Erweiterung des KIEL-Systems

Diplomarbeit
vorgelegt von
Paul Mallach

Juli 2006

Betreuer: Prof. Dr. R. Berghammer
Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Zwischen 1996 und 2002 wurde das KIEL-System im Rahmen von zwei Diplomarbeiten [1,2] im Fach Informatik an der Christian-Albrechts-Universität zu Kiel entwickelt. Das Akronym KIEL ist die verkürzte Form von "Kiel Interactive Evaluation Laboratory". Mit Hilfe des Systems können Ausdrücke, also Terme über Rechenvorschriften, benutzergesteuert ausgewertet werden. Es wird seit Jahren erfolgreich bei der Ausbildung von Studenten eingesetzt.

Als Schnittstelle zwischen Anwender und System dient eine moderne grafische Oberfläche. Diese stellt den bearbeiteten Ausdruck als Baumstruktur dar. Zum einen kann der Anwender Schritt für Schritt durch Klicken einzelne Baumknoten auswerten lassen, zum anderen kann er vordefinierte Auswertungsstrategien benutzen, die automatisch ganze Teilbäume auswerten.

Das KIEL-System orientiert sich an der funktionalen Programmiersprache Standard ML, die ursprünglich Ende der 70er Jahre an der Universität von Edinburgh entwickelt wurde. An der Christian-Albrechts-Universität zu Kiel wird diese Programmiersprache seit Jahren im Informatik-Grundstudium zur Einführung der Grundlagen funktionaler Programmierung eingesetzt. In diesem Zusammenhang wird auch das KIEL-System benutzt, um es den Studenten zu ermöglichen, die Mechanismen bei der Auswertung ihrer Programme besser zu verstehen.

Im Rahmen dieser Diplomarbeit wurde das KIEL-System so erweitert, dass nun zusätzliche Sprachelemente von Standard ML unterstützt werden. Mit den neu eingeführten Produkttypen und lokalen Wertedeklarationen lassen sich fortgeschrittene Programmiertechniken verwenden und effizientere Programme erstellen.

Daneben wurden einige bestehende Fehler behoben, Verbesserungen an der grafischen Benutzeroberfläche vorgenommen und eine neue Funktion zur Analyse von Programmen eingeführt.

Die vorliegende Diplomarbeit ist wie folgt gegliedert:

Im anschließenden Kapitel 2 werden die Grundlagen des KIEL-Systems und der verwendeten Termersetzungssemantik skizziert. Kapitel 3 problematisiert die Aufgabenstellung für diese Diplomarbeit. Danach führt Kapitel 4 den Leser in die Bedienung der aktuellen

Version des KIEL-Systems ein. Kapitel 5 zeigt anhand von Beispielen die neuen Möglichkeiten auf. In Kapitel 6 wird die Implementierung des KIEL-Systems mit Fokus auf die Erweiterung des Programms um die neuen Funktionen beschrieben. Kapitel 7 fasst die Ergebnisse der Diplomarbeit zusammen und mündet in einem Ausblick auf denkbare zusätzlichen Erweiterungen für das KIEL-System.

2 Theoretische Grundlagen

Das KIEL-System ist eine Anwendung zur Visualisierung und interaktiven Auswertung von Ausdrücken. Dieses Kapitel erklärt die Grundlagen der unterstützten Sprache und beschreibt die vom KIEL-System verwendete operationelle Semantik zur Auswertung von Ausdrücken.

2.1 Standard ML

Das KIEL-System lehnt sich an die Programmiersprache ML an und unterstützt eine Teilmenge der von ML unterstützten Ausdrücke.

ML steht für "Meta-Language" und ist eine Familie von Programmiersprachen mit vorwiegend funktionalen Kontrollstrukturen. Gemeinsames Merkmal sind die klar definierte Semantik und das polymorphe Typsystem.

Die Grundlagen für ML wurden am Ende der 70er Jahre in Edinburgh gelegt. Dort entwickelten Robin Milner und andere eine Meta-Sprache, mit der sich Beweistechniken für ein automatisiertes Beweissystem namens LCF formulieren ließen. Bald bemerkten sie, dass sich diese Meta-Sprache auch als eigenständige Programmiersprache nutzen ließ.

Heutzutage sind Standard ML und CAML die wichtigsten Vertreter der ML-Sprachfamilie. Daneben existiert eine ganze Reihe von Sprachen, die vor allem zu Forschungszwecken dienen und an denen unterschiedliche Spracherweiterungen ausgetestet werden.

ML basiert auf einer typisierten Lambda-Algebra. Durch die Typen werden die Ausdrücke der Sprache klassifiziert. Mit Hilfe der statischen Zuordnung von Typen zu Ausdrücken kann der Compiler bereits bei der Übersetzung eine ganze Reihe von Fehlern erkennen.

An vielen Universitäten wird Standard ML in den Informatik-Vorlesungen zur Einführung in die funktionale Programmierung benutzt.

2.2 Signaturen und Terme

Die Grundlage der meisten funktionalen Programmiersprachen sind Terme. Der Begriff Term steht für Zeichenreihen, die in ihrem Aufbau festgelegten Regeln folgen. Alternativ

kann auch der Begriff Ausdruck verwendet werden. Beispiele für Terme über natürlichen Zahlen sind:

$$(1 + 3) * 4 \qquad (4 * 3) + 3$$

Intuitiv meint man zu erkennen, dass Zeichenreihen wie

$$3 + (1+ \qquad 3!3)$$

keine Terme sind, da sie nicht "wohlgeformt" erscheinen. Eine Definition der Begriffe Signatur und Term schafft hier Klarheit.

Die Symbole, die in einem Term verwendet werden können, werden durch eine Signatur $\Sigma = (S, K, F)$ mit den disjunkten Mengen S, K, F festgelegt. Dabei ist

- S die Menge der Sorten beziehungsweise Typen und nicht leer,
- K die Menge der Konstantensymbole und nicht leer; jedem $k \in K$ ist genau eine Sorte $s \in S$ zugeordnet,
- F die Menge der Funktionssymbole; jedem $f \in F$ sind genau die Argumentsorten $(a_1, a_2, ..., a_n) \in S^+$ und eine Resultatssorte $r \in S$ zugeordnet. Man sagt, dass f die Funktionalität $a_1 \times a_2 \times ... \times a_n \to r$ hat.

Eine Signatur für Terme, mit denen sich Mathematik auf Grundschulniveau ausdrücken lässt, ist zum Beispiel:

$$\begin{aligned} S =& \{nat\} \\ K =& \{0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, ...\} \\ F =& \{+ : nat \times nat \to nat,\ - : nat \times nat \to nat,\ * : nat \times nat \to nat, \\ & \quad div : nat \times nat \to nat,\ mod : nat \times nat \to nat\} \end{aligned}$$

In dieser Signatur wird eine Sorte mit dem Namen nat festgelegt. Die Konstanten dieser Sorte sind eine Menge von Symbolen, die die natürlichen Zahlen in ihrer üblichen Schreibweise widerspiegeln. Als Funktionssymbole sind die Symbole für die Grundrechenarten zugelassen.

Aufbauend auf einer Signatur können Terme induktiv definiert werden. Dafür sei zunächst $\Sigma = (S, K, F)$ eine Signatur und X eine Menge von Variablen, die zu den Mengen S, K, F disjunkt ist. Jedem $x \in X$ sei genau eine Sorte $s \in S$ zugeordnet.

Die Menge der $T^s_\Sigma(X)$ der Σ-Terme der Sorte $s \in S$ mit Variablen aus X ist induktiv definiert:

- Für alle Konstantensymbole k der Sorte s gilt $k \in T_\Sigma^s(X)$.
- Für alle Variablen x der Sorte s gilt $x \in T_\Sigma^s(X)$.
- Für alle Funktionssymbole f der Funktionalität $s_1 \times s_2 \times ... \times s_n \rightarrow s$ und Terme $t_i \in T_\Sigma^{s_i}(X)$ gilt $f(t_1, t_2, ..., t_n) \in T_\Sigma^s(X)$.

Wie in der Mathematik haben sich in vielen Programmiersprachen wegen der besseren Lesbarkeit Konventionen und vereinfachte Schreibweisen für Terme eingebürgert. So verwendet man häufig Infix-, Präfix- und Postfixnotationen statt der funktionalen Definition. Durch die Festlegung von Vorrangsregeln können Klammern eingespart werden, mit denen sonst die Reihenfolge der Auswertung bestimmt werden müsste.

2.3 Typen

In der Signatur von Standard ML sind die Sorten, Konstanten-, Operatoren- und Funktionssymbole festgelegt. Die Basis der vom KIEL-System unterstützten Typen bilden die Basissorten `unit`, `bool`, `int`, `real` und `string`. Auf ihnen bauen die verschiedenen Listentypen und die nutzerdefinierten Datentypen auf.

Einelementiger Bereich

Die Sorte `unit` besitzt nur eine einzige Konstante `()`.

Dem Anwender stehen folgende Operatoren und Basisfunktionen zur Verfügung:

=,<>	`unit` × `unit` → `bool`	Test auf Gleichheit und Ungleichheit

Wahrheitswerte

Mit der Sorte `bool` bildet das KIEL-System Wahrheitswerte ab. Dabei werden die beiden Konstanten `true` und `false` unterstützt.

Dem Anwender stehen folgende Operatoren und Basisfunktionen zur Verfügung:

`not`	`bool` → `bool`	Negation
`andalso`	`bool` × `bool` → `bool`	sequentielle Konjunktion
`orelse`	`bool` × `bool` → `bool`	sequentielle Disjunktion
=, <>	`bool` × `bool` → `bool`	Test auf Gleichheit und Ungleichheit

Ganze Zahlen

Die Sorte `int` repräsentiert im KIEL-System die Menge der ganzen Zahlen, die innerhalb des Intervalls $[-2^{30}, 2^{30} - 1]$ liegen.

Als Konstantensymbole dienen die entsprechenden Darstellungen (zur Basis 10) der Zahlen, wobei negativen Werten statt des üblichen - ein ~ vorangestellt wird. Nachstehend sind zwei Beispiele für die Darstellung von Konstantensymbole für ganze Zahlen angegeben:

- `2341`
- `~2341`

Dem Anwender stehen folgende Operatoren und Basisfunktionen zur Verfügung:

`~`	`int` → `int`	Negation
`+,-,*`	`int` × `int` → `int`	arithmetische Operationen
`abs`	`int` → `int`	absoluter Wert
`div`	`int` × `int` → `int`	ganzzahlige Division
`mod`	`int` × `int` → `int`	Rest einer ganzzahligen Division
`min`	`int` × `int` → `int`	kleinerer Wert
`max`	`int` × `int` → `int`	größerer Wert
`=,<>, <,>,<=, >=`	`int` × `int` → `bool`	Test auf Gleichheit, Ungleichheit, ...

Reelle Zahlen

Im KIEL-System werden die Maschinenzahlen im Intervall $[-10^{308}, 10^{308}]$ durch die Sorte `real` repräsentiert.

Die Konstanten werden mit Mantisse und Exponent geschrieben. Dabei hat die Mantisse eine oder mehrere durch einen Punkt abgetrennte Nachkommastellen. Danach folgen optional ein `e` oder `E` und der Exponent als ganze Zahl. Wie bei den ganzen Zahlen wird bei negativen Zahlen statt eines - ein ~ vorangestellt. Zur Verdeutlichung sollen hier zwei Beispiele für die Mantisse-Exponent-Darstellung von Konstanten angegeben werden:

- `2341.4E2`
- `~2341.2e~3`

Dem Anwender stehen für die reellen Zahlen folgende Operatoren und Basisfunktionen zur Verfügung:

`~`	`real` → `real`	Negation
`+,-,*,/`	`real` × `real` → `real`	arithmetische Operationen
`abs`	`real` → `real`	absoluter Wert
`sqrt`	`real` → `real`	Quadratwurzel
`sin, cos`	`real` → `real`	Sinus, Cosinus
`exp, ln`	`real` → `real`	e-Funktion und Logarithmus Naturalis
`=,<>, <,>,<=, >=`	`real` × `real` → `bool`	Test auf Gleichheit, Ungleichheit, ...
`real`	`int` → `real`	Konvertierung `int` nach `real`
`truncate`	`real` → `int`	Konvertierung `real` nach `int`

Zeichenreihen

Für Zeichenreihen stellt das KIEL-System die Sorte `string` bereit. Eine Konstante dieser Sorte ist eine Folge von Zeichen, die durch doppelte Anführungszeichen " begrenzt ist. Es werden keine Escape-Sequenzen unterstützt. Das KIEL-System kann also keine Zeichenreihen als Konstanten verarbeiten, die das Zeichen " enthalten. Zwei konkrete Beispiele für die Darstellung dieser Konstanten werden nachfolgend angeführt:

- `"christian-albrechts-universität zu kiel"`
- `"~2341.2E-3"`

Dem Anwender stehen folgende Operatoren und Basisfunktionen zur Verfügung:

`size`	`string` → `int`	Anzahl der Zeichen
`=,<>, <,>,<=, >=`	`string` × `string` → `bool`	lexikographische Vergleiche
`^`	`string` × `string` → `string`	Konkatenation
`ord`	`string` → `int`	ASCII-Code
`chr`	`int` → `string`	Zeichen mit ASCII-Code
`substring`	`string` × `int` × `int` → `string`	Teilzeichenreihe

Weitere Sorten in Standard ML

Von Standard ML selbst werden noch weitere Basissorten unterstützt, die im KIEL-System nicht implementiert wurden. Dazu gehören `word` für 16-Bit Ganzzahlen und `char` für einzelne Zeichen. Zusätzlich unterstützten viele Implementierungen von Standard ML noch die optionalen Typen `IntN.int`, `IntInf.int`, `WordN.word`, `RealN.real`, `WideString.string` und `WideChar.char`, wie in [6] beschrieben wird.

Zusammengesetzte Typen – Listen

In Standard ML ist es möglich, mit Hilfe von Typkonstruktoren wie `->`, `*` oder `list` neue Typen einzuführen. Das KIEL-System unterstützt eine Teilmenge dieser Typkonstruktoren. Durch Anwendung des Typkonstruktors `list` lassen sich Listentypen erstellen: Wenn `m` ein bereits eingeführter Typ ist, dann ist auch

`m list`

ein gültiger Typ. In der ersten Version des KIEL-Systems waren nur Listen mit Elementen der Basissorten `int`, `real`, `string`, `unit` und `bool` erlaubt. Nach der Erweiterung des KIEL-Systems in [2] werden auch Listen von beliebigen bereits eingeführten Typen unterstützt. Damit können zum Beispiel Matrizen von Maschinenzahlen als `real list list` repräsentiert werden.

Ein Ausdruck eines Listentyps hat die Form

$$[e_1, e_2, ..., e_k],$$

wobei die e_i jeweils Ausdrücke der entsprechenden Sorte sind und bei geschachtelten Listentypen auch wieder Listen sein können. Die leere Liste kann als [] oder `nil` geschrieben werden. Um mit Listen zu arbeiten, stehen dem Anwender folgende Operatoren und Basisfunktionen zur Verfügung:

`::`	`m` × `m list` → `m list`	Element am Anfang der Liste anfügen
`hd`	`m list` → `m`	erstes Element
`tl`	`m list` → `m list`	Liste ohne erstes Element
`null`	`m list` → `bool`	Test, ob Liste leer ist
`length`	`m list` → `int`	Anzahl der Elemente
`@`	`m` × `m list` → `m list`	Konkatenation
`rev`	`m list` → `m list`	Umkehrung
`=, <>`	`m list` × `m list` → `bool`	Tests auf Gleichheit und Ungleichheit

2.4 Einfache Ausdrücke und Rechenvorschriften

Bereits in der ersten Version des KIEL-Systems [1] wurden die grundlegenden Elemente von Standard ML implementiert, so dass bereits sinnvolle Programme geschrieben und ausgeführt werden konnten.

Im folgenden Abschnitt sollen diese Elemente vorgestellt werden.

2.4.1 Ausdrücke

In der ersten Version des KIEL-Systems werden Ausdrücke aus Konstanten, Basisoperatoren und -funktionen, Verzweigungen und Aufrufen bereits definierter Rechenvorschriften aufgebaut. Jeder Ausdruck hat einen festen Typ, der entweder vom Benutzer vorgegeben oder vom KIEL-System abgeleitet wurde.

Dabei gilt im KIEL-System, dass jede *Konstante* ein Ausdruck ist. Einige Beispiele für solche Ausdrücke sind:

- `14356`
- "`konstante`"
- `1.3e7`
- `nil`

Zusätzlich ist jede *Anwendung der Basisoperatoren und -funktionen* auf Ausdrücke der entsprechenden Sorte ein Ausdruck. Nachfolgend werden einige Beispiele für entsprechend zusammengesetzte Ausdrücke angegeben:

- "`teil eins`" ^ "`teil zwei`"
- $34.3 + 12.1E2$
- $5 = 1 + \texttt{size}(\text{"}\texttt{vier}\text{"})$
- $\texttt{real}(3) + 3.4 + \texttt{real}(\texttt{length}([]))$

Eine *Verzweigung* ist ebenfalls ein Ausdruck und hat die Form

$$\texttt{if b then } e_1 \texttt{ else } e_2.$$

Dabei ist `b` ein Ausdruck der Sorte `bool` und e_1 und e_2 sind Ausdrücke eines beliebigen aber identischen Typs. Der Typ des vollständigen Ausdrucks entspricht dem Typ von e_1 und e_2.

2.4.2 Rechenvorschriften

Rechenvorschriften werden in Standard ML über ihren frei wählbaren *Namen*, ihre *Funktionalität* und ihren *Rumpf* festgelegt. Die Funktionalität sind die Namen und Sorten der

formalen Parameter. Der Rumpf ist ein Ausdruck, der die Berechnung des Ergebnisses in Abhängigkeit von den Parametern festlegt.

Standard ML unterstützt polymorphe Rