es geht dem inkrementellen plangesteuerten Attributauswerter also darum, möglichst viele Besuche in die Umgebung eines Knotens zu vermeiden.

Literatur

[AM91] H. Alblas and B. Melichar, editors. *Attribute grammars, applications and systems.* Springer-Verlag, 1991. LNCS 545.

[DJ90] P. Deransart and M. Jourdan, editors. *Attribute grammars and their applications.* Springer-Verlag, 1990. LNCS 461.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. *Attribute grammars, definitions and bibliography.* Springer-Verlag, 1988. LNCS 323.

[Eng94] J. Engelfriet. Graph grammars and tree transducers. In S. Tison, editor, *Trees in algebra and programming - CAAP 94*, pages 15–36. Springer-Verlag, 1994.

[Kas80] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13:22–, 1980.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2:127–145, 1968.

Die funktionale Programmiersprache NoName

Frank Houdek
Universität Ulm, Fakultät für Informatik
Abt. Programmiermethodik und Compilerbau
89069 Ulm, Germany
E–mail: fhoudek@bach.informatik.uni-ulm.de

Einführung

Die funktionale Programmiersprache NoName wurde 1994 an der Universität Ulm im Rahmen eines Praktikums im Hauptstudium von einer Gruppe Studenten unter Leitung von Prof. Dr. Heiko Vogler entwickelt. Die Sprache besitzt eine Reihe von Eigenschaften, die in dieser Zusammenstellung bei keiner anderen Programmiersprache zu finden sind:

- **Strenges Typkonzept**: Jede Funktion hat eine genau festgelegte Stelligkeit und der Typ jeder Parameterposition ist festgelegt. Die Stelligkeit und Typisierung müssen bei jedem Funktionsaufruf beachtet werden. Der Aufruf einer Funktion mit weniger Parametern als in der Funktionsdeklaration festgelegt (sog. Unterversorgung) ist folglich nicht möglich.

- **Mathematische Syntax**: Die Notation einer Funktionsdefinition orientiert sich mehr an der Mathematik als am λ–Kalkül, das vielen anderen funktionalen Programmiersprachen zugrundeliegt.

- **Verschiedene Auswertungsstrategien**: Für jede Funktion kann explizit festgelegt werden, ob diese mit der Auswertungsstrategie Call–by–Value oder Call–by–Name berechnet werden soll.

- **Unterstützung von tree transducern**: Durch eigene syntaktische Konstrukte wird die einfache Angabe von tree transducern ermöglicht. Unterstützt werden top–down tree transducer, macro tree transducer, attributed tree transducer, macro attributed tree transducer und modular tree transducer.

- **Strukturierung und Modularisierung**: Die Sprache unterstützt durch eigene Schlüsselworte die Möglichkeit, gut strukturierte Programme zu erstellen. Mit Hilfe der IMPORT–Anweisung ist es möglich, Funktionen und Datentypen aus anderen Programmen zu nutzen. Der Aufbau von Bibliotheksmodulen ist somit möglich.

Im folgenden Abschnitt wird die Syntax von NoName anhand einiger Beispiele eingeführt. Eine genaue Beschreibung der Sprache findet sich in [EGH+94].

Die Sprache NoName

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 Binärbäume erzeugt werden können, ist in Abbildung 1 wiedergegeben.

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

Abb. 1: Definition des algebraischen Datentyps `Baum`.

Außerdem können in NoName, wie die Abbildung 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. 2: Definition von Datentypsynonymen.

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

Funktionsspezifikation

Funktionen werden innerhalb von Spezifikationen erklärt. Eine solche Spezifikation besteht aus mehreren Funktionsdeklarationen und den zugehörigen Funktionsdefinitionen. In Abbildung 3 ist ein Beispiel für die Spezifikation einer Funktion wiedergegeben.

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

Abb. 3: Einfache Funktionsspezifikation.

Das Zusammenfassen mehrerer Funktionen in einer Spezifikation kann als Strukturierungshilfe verwendet werden.

Außer der in Abbildung 3 angegebenen einfachen Definition kann eine Funktionsdefinition auch mittels IF oder Pattern Matching erfolgen. Eine Mischung dieser beiden Verfahren ist aber nicht zulässig. In Abbildung 4 ist die Fibonacci–Funktion mittels Pattern Matching definiert.

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

Abb. 4: Fibonacci mit Pattern Matching.

Die verschiedenen Funktionsgleichungen des Pattern Matching müssen überschneidungsfrei und erschöpfend sein, d.h. auf jeden möglichen Eingabevektor der Funktion muß genau eine Funktionsgleichung passen. In Abbildung 5 ist die Fibonacci–Funktion nun mittels IF definiert.

```
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. 5: Fibonacci mittels IF.

Die Angabe eines `OTHERWISE`–Falles ist nicht zwingend erforderlich. Zur Angabe von Funktionen höherer Ordnung, die in NoName ebenfalls definiert werden können, steht das Schlüsselwort `WHERE` zur Verfügung, mit dessen Hilfe lokale Funktionen eingeführt werden können. In Abbildung 6 ist ein Beispiel für die Definition einer höhere Funktion (`inc_n`) unter Verwendung einer lokalen Funktion (`help_inc_n`) gegeben.

```
FUNC_SPEC
  inc_n : CARDINAL -> (CARDINAL -> CARDINAL).
  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. 6: Funktion höherer Ordnung.

Definition von tree transducern

Tree transducer beschreiben — ganz allgemein betrachtet — ein Schema, mit dem eine Baum zu Baum–Übersetzung beschrieben werden kann. NoName unterstützt als bisher einzige Programmiersprache auch auf syntaktischer Ebene die Definition von verschiedenen tree transducern, nämlich top–down tree transducer [Rou70, Tha70], macro tree transcuder [Eng80, CF82, EV85], attributed tree transducer [Fül81], macro attributed tree transducer [KV94] und modular tree transducer [EV91]. Diese verschiedenen tree transducer sollen hier jedoch nicht vorgestellt werden. Der interessierte Leser sei diesbezüglich auf die Literatur verwiesen.

In NoName ist das Konzept der tree transducer ein wenig gegenüber dem aus der Literatur bekannten Konzept erweitert worden. So muß das Funktionsergebnis einer tree transducer Funktion kein Baum sein; mit Ausnahme von Funktionen höherer Ordnung dürfen alle zur Verfügung stehenden Datentypen verwendet werden. Ebenfalls sind für die Kontextparameter, die einige tree transducer neben dem Rekursionsargument besitzen dürfen, beliebige Typen zulässig.

In Abbildung 7 ist die Definition eines macro attributed tree transducers in NoName–Syntax wiedergegeben. Der in diesem Beispiel verwendete algebraische Datentyp `Baum` wurde bereits in Abbildung 1 definiert.

```
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>,1);
   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
```

Abb. 7: Spezifikation eines macro attributed tree tranducers.

Weitere Eigenschaften der Sprache NoName

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

In der Einleitung wurde bereits angedeutet, daß für jede Funktion explizit eine Auswertungsstrategie angegeben werden kann. Hierfür stehen die Strategien *Call–by–Name* und *Call–by–Value* zur Verfügung. Die Angabe der Auswertungstrategie erfolgt, indem hinter `FUNC_SPEC` bzw. `BEGIN_MAT` der Text

`CALL_BY_NAME` id_1, id_2, ..., id_n;

für die Auswertungsstrategie Call–by–Name und

`CALL_BY_VALUE` id_1, id_2, ..., id_n;

für die Auswertungsstrategie Call–by–Value gesetzt wird. Die Angabe von zwei Auswertungsstrategien für eine Funktion ist nicht zulässig. Wird für eine Funktion keine Auswertungsstrategie angegeben, so kann diese beliebig festgelegt 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 $id_1, \ldots, id_n$ benötigte Hilfsfunktionen und Datentypen in P verwendet werden.

Ausblick

Mit der Sprache NoName ist bisher intensiv gearbeitet worden ([May95]). Aus den Beobachtungen, die bei dieser Arbeit gewonnen wurden, konnten eine Reihe von Ansatzpunkten für Erweiterungen abgeleitet werden. In folgenden werden drei dieser Ansatzpunkte skizziert.

- Unterstützung der Erstellung von Listen.
  Mit algebraischen Datentypen können Listenstrukturen gebildet werden. Allerdings sind diese durch die notwendigen Konstruktoren, die mit im Ergebnis erscheinen, ziemlich schwer lesbar. Eine eigene Notation, insbesondere für Listen über Zeichen (Strings), ist daher sinnvoll.
- Untertsützung von Polymorphie.
  Durch das strenge Typkonzept ist es notwendig, eine Funktionen, die für unterschiedliche Datentypen benötigt wird, mehrfach zu implementieren. Dadurch wird die Programmierung unnötig erschwert.
- Ausgabe von Fehlern.
  In der momentanen Fassung von NoName ist es nicht möglich, benutzergesteuert Fehlermeldungen erzeugen zu lassen. Da viele Funktionen aber nicht für alle möglichen Eingaben definiert sind, ist eine solche Ausgabemöglichkeit wünschenswert.

Der SGen und der NoName-Editor

Dietmar Ernst, Erich Müller
Universität Ulm, Fakultät für Informatik,
Abt. Programmiermethodik und Compilerbau,
89069 Ulm, Germany
E-mail: {dietmar,erich}@bach.informatik.uni-ulm.de

Einleitung

Der Synthesizer Generator (kurz: SGen) der Firma GRAMMATECH ist ein Werkzeug zur Erstellung syntaxgesteuerter Editoren. Ein derartiger Editor ist für genau eine Programmiersprache geeignet und soll dem Benutzer eine vereinfachte Programmentwicklung bieten. Der Editor basiert auf einer kontextfreien Grammatik, durch welche die Syntax der Programmiersprache festgelegt ist. Der Benutzer hat die Möglichkeit, Programme menügesteuert top-down zu entwerfen. Dabei werden Verletzungen der *kontextsensitiven Nebenbedingungen* der Sprache sofort erkannt und als Fehlermeldung angezeigt. Eine denkbare Verletzung kontextsensitiver Nebenbedingungen ist z.B. ein angewandtes Vorkommen einer Variablen, dem kein definierendes Vorkommen vorausgeht. Der Benutzer kann diesen Fehler löschen, indem er die Deklaration nachträglich einfügt. Die Überprüfung der kontextsensitiven Nebenbedingungen wird mittels Attributgrammatiken erreicht (siehe [Knu68] und [Vog93]). Die Auswertung der Attribute erfolgt dabei inkrementell, so daß bei einer Änderung des Programms nur die Werte der davon betroffenen Attribute neu berechnet werden müssen.

Erstellung eines syntaxgesteuerten Editors

Die Spezifikation eines syntaxgesteuerten Editors erfolgt in der Sprache SSL (Synthesizer Specification Language). Die folgenden Angaben können vom Programmierer gemacht werden:

Abstrakte Syntax: Mit der abstrakten Syntax wird der syntaktische Aufbau der zugrundeliegenden Programmiersprache beschrieben. Im Gegensatz zur konkreten Syntax (also der zugrundeliegenden kontextfreien Grammatik) enthält die abstrakte Syntax keine Terminalsymbole.

Eingabesyntax: Die Eingabesyntax legt fest, wie aus einer eingegebenen Zeichenfolge ein Syntaxbaum erzeugt wird.

Ausgabesyntax: Die Ausgabesyntax legt die Darstellungsform des aktuell bearbeiteten Syntaxbaumes fest.

Transformationen: Mit einem syntaxgesteuerten Editor können Programmfragmente „per Mausklick" weiterentwickelt werden. Dabei werden sogenannte Platzhalter durch Sprachkonstrukte, die selbst auch wieder Platzhalter enthalten dürfen, ersetzt.

Attributierung: Die kontextfreien Nebenbedingungen werden mit Attributgrammatiken abgeprüft. Die Attributauswertung übernimmt der Editor.

Externe Funktionen: Manchmal können nicht alle gewünschten Aufgaben mit den von GrammaTech vorgegebenen Funktionen und Menüpunkten erledigt werden. Für diese Fälle steht ein Interface zu der Programmiersprache C zur Verfügung, mit welchem eigene C–Funktionen in den Editor eingebunden werden können.

Ein SSL–Compiler übersetzt die Spezifikationen in einige C–Quelldateien, die dann — eventuell zusammen mit externen Funktionen — in ein ausführbares Programm übersetzt werden können.

Das Hauptfenster des NoName–Editors

Wir werden im folgenden einen syntaxgesteuerten Editor für die Sprache NoName vorstellen. Der größte Teil der Arbeit wird im Hauptfenster des Editors erledigt, welches nach dem Start auf dem Bildschirm angezeigt wird. Abbildung 1 zeigt das Hauptfenster des NoName–Editors.

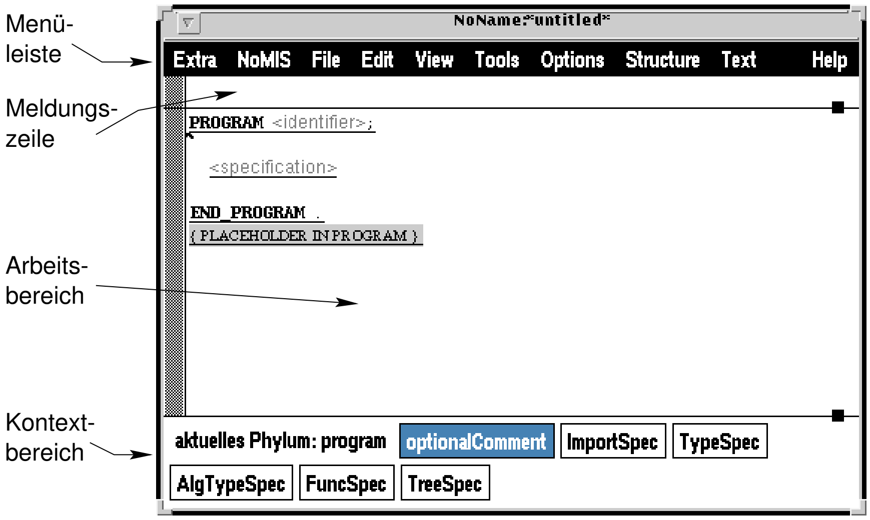

Abb. 1: Das Hauptfenster des NoName–Editors

Die Menüleiste enthält alle verfügbaren Menüs. Einige von diesen, z.B. das Menü File, sind standardmäßig in jedem erzeugten Editor vorhanden, andere wiederum, z.B. das Menü Extra, wurden vom Editor–Programmierer selbst hinzugefügt.

Die Nachrichtenzeile dient der Anzeige von Fehlermeldungen und vergleichbaren Nachrichten, etwa das erfolgreiche Laden einer Datei etc.

Im Arbeitsbereich wird das aktuell editierte Programm entwickelt.

Im Kontextbereich werden der aktuelle Kontext sowie die darin verfügbaren Transformations–Buttons angezeigt.

Arbeiten mit dem NoName–Editor

Um die Vorgehensweise bei der Arbeit mit dem Editor gut zu verstehen, ist es sinnvoll, immer im Gedächtnis zu behalten, daß der Editor das aktuelle Programm als Syntaxbaum im Speicher darstellt. Auch wenn der Benutzer nur die textuelle Repräsentation desselben sieht, so bewirken alle seine Änderungen eine Modifikation des Syntaxbaumes.

Nach dem Starten des Editors wird das Hauptfenster aus Abbildung 1 geöffnet. Darin ist bereits ein „leeres Programm“ vorgegeben. Der Benutzer kann nun ein NoName–Programm erstellen, indem er schrittweise die sogenannten Platzhalter — z.B. `<speclist>` — durch Konstrukte der Sprache NoName ersetzt. Diese Sprachkonstrukte können selbst wiederum Platzhalter enthalten. Der Ersetzungsvorgang wird solange durchgeführt, bis keine Platzhalter mehr vorhanden sind.

Prinzipiell existieren zwei Möglichkeiten, Platzhalter zu ersetzen:

1. Verwendung sogenannter *Transforms*: Dabei wird der aktuell markierte Platzhalter mittels Mausklick auf einen Transform–Button durch ein vordefiniertes Programmstück ersetzt.

2. Direkte Eingabe des Programmtextes über die Tastatur.

Die erste Variante hat den entscheidenden Vorteil, daß durch die Ersetzung mit vordefinierten Programmstücken die syntaktische Korrektheit des Programms erhalten bleibt. Bei der Eingabe über die Tastatur muß der Benutzer selbst dafür sorgen, daß die Syntaxregeln eingehalten werden.

Zu beachten ist, daß der aktuell markierte Platzhalter nur durch Programmfragmente desselben „Typs“ ersetzt werden können. Wenn z.B. der Platzhalter `<identifier>` markiert ist, dann kann dieser nur durch einen Bezeichner ersetzt werden. Eine Funktionsdefinition etwa ist an dieser Stelle nicht erlaubt.

Prüfung auf syntaktische und semantische Korrektheit

Der Editor kann prinzipiell zwei verschiedene Arten von Fehlern erkennen: syntaktische und semantische Fehler.

Syntaxfehler

Syntaxfehler werden unmittelbar nach dem Auftreten (bzw. nach dem Parsen des fehlerhaften Programmteils) in der Meldungszeile (vgl. Abbildung 1) angezeigt. Ein Syntaxfehler in der Sprache NoName tritt z.B. bei der Definition eines Tupel–Typs auf, wenn statt des Kleinbuchstabens `x` der Großbuchstabe `X` verwendet wird:

```
matrix = CARDINAL X CARDINAL
```

anstatt

```
matrix = CARDINAL x CARDINAL
```

Semantische Fehler

Fehler, die aufgrund der statischen Semantik entstehen, werden dem Benutzer direkt an der Fehlerquelle angezeigt. Zu diesen Fehlern gehören im wesentlichen Typfehler sowie die Verwendung nichtdeklarierter Bezeichner. Typfehler treten z.B. bei der Definition von Funktionen auf:

```
quad: INTEGER -> CARDINAL.
quad(x) = x*x
```

Da die Variable `x` offensichtlich den Typ `INTEGER` besitzt, ist auch der Ausdruck `x*x` des Typs `INTEGER`. Die Funktionsdeklaration schreibt aber hier den Typ `CARDINAL` vor, so daß ein Typfehler entsteht.

Wenn der Benutzer im obigen Beispiel statt `x*x` den Ausdruck `x*y` zur Berechnung der Funktion `quad` angibt, dann entsteht ein Fehler aufgrund der Verwendung der nichtdeklarierten Variablen `y`.

Einfache „Programmtransformationen“

Der NoName–Editor bietet kleinere „Programmtransformationen“ an. So kann z.B. die Angabe `CALL_BY_VALUE` bei der Deklaration von Funktionen mittels Transformation umgewandelt werden zu `CALL_BY_NAME` und umgekehrt.

Eine weitere Anwendung ist die Umwandlung einer Funktionsdefinition, die durch Pattern–Matching gegeben ist, in eine äquivalente Funktionsdefinition mittels IF–Bedingungen:

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

Durch Transformation ergibt sich:

```
FUNC_SPEC
  fak: CARDINAL -> CARDINAL.
  fak(x1) = 1,                            IF x1 = 0,
          = ((x1 - 1) + 1) * fak(x1 - 1), IF x1 >= 1;
END_FUNC_SPEC
```

Derartige Transformationen ersparen dem Benutzer das Löschen und anschließendes Neueingeben von Programmteilen.

Implementierung und Erweiterung des NoName–Editors

Dietmar Ernst
Universität Ulm, Fakultät für Informatik,
Abt. Programmiermethodik und Compilerbau,
89069 Ulm, Germany
E-mail: dietmar@bach.informatik.uni-ulm.de

Einführung

In den letzten Jahren wurde auf eine sorgfältige Erstellung von Programmen immer mehr Wert gelegt. Man wünschte sich Werkzeuge, die einen Programmierer bei seiner Arbeit unterstützen. Eine Möglichkeit ist dabei ein syntaxgesteuerter Editor. Ein solcher Editor nimmt seinem Benutzer manche Arbeit ab, z.B. kann er sofort auf Korrektheit von Syntax und statischer Semantik testen.

Das Ziel meiner Diplomarbeit (siehe [Ern95]) war es, für die Sprache NoName (siehe [EGH+94] mit Hilfe des „Synthesizer Generators“ (kurz: SGen) — einem Hilfswerkzeug zur Erstellung von syntaxgesteuerten Editoren — einen solchen Editor zu implementieren.

Der Test auf syntaktische Fehler wird durch einen Parser durchgeführt, der einen eingegebenen Text sofort als Programmfragment strukturieren möchte. Der Test auf Korrektheit der statischen Semantik ist wesentlich schwieriger. Hier bedienen wir uns mit dem Konzept der Attributgrammatiken. Da der Benutzer wahrscheinlich auch öfters kleine Programmteile ändern möchte, ist es unnötig, nach jeder Programmänderung alle Attributinstanzen neu zu berechnen. Deswegen arbeitet im Hintergrund des NoName–Editors ein inkrementeller Attributauswerter.

Die Arbeit der Diplomarbeit war also im wesentlichen die Implementierung einer Attributgrammatik, die ein syntaktisch korrektes NoName–Programm auch auf Korrektheit bezüglich der statischen Semantik testet. Eine weitere Aufgabe war die Implementierung von verschiedenen Funktionen, die die Arbeit mit dem Editor erleichtern. Als Beispiel sei hier eine Funktion genannt, mit deren Hilfe es möglich ist, ein eingegebenes Programm auch auszuführen.

Die Attributierung

In diesem Abschnitt werden die grundlegenden Ideen beschrieben, die die Attributierung beeinflußt haben. Auf die genaue Implementierung kann aus Platzgründen nicht eingegangen werden.

Ziele für die Attributgrammatik

Der SGen bietet die Möglichkeit, sogenannte Fehlerattribute zu definieren. Der Wert eines solchen Fehlerattributs, der immer eine Zeichenfolge sein muß, wird an einer gewünschten Stelle im Programm eingefügt. Damit ist es also möglich, Fehler direkt an der Fehlerstelle auszugeben. Abbildung 8 zeigt eine fehlerhafte

Funktionsdeklaration. Es ist auch wichtig, Fehlermeldungen möglichst direkt an der Fehlerstelle auszugeben, da dem Benutzer die Korrektur dieser Fehler dann sehr viel leichter fällt.

```
ALG_DEF
  Baum = Null
       | Paar( Baum, Baum );
END_ALG_DEF

FUNC_SPEC
  fak : CARDINAL -> card { IDENTIFIER NOT DECLARED } ;
  count : Null { NO TYPE OR ALG_TYPE }  -> CARDINAL.
  <funcdef>;
END_FUNC_SPEC
```

Abb. 8: Beispiel für eine fehlerhafte Funktionsdeklaration

In Abbildung 8 wird noch ein zweite Aufgabe für die Attributierung deutlich. In der Sprache NoName muß jede deklarierte Funktion auch definiert werden. In diesem Beispiel fehlt für die zwei Funktionen `fak` und `count` noch eine Funktionsdefinition. Trotzdem ist noch keine entsprechende Fehlermeldung vorhanden. Der Grund ist der folgende: In dieser Funktionsspezifikation ist noch ein Platzhalter für eine Funktionsdefinition vorhanden. Wir können also annehmen, daß der Programmierer später diesen Platzhalter noch für eine konkrete Funktionsdefinition verwenden wird. Betrachten wir die Abbildung 9:

```
ALG_DEF
  Baum = Null
       | Paar( Baum, Baum );
END_ALG_DEF

FUNC_SPEC
  fak : CARDINAL -> CARDINAL;
  count { FUNCTION IS NOT DEFINED }  : Baum -> CARDINAL.
  fak(0) = 1
  fak(n+1) = (n + 1) * fak(n);
END_FUNC_SPEC
```

Abb. 9: Die Funktion `count` ist zwar deklariert, aber noch nicht definiert.

Inzwischen wurden die angezeigten Fehler behoben und die Funktion `fak` definiert. Da jetzt kein Platzhalter mehr für eine Funktionsdefinition existiert, erscheint auch eine Fehlermeldung, die besagt, daß die Funktion `count` zwar deklariert, aber noch nicht definiert wurde.

Durch dieses Beispiel wird deutlich, daß die Erstellung der Attributgrammatik stark vom Vorhandensein etwaiger Platzhaltern abhängig ist. Als grundsätzliche Regel für die Behandlung von Platzhaltern wurde die folgende gewählt:

> Ein NoName–Programm ist auch dann korrekt, wenn etwaige Fehler noch durch Verfeinerung von schon vorhandenen Platzhaltern korrigiert werden können.

Die Symboltabelle

In realen Übersetzern wird zur Verwaltung von Bezeichnern eine Symboltabelle angelegt, in der gespeichert wird, welche Bedeutung jeder Bezeichner besitzt. Dieser Technik bedienen wir uns auch bei der Attributierung von NoName. Ein inherites und ein synthetisches Attribut berechnen gemeinsam in einem left–to–right tree–traversal diese Symboltabelle, indem bei jedem deklarierenden oder definierenden Vorkommen eines Bezeichners die Symboltabelle um diesen Bezeichner angereichert wird.

Wie sieht diese Tabelle nun aus? Es ist klar, daß es sich im wesentlichen um eine Liste von Einträgen handelt. Jeder Eintrag besteht aus zwei Komponenten: Die erste gibt an, welcher Bezeichner hier definiert wird; die zweite Komponente speichert die genaue Typisierung dieses Bezeichners.

Allerdings reichen diese Informationen noch nicht aus. Betrachten wir dazu folgendes Beispiel:

```
TYPE_DEF
  funcType = CARDINAL -> CARDINAL;
END_TYPE_DEF

FUNC_SPEC
  pow2 : CARDINAL -> CARDINAL.
  pow2(x) = x * x;
END_FUNC_SPEC
```

Wir haben einen allgemeinen Datentyp und eine Funktion definiert. Sowohl `funcType` als auch `pow2` besitzen den Typ `CARDINAL -> CARDINAL`. Ohne weitere Informationen können wir also in einer Anwendung nicht entscheiden, ob sie korrekt ist oder nicht: Der Ausdruck `pow2(3)` ist natürlich korrekt, dagegen ist der Ausdruck `funcType(3)` nicht erlaubt.

Ein Eintrag in unserer Symboltabelle besitzt deshalb außer dem Bezeichner und dessen Typisierung noch eine dritte Komponente: seine Klassifikation. Dort halten wir fest, um welche Art von Bezeichner es sich handelt, also z.B. um einen Funktionsbezeichner oder einen Bezeichner für einen allgemeinen Datentypen.

Mit Hilfe dieser Symboltabelle, die als Grundinformation für die Attributierung benötigt wird, lassen sich nun viele Komponenten der statischen Semantik abprüfen. Für NoName müssen unter anderem folgende Punkte überprüft werden:

Deklariertheit von Bezeichner: Jeder Bezeichner, der an beliebiger Stelle verwendet wird, muß im Programm auch deklariert sein.

Typüberprüfung: NoName besitzt ein strenges Typkonzept, d.h. für jeden Ausdruck ist dessen Typ genau festgelegt. Deshalb muß getestet werden, ob der tatsächliche Typ eines Ausdrucks mit dem erwarteten Typ, den dieser Ausdruck haben sollte, übereinstimmt. In diesem Bereich muß auch getestet werden, ob die Anzahl der Parameter bei Funktionsanwendungen stimmt.

Pattern–Matching: In NoName kann eine Funktion mittels Pattern–Matching definiert werden. Allerdings muß für jedes Element aus dem Wertebereich der Funktion *genau ein* Muster passen. Außerdem ist ein Pattern–Matching nur auf bestimmten Datentypen erlaubt. Diese Bedingungen müssen abgeprüft werden.

Zusätzliche Editorfunktionen

Der NoName–Editor soll ein möglichst komfortabler Editor sein, der den Benutzer in vielerlei Hinsicht beim Programmieren unterstützen soll. Ein wesentlicher Punkt dabei ist natürlich die sofortige Überprüfung auf Korrektheit des Programms bezüglich Syntax und statischer Semantik.

Doch lassen sich noch weitere Hilfsmittel denken, die den Benutzer beim Programmieren unterstützen. Einige solcher Hilfsmittel wurden in den NoName–Editor integriert.

Programmausführung: Wir haben für den NoName–Editor eine Funktion implementiert, die ein Programm zusammen mit einem zu berechnenden Ausdruck in die Programmiersprache Miranda übersetzt, dort den Ausdruck auswertet und das Resultat zurück wieder nach NoName übersetzt und auf dem Bildschirm ausgibt. Damit ist es möglich, Programme in der Editorumgebung zu testen.

Umwandlung von Pattern–Matching in IF–Bedingungen: In NoName gibt es zwei Möglichkeiten, verschiedenen Fälle einer Funktion zu definieren — mittels Pattern–Matching oder über IF–Bedingungen. Deshalb wurde eine Transformation implementiert, die eine Funktionsdefinition mittels Pattern–Matching in eine äquivalente Definition umwandelt, deren einzelne Fälle über IF–Bedingungen angegeben sind.

Umwandlung von tree transducern: Falls möglich, können die verschiedenen Arten von tree transducern in andere Arten von tree transducern umgewandelt werden.

Einfügen von Definitionsgleichungen: Funktionen eines tree–transducers werden immer über ein Pattern–Matching, das auf dem ersten Argument durchgeführt wird, definiert. Da der Typ dieses Arguments immer ein algebraischer Datentyp sein muß, ist genau festgelegt, welche Muster existieren müssen. Diese Definitionsgleichungen können auch automatisch eingefügt werden, der Benutzer muß nur noch die rechte Seite jeder Funktionsgleichung angeben.

Außer diesen schon implementierten Hilfsfunktionen sind noch viele andere möglich: Wir denken z.B. an einen Zirkularitätstest für tree transducer oder auch an die Umwandlung von IF–Funktionen in Funktionen mittels Pattern–Matching — diese Richtung ist wesentlich schwieriger als die umgekehrte und auch nicht immer durchführbar. Aber trotz aller Hilfsfunktionen wird die Hauptarbeit immer beim Programmierer selbst liegen: Er muß sich überlegen, welche Aufgabe ein Programm leisten soll und wie die Algorithmen aussehen müssen.

Entwicklung eines Modul–Informationssystems für den NoName–Editor

Erich Müller
Universität Ulm, Fakultät für Informatik,
Abt. Programmiermethodik und Compilerbau,
89069 Ulm, Germany
E-mail: erich@bach.informatik.uni-ulm.de

Hintergrund und Aufgabenstellung

In der funktionalen Programmiersprache NoName ist es möglich, mit der sogenannten *Import–Spezifikation* Konstrukte aus anderen NoName–Quelldateien zu importieren. Dazu müssen die Namen der Konstrukte sowie der Name der Quelldatei bekannt sein. In der Syntax von NoName kann eine derartige Import–Spezifikation z.B. folgendermaßen aussehen:

```
FROM mathematik.nnm IMPORT mult2;
```

Im Beispiel wird aus der Datei `mathematik.nnm` ein Konstrukt importiert, in welchem der Bezeichner `mult2` deklariert wird.

Die Möglichkeiten, welche die Import–Spezifikation zur Modularisierung und Wiederverwendung bestehener Konstrukte bietet, sind für kleinere Programmierprojekte sicherlich ausreichend. In größeren Projekten allerdings könnte die Übersicht leicht verlorengehen: „Wie heißt die Funktion? In welcher Datei wird sie deklariert?“ Um solche Fragen zu vermeiden, sollten die Programmierer durch ein System unterstützt werden, welches die Verwaltung der Konstrukte auf einfache Weise ermöglicht. An dieser Stelle setzt das NoName–Modul–Informationssystem, kurz NoMIS, an. In einer Datenbank sollen Sprachkonstrukte so abgelegt werden, daß sie wiedergefunden und wiederverwendet werden können. Zu den speicherbaren Konstrukten der Sprache NoName zählen die folgenden Spezifikationen:

- Funktions–Spezifikationen
- Datentyp–Spezifikationen
- algebraische Datentyp–Spezifikationen
- tree transducer–Spezifikationen

Import–Spezifikationen können nicht in der Datenbank abgelegt werden, weil bei der Verwendung einer Datenbank keine Import–Spezifikationen benötigt werden.

Bei der Realisierung des NoMIS sollte eine relationale Datenbank verwendet werden (in der aktuellen Implementierung wurde Ingres eingesetzt, [Pet92]). Die Bedienung des Systems sollte sich möglichst an der Bedienung des NoName–Editors orientieren, d.h. das System sollte in die grafische Benutzeroberfläche des SGen integriert werden.

SGen und Ingres — Zusammenführung der Softwarepakete

Sowohl der SGen als auch Ingres stellen ein Interface zu der Programmiersprache C zur Verfügung. Über dieses Interface wurde die Kopplung der beiden Systeme realisiert. Die Kommunikation erfolgt ausschließlich über gegenseitige Funktionsaufrufe; Pipes und ähnliches wurden nicht verwendet.

Die Implementierung der Datenbankanfragen erfolgt mittels Embedded SQL, einer Erweiterung von SQL um Konstrukte zur Bearbeitung von Datenbankanfragen in einer Host–Sprache (siehe [EN89] und [DS94]).

Zur Realisierung der Menüpunkte und Dialogfenster, welche für die Bedienung des NoMIS benötigt werden, werden die Interface–Funktionen des SGen verwendet. Damit sind allerdings die Möglichkeiten des Interfaces bereits ausgeschöpft. Weitergehende Funktionen, insbesondere zur Manipulation des Syntaxbaumes, sind im Interface nicht enthalten und mußten daher durch intensive Quellcode–Analyse ermittelt werden. Zu den wichtigsten Punkten hierbei zählen

- das Ermitteln der textuellen Repräsentation,
- das Einfügen einer Zeichenreihe in das editierte Programm und
- die Manipulation von Attributen.

Funktionen für diese und ähnliche Aufgaben wurden in einer editorunabhängigen „Toolbox“ zusammengefaßt, um auch in Zukunft damit arbeiten zu können.

Das verwendete ER–Modell

Das der Implementierung zugrundeliegende Entity–Relationship–Modell (siehe [Teo94]) ist in der Abbildung 2 zu sehen.

Das Entity *SK–Version* steht für die bereits angesprochenen Sprachkonstrukte. Die Suchkriterien (Topics) werden durch das Entity *Topic* dargestellt. Die n:m–Beziehung zwischen diesen beiden Entities besagt, daß jedem Sprachkonstrukt beliebig viele Topics zugeordnet werden können (und umgekehrt).

Da die Größe eines Sprachkonstruktes zumindest theoretisch nicht nach oben beschränkt ist, kann die Speicherung des NoName–Codes nicht in einem Attribut erfolgen. Stattdessen wird hierfür ein eigenes Entity verwendet. Der NoName–Code wird in Teile fester Länge aufgespaltet, wobei diese Teile fortlaufend numeriert werden, um sie später korrekt zusammensetzen zu können.

Die rekursive n:m–Beziehung mit der Bezeichnung *ruft auf* dient zur Speicherung von Abhängigkeiten der einzelnen Sprachkonstrukte untereinander. SQL kennt (noch) keine Rekursion, daher müssen die Daten für diese Beziehung in der Host–Sprache bereitgestellt werden.

Das Attribut *Classification* beschreibt den Typ des Sprachkonstruktes, die Versionsnummer wird in *Vers. Nr.* abgelegt. Die *Beschreibung* kann einen kurzen Kommentar zum Sprachkonstrukt enthalten. Der *Programmierer* entspricht dem Account–Name des Benutzers, der das Sprachkonstrukt in der Datenbank abgelegt hat. Das *Datum* ist identisch mit dem Systemdatum zum Zeitpunkt der Speicherung des Sprachkonstruktes.

Die Topics besitzen lediglich einen *Namen* und optional eine kurze *Beschreibung*.

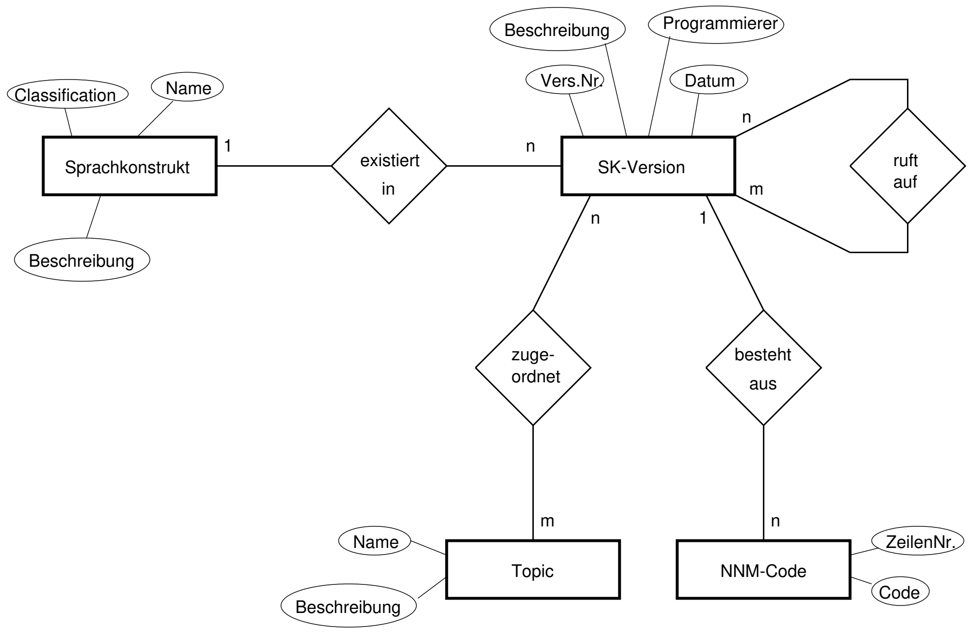

Abb. 2: ER–Modell für die Implementierung

Die Verwaltung der Suchkriterien

Die Verwaltung der Suchkriterien ist von der Verwaltung der Sprachkonstrukte strikt getrennt, damit die Benutzer bei der Speicherung eines Sprachkonstruktes nicht „mal eben" ein neues Topic generieren. Die Topic–Verwaltung gliedert sich in die folgenden drei Teilaufgaben:

1. Neues Topic anlegen.
2. Nicht benötigtes Topic löschen.
3. Anzeige aller gespeicherten Topics.

Zu beachten ist, daß ein Topic nur dann gelöscht werden kann, wenn kein Sprachkonstrukt in der Datenbank existiert, welches diesem Topic zugeordnet ist. Auf diese Weise wird verhindert, daß Sprachkonstrukte ohne zugeordnete Suchkriterien in der Datenbank vorhanden sind.

Die Verwaltung der Sprachkonstrukte

Die Verwaltung der Sprachkonstrukte gliedert sich in die folgenden drei Teilaufgaben:

- Speichern eines neuen Sprachkonstruktes in der Datenbank.
- Übertragen eines gespeicherten Sprachkonstruktes aus der Datenbank in den NoName–Editor.

- Löschen eines Sprachkonstruktes aus der Datenbank.

Diese Teilaufgaben sind im folgenden näher beschrieben.

Bei der **Speicherung** eines neuen Sprachkonstruktes wird das im Editor aktuell markierte Sprachkonstrukt in der Datenbank abgelegt. Dazu wird vorausgesetzt, daß sich diese Spezifikation ohne Fehler parsen läßt und auch bezüglich der statischen Semantik von NoName korrekt ist. Die Classification, also der „Typ" der Spezifikation, sowie das Datum und der Programmierer werden vom Programm automatisch bestimmt. Der Benutzer gibt lediglich den Namen, die Version und optional den Kommentar des neuen Sprachkonstruktes ein. Die Zuordnung zu einem oder mehreren Suchkriterien erfolgt einfach per Mausklick auf eine Liste aller vorhandenen Topics. Die Aufteilung des NoName–Codes auf mehrere Datensätze erfolgt bei Bedarf ebenfalls automatisch.

Eine Stärke des NoMIS ist die automatische Speicherung aller zusätzlich benötigten Sprachkonstrukte. Wenn der Benutzer z.B. die Funktion f in der Datenbank ablegen möchte und f eine andere Funktion g aufruft, dann wird neben der Funktion f auch die Funktion g in der Datenbank abgespeichert. Zur Sicherung eines eindeutigen Namens werden diese sogenannten „Hilfskonstrukte" fortlaufend numeriert. Bei dieser Vorgehensweise wird eine redundante Abspeicherung von Sprachkonstrukten weitgehend vermieden.

Der Vorteil dieser automatischen Speicherung benötigter Konstrukte zeigt sich beim **Übertragen** eines Sprachkonstruktes aus der Datenbank in den NoName–Editor. Der Benutzer gibt lediglich die vermutlich zugeordneten Suchkriterien an und wählt dann aus einer Liste aller passenden Sprachkonstrukte das gewünschte Konstrukt aus. Falls dieses Konstrukt andere Sprachkonstrukte aufruft, dann werden auch diese mit in den NoName–Editor übertragen. Im obigen Beispiel kann der Benutzer etwa die Funktion f importieren wollen, dann wird zusätzlich auch die Funktion g importiert. Somit kann der Benutzer die gespeicherten Sprachkonstrukte gleich einer „Black Box" verwenden.

Beim **Löschen** eines gespeicherten Sprachkonstruktes muß der Benutzer wiederum nur die vermutlich zugeordneten Suchkriterien angeben, um das gewünschte Konstrukt zu finden. Zu beachten ist allerdings, daß ein Benutzer nur Sprachkonstrukte löschen darf, die er selbst in der Datenbank abgelegt hat. Beim Löschen werden prinzipiell auch alle Hilfskonstrukte aus der Datenbank entfernt. Allerdings hat der Benutzer bei jedem Hilfskonstrukt die Möglichkeit, den Löschvorgang zu unterbinden.

Ausgabe des Datenbankinhaltes

Als eine kleine Hilfestellung für „Systemeinsteiger" ist die Möglichkeit gedacht, den Datenbankinhalt in LaTeX–Dateien auszugeben. Dabei werden die gespeicherten Daten in Listenform dargestellt. Es stehen die folgenden Listen zur Verfügung:

- Übersicht über alle gespeicherten Topics (Name und Beschreibung).
- Übersicht über alle gespeicherten Sprachkonstrukte (Name, Version, Beschreibung, Programmierer, Speicherungsdatum und Spezifikationstyp).

- Übersicht über die Zuordnungen von Topics zu Sprachkonstrukten (Name und Version der Sprachkonstrukte, jeweils mit den Namen der zugeordneten Topics).

Einige Erweiterungsmöglichkeiten

Das NoName–Modul–Informationssystem kann noch in einigen Punkten erweitert werden:

- Die Anbindung an andere Datenbanksysteme wäre wünschenswert.
- Ein Konzept zur Versionierung der gespeicherten Sprachkonstrukte wäre sinnvoll.
- Neben dem NoName–Quellcode könnte man auch an die Speicherung des bereits übersetzten Codes denken. Dazu sind allerdings einige offene Fragen zu beantworten, z.B:
  - Welche Informationen sind zusätzlich erforderlich, um den gespeicherten Code wiederverwenden zu können?
  - Wie sieht die Kommunikation zwischen Compiler und Datenbank bzw. Compiler und Editor aus?
  - Zu welchem Zeitpunkt erfolgt die Übernahme des übersetzten Codes von der Datenbank?
- Das Benutzerinterface des SGen ist nicht auf dem neuesten Stand. Allerdings hat die Firma GrammaTech für die nächste Version des SGen eine verbesserte Benutzeroberfläche in Aussicht gestellt, so daß sich damit dieses Problem eventuell erledigt.

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

Frank Houdek
Universität Ulm, Fakultät für Informatik
Abt. Programmiermethodik und Compilerbau
89069 Ulm, Germany
E–mail: fhoudek@bach.informatik.uni-ulm.de

Im diesem Text wird ein Übersetzer für eine Teilsprache von NoName (vgl. [EGH+94]), genannt NoName*, vorgestellt ([Hou95]).

Die zu übersetzende Sprache NoName*

Die Einschränkungen von NoName* gegenüber NoName sind im folgenden aufgeführt.

1. Ein NoName*–Programm darf keine Funktionen höherer Ordnung enthalten. Aus diesem Grund wird die Teilsprache, die übersetzt wird, auch first–level NoName genannt.
2. Ein NoName*–Programm darf als einzige tree transducer Art nur macro attributed tree transducer enthalten.
3. In einem NoName*–Programm dürfen keine Datentypsynonyme definiert sein, die Verwendung der Import–Anweisung ist nicht gestattet und die Spezifikation von Hilfsfunktionen mit Hilfe von `WHERE` ist nicht erlaubt.

Während die erste Bedingung eine wirkliche Einschränkung des Sprachumfangs bedeutet, lassen sich die anderen Bedingungen durch Transformationen des zu übersetzenden Programmes P erfüllen. So kann die Bedingung 2 erfüllt werden, indem jeder vorkommende tree transducer in P als macro attributed tree transducer dargestellt wird. Eine solche Transformation läßt sich sogar automatisch durchführen (vgl. [Ern95]).

Der Ausgangspunkt der Übersetzung ist ein korrektes NoName*–Programm. Ein Test auf Korrektheit wird also nicht durchgeführt. Im Gegensatz zu einem NoName–Programm gehört zu einem NoName*–Programm neben der Angabe der Datentypen und Funktionen auch noch ein Ausdruck, der mit Hilfe der angegebenen Funktionen berechnet werden soll. Dadurch ist es möglich, ein Maschinenprogramm zu erzeugen, das keine weiteren Eingaben oder Parameter zur Ausführung benötigt.

Die verwendete Zielmaschine

Die Zielmaschine der Übersetzung ist eine abstrakte Runtime–Stack Maschine, also eine Maschine, deren zentrale Komponente ein Stack ist (siehe z.B. [WM92, FGV91]). Weitere Komponenten der Zielmaschine sind eine Halde, auf der Objekte von unterschiedlicher Größe gespeichert werden, zwei Hilfskeller und einige Register. Ein Hilfskeller wird benötigt, um die Rücksprungadresse bei Unterprogrammaufrufen zu verwalten. Der andere Hilfskeller findet im Zuge der Parameterbestimmung bei der Auswertungsstrategie Call–by–Name Verwendung. Als Register stehen ein Programmzähler, ein Kellerrahmen–Zeiger und ein Rückgabewert–Register zur Verfügung.

Vorgehensweise bei der Übersetzung

Die Übersetzung erfolgt anhand der Struktur des zu übersetzenden NoName*–Programmes. Während des Übersetzungsvorgangs werden baumstrukturierte Adressen verwendet. Dadurch ist es möglich, die einzelnen Übersetzungsfunktionen weitgehend zu entkoppeln. Durch eine anschließende Linearisierung werden die baumstrukturierten Adressen dann durch fortlaufende Adressen ersetzt.

Der prinzipielle Aufbau des erzeugten Maschinenprogramms wird anhand des in Abbildung 10 angegebenen NoName*–Programms zur Berechnung der Fakultäts–Funktion erläutert.

```
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_PROGRAMM fakultaet.
```

Abb. 10: NoName*–Programm zur Berechnung von `fak(3)`.

Zuerst werden für die Anfrage Maschinenbefehle erzeugt. Dann folgt ein stop Befehl und schließlich die Maschinebefehle für die einzelnen Funktionen. Da die Abarbeitung eines jeden Maschinenprogramms bei der Programmadresse 1 beginnt, wird auf diese Weise nur die Anfrage ausgeführt, wobei dort in der Regel die verschiedenen Funktionen aufgerufen werden. In Abbildung 11 ist der Aufbau des erzeugten Maschinenprogramms für das Programm aus Abbildung 10 graphisch dargestellt.

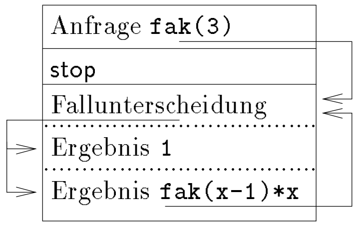

Abb. 11: Aufbau des erzeugten Maschinenprogramms für `fakultaet`.

Behandlung von Funktionsaufrufen

Für jeden Funktionsaufruf wird auf dem Runtime–Stack ein sog. Kellerrahmen (siehe Abbildung 12) aufgelegt und nach Abarbeitung des Funktionsaufrufs wieder entfernt (vgl. [WM92, FGV91]). Auf den jeweils obersten Kellerrahmen zeigt der Kellerrahmen–Zeiger. Untereinander sind die verschiedenen Kellerrahmen, wie auch in der Abbildung 12 angedeutet, verkettet.

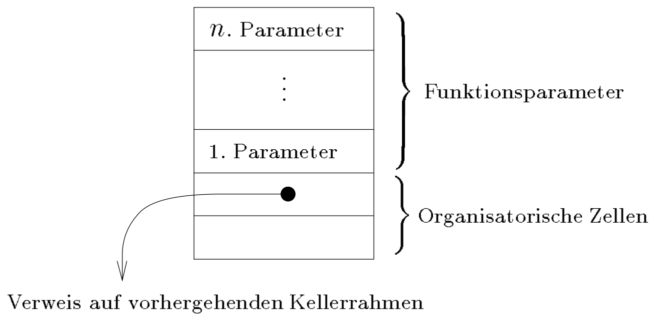

Abb. 12: Kellerrahmen.

Um während der Ausführung eines Funktionsaufrufs auf den n–ten Funktionsparameter zuzugreifen, muß nur ein Zugriff auf die $(n+2)$–te Zelle relativ zum Beginn des aktuellen Kellerrahmens erfolgen.

Soll die Auswertung einer Funktion nach der Strategie Call–by–Name erfolgen, so handelt es sich bei den Parametern nicht um Werte, sondern um sog. Funktionselemente. In einem Funktionselement sind eine Programmadresse *adr* und ein Verweis v gespeichert. Durch Anspringen der Routine an der Adresse *adr* unter Beachtung des Verweises v kann bei Bedarf der betreffende Parameterwert ermittelt werden.

Übersetzung von Fallunterscheidung mittels Pattern Matching

Bei Funktionen, die mittels Pattern Matching definiert sind, muß zur Laufzeit eine Fallunterscheidung durchgeführt werden, um die jeweils anzuwendende Funktionsgleichung zu bestimmen. Die sich hier ergebenden Problematik ist in der Funktionsdefinition in Abbildung 13 angedeutet.

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

Abb. 13: Fallunterscheidung mittels Pattern Matching.

Wie man leicht erkennen kann, sollte das Ergebnis eines Funktionsaufrufs der Form `test(...,0)` die Zahl 1 sein. Eine Auswertung des ersten Parameterwertes ist also nicht erforderlich. Im Falle des Funktionsaufrufs `test(1 div 0,0)` und der Auswertungsstrategie Call–by–Name würde die unnötige Auswertung dieses Parameters sogar die Berechnung abbrechen lassen.

Durch Anwendung eines in [Pey87] vorgestellten Verfahrens ist es möglich, aus den Funktionsgleichungen einer mittels Pattern Matching definierten Funktion die nötigen Fallunterscheidungen ermitteln zu lassen. Wir erhalten eine baumartige Struktur. In Abbildung 14 ist diese Struktur für die obige Funktion `test` dargestellt.

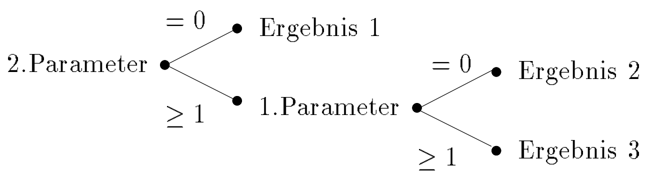

Abb. 14: Fallunterscheidungen beim Auswerten von `test`.

Übersetzung von Fallunterscheidung mittels IF

Die verschiedenen Fälle, die im Zuge einer Funktionsdefinition via IF angegeben sind, sollten überschneidungsfrei und erschöpfend sein. Leider kann das Einhalten dieser Forderung nicht überprüft werden, da sich hinter dieser Aufgabenstellung ein unentscheidbares Problem verbirgt. Um den Programmierer dennoch anzuhalten, diese Forderung einzuhalten, wurde in der Sprachbeschreibung [EGH$^+$94] festgelegt, daß die Überprüfung der Fälle nicht in der angegebenen Reihenfolge erfolgen sollte.

Bei der Übersetzung wurde diese Forderung nach beliebiger Auswertungsreihenfolge der Fälle so erfüllt, daß zum Übersetzungszeitpunkt eine beliebige Reihenfolge festgelegt wird. Während der einzelnen Programmausführungen bleibt diese Reihenfolge dann immer bestehen.

Übersetzung von algebraischen Datentypen und tree transducern

Baumstrukturen, die mit Hilfe von algebraischer Datentypen aufgebaut werden können, werden aufgrund ihrer Größe, die sich zur Laufzeit ändern kann, auf der Halde abgelegt. Auf dem Runtime–Stack werden nur Verweise auf diese Haldenobjekte gespeichert.

Funktionen eines marco attributed tree transducer verwenden bei der Berechnung Pfadangaben. Diese Pfadangaben werden in eigenen Haldenobjekten in Form von Kellern verwaltet. Durch Auflegen oder Entfernen eines neuen Eintrags auf einen solchen Keller wird die Pfadangabe verändert.

Implementierung

Für den Übersetzer und die abstrakte Maschine existiert eine Implementierung. Der Übersetzer ist der funktionalen Programmiersprache Gofer ([Jon91]) erstellt. Der Simulator für die abstrakte Maschine ist in C implementiert und erlaubt eine schrittweise Abarbeitung der erzeugten Maschinenprogramme.

Literatur

[CF82] B. Courcelle und P. Franchi–Zannettacci. *Attribute grammars and recursive program schemes.* Theoret. Comput. Sci., 17:163–191 and 235–257, 1982.

[DS94] P. Dadam und M. Scholl. *Blätter zur Vorlesung „Datenbanksysteme I: Grundlagen, Datenmodelle und Systemkonzepte“.* Universität Ulm, WS 1993/94.

[EGH+94] D. Ernst, D. Gluche, F. Houdek, A. Kühnemann, A. Mösle, E. Müller, S. Sablatnög, H. Vogler und E. Wieser. *NoName — Eine funktionale Sprache mit Unterstützung der primitiven Rekursion.* Interner Bericht, Universität Ulm, 1994.

[EN89] R. Elmasri und S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[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 241–256. 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.* J. Comput. Syst. Sci., 31:71–146, 1985.

[EV91] J. Engelfriet und H. Vogler. *Modular tree transducers.* Theoret. Comput. Sci., 78:267–303, 1991.

[FGV91] H. Faßbender, K. Gladitz und H. Vogler. *Compiler–based Implementation of syntax–directed functional programming.* Technischer Bericht91–02, Universität Ulm, 1991.

[Fül81] Z. Fülöp. *On attributed tree transducers.* Acta Cybernetica, 5:261–279, 1981.

[Hou95] F. Houdek. *Implementation von first–level NoName auf einer Runtime–Stack Maschine.* Diplomarbeit, Universität Ulm, 1995.

[Jon91] Mark P. Jones. *GOFER — An Introduction to Gofer*, 1991.

[Knu68] D.E. Knuth. *Semantics of context–free languages.* Math. Syst. Theory, 2:127–145, 1968.

[KV94] A. Kühnemann und H. Vogler. *Synthesized and inherited functions — a new computational model for syntax–directed semantics.* Acta Informatica, 31:431–478, 1994.

[May95] R. Mayer. *Testbericht über den NoName–Editor*, 1995.

[Pet92] D. Petković. *INGRES — Das relationale Datenbanksystem mit Knowledge–Base und Object–Base.* Addison Wesley, erste Auflage, 1992.

[Pey87] Simon L. Peyton Jones. *The implementation of functional programming languages.* Prentice–Hall, 1987.

[Rou70] W. C. Rounds. *Mappings and grammars on trees.* Math. System Theory, 4:257–287, 1970.

[Teo94] T. J. Teorey. *Database Modeling & Design.* Morgan Kaufmann Publishers, Inc., zweite Auflage, 1994.

[Tha70] J.W. Thatcher. *Generalized*^{2} *sequential machine maps.* Journal of Comput. Syst. Sci., 4:339–367, 1970.

[Vog93] H. Vogler. *Vorlesungsscript Attributgrammatiken.* Universität Ulm, WS 93/94, 1993.

[WM92] R. Wilhelm und D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung.* Springer Verlag, 1992.

Effiziente call-by-value Auswertungsstrategie primitiv-rekursiver Programmschemata

– Abstrakt –

Andrea Mößle[1]
Universität Ulm, Fakultät für Informatik
Abt. Programmiermethodik und Compilerbau
89069 Ulm, Germany
E-mail: andrea@bach.informatik.uni-ulm.de

und

Heiko Vogler
TU Dresden, Fakultät Informatik
Grundlagen der Programmierung
01062 Dresden, Germany
E-mail: vogler@inf.tu-dresden.de

Gerade im Gebiet der funktionalen Programmierung wurden viele Anstrengungen unternommen, um die Auswertung funktionaler Programme effizienter zu machen, deren Ausführungszeit im Vergleich zu prozeduralen Programmen immer noch länger ist. Eine Methode hierfür ist die Laufzeitverbesserung mittels Programmtransformationstechniken; ein wichtiges Stichwort hierbei ist die Rekursionselimination (siehe für einen Überblick Partsch 90 [11] und Pettorossi and Proietti 93 [13]).

Durch die Seiteneffektfreiheit läßt sich jedes funktionale Programm in kleine Teilstücke (Funktionsspezifikationen) zerlegen, die getrennt für sich betrachtet werden können. Betrachten wir z.B. die Fibonacci-Funktion in der funktionalen Programmiersprache No-Name [10]:

```
FUNC_SPEC
   fib : CARDINAL -> CARDINAL.

   fib(0) = 1
   fib(1) = 1
   fib(n+2) = fib(n) + fib(n+1);
END_FUNC_SPEC
```

Für die Berechnung des Ausdrucks $fib(5)$ müssen folgende Ausdrücke berechnet werden: einmal $fib(4)$, zweimal $fib(3)$, dreimal $fib(2)$, fünfmal $fib(1)$ und dreimal $fib(0)$. Unser Ziel ist es nun, eine solche mehrfache Auswertung gleicher Teilausdrücke zu vermeiden.

[1] Die Arbeit dieser Autorin wurde von der Deutschen Forschungsgemeinschaft (DFG) unterstützt.

Andere Arbeiten, die ebenfalls in diese Richtung gehen, basieren auf der Unfold/Fold Technik (Burstall and Darlington 77 [2]). Unglücklicherweise ist diese Technik in dem Sinn interaktiv, daß für gewisse Transformationsschritte die sinnvolle Mithilfe des Benutzers verlangt wird, um eine effizientere Version des Programms zu erhalten. Dabei verlangt diese sinnvolle Mithilfe viel Erfahrung und Kreativität. Viele der Arbeiten auf diesem Gebiet behandeln nur Programme über natürlichen Zahlen, da hier dann Nutzen aus speziellen Eigenschaften wie Kommutativität und Transitivität gewonnen werden kann (z.B. Bird 80 [1]). Eine weitere angenehme Eigenschaft, die gerne ausgenutzt wird, ist die Eindeutigkeit eines Vorgängers (in Partsch und Pepper 86 [12] wird allerdings auch Rekursionselimination im Bereich der freien Termalgebra betrachtet). Gerne werden Funktionen auch auf einstellige Funktionen beschränkt, bei denen dann keine Schachtelung von Aufrufen in Parameterpositionen auftreten können.

Wir betrachten primitiv-rekursive Programmschemata mit Parametern in der Art und Weise, wie sie von Engelfriet 80 [7], Courcelle und Franchi-Zannettacci 82 [6] eingeführt wurden (siehe ebenfalls Engelfriet und Vogler 85 [8], Engelfriet und Vogler 88 [9]). Für solche Schemata stellen wir eine Strategie vor, die vollautomatisch ohne die Unterstützung eines Benutzers diese Schemata in Schemata übersetzt, die garantiert effizienter ausgewertet werden können, falls die gegebenen Schemata nicht trivial waren und eine call-by-value Auswertung zugrundegelegt wird.

Weitere bemerkenswerte Arbeiten auf diesem Gebiet wurden von Chin ([3, 5, 4]) vorgestellt, die im Prinzip fast gleiche Ergebnisse liefern. Allerdings können dort keine Aussagen über eine garantierte Effizienzsteigerung getroffen werden.

Obige Funktionsdefinition ist nicht primitiv-rekursiv, kann jedoch mittels einer Hilfsfunktion in einer solchen Art und Weise umgeschrieben werden.

```
FUNC_SPEC
   fib : CARDINAL -> CARDINAL;
   h : CARDINAL -> CARDINAL.

   fib(0) = 1
   fib(x+1) = fib(x) + h(x);
   h(0) = 0
   h(x+1) = fib(x);
END_FUNC_SPEC
```

Bei Anwendung unserer Transformationsstrategie entstehen folgende rekursive Funktionsspezifikationen:

```
FUNC_SPEC
   fib : CARDINAL -> CARDINAL;
   (fib,h) : CARDINAL -> CARDINAL;
   ⟨fib,1,1⟩ : TREE -> CARDINAL;
   ⟨(fib,h),1,1⟩ : TREE -> CARDINAL.

   fib(0) = 1
```

```
    fib(x+1) = ⟨fib,1,1⟩((fib,h)(x));
    (fib,h)(0) = comb_2(1,0)
    (fib,h)(x+1) = ⟨(fib,h),1,1⟩((fib,h)(x));
    ⟨(fib,h),1,1⟩(comb_2(z_1,z_2)) = comb_2(z_1 + z_2,z_1);
    ⟨ fib,1,1⟩(comb_2(z_1,z_2)) = z_1 + z_2;
END_FUNC_SPEC
```

Dabei ist $TREE$ ein algebraischer Datentyp, der den zweistelligen Konstruktor *comb_2* enthält und als Teilbäume nur $CARDINAL$–Werte zuläßt. Um nun $fib(5)$ zu berechnen, müssen $(fib, h)(4), \ldots, (fib, h)(0)$ jeweils einmal berechnet werden und die zugehörigen Registerfunktionsaufrufe ebenfalls jeweils einmal berechnet werden. Unter Registerfunktionen verstehen wir die aus der Transformation entstehenden Funktionen der Form $\langle f, c, n\rangle$, wobei f ein Funktionsname, c ein Konstruktorname und n eine natürliche Zahl bezeichnet.

Die Transformation eines primitiv-rekursiven Programmschemas wird mittels drei einzelner Relationen realisiert, die in geeigneter Weise miteinander kombiniert werden. Jede dieser Einzelrelationen und die Kombination der Relationen sind semantikerhaltend und kanonisch.

Vorraussetzung für die Durchführung der angesprochenen Transformation ist das Vorliegen eines primitiv-rekursiven Programmschemas und die Möglichkeit, die Funktionen in einer call-by-value Strategie auszuwerten. Das Programmschema darf simultane Rekursion und geschachtelte Funktionsaufrufe in Parameterpositionen enthalten. Solche Programmschemata und die explizite Angabe der Auswertungsstrategie werden von der funktionalen Programmiersprache NoName durch spezielle Schlüsselworte syntaktisch unterstützt. Damit ist NoName sehr gut als Testsprache für unsere Überlegungen geeignet.

Literatur

[1] Bird, R. Tabulation techniques for recursive programs. *ACM Computing Surveys 12*, 4 (1980), 403–417.

[2] Burstall, R., and Darlington, J. A transformation system for developing recursive programs. *Journal of the ACM 24*, 1 (1977), 44–67.

[3] Chin, W. Towards an automated tupling strategy. In *3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Kopenhagen, 1993), ACM Press, pp. 119–132.

[4] Chin, W., and Hagiya, M. A transformation method for dynamic-sized tabulation. *Acta Informatica 32* (1995), 93–115.

[5] Chin, W., and Khoo, S. Tupling functions with multiple recursion parameters. *LNCS 724* (1993), 124–140.

[6] Courcelle, B., and Franchi-Zannettacci, P. Attribute grammars and recursive program schemes. *Theoretical Computer Science 17* (1982), 163–191 and 253–257.

[7] Engelfriet, J. Some open questions and recent results on tree transducers and tree languages. In *Formal language theory; prespectives and open problems*, R. Book, Ed. New York, Academic Press, 1980.

[8] Engelfriet, J., and Vogler, H. Macro tree transducers. *J. Comput. System Sci. 31*, 1 (august 1985), 71–176.

[9] Engelfriet, J., and Vogler, H. High level tree transducers and iterated pushdown tree transducers. *Acta Informatica 26* (1988), 131–192.

[10] Ernst, D., Gluche, D., Houdek, F., Kühnemann, A., Mössle, A., Müller, E., Sablatnög, S., Vogler, H., and Wieser, E. Noname – Eine funktionale Sprache mit Unterstützung der primitiven Rekursion. Interner Bericht der Universität Ulm, 1994.

[11] Partsch, H. *Specification and Transformation of Programs: a formal approach to software development.* Texts and monographs in computer science. Springer-Verlag, Berlin, 1990.

[12] Partsch, H., and Pepper, P. Program transformations expressed by algebraic type manipulations. *Technique et Science Informatiques 03* (1986), 197–212.

[13] Pettorossi, A., and Proietti, M. Rules and strategies for program transformation. In *Formal Program Development, IFIPTC2/WG 2.1 State-of-the-Art Report* (1993), B. Möller, H. Partsch, and S. Schumann, Eds., LNCS 755, pp. 263–304.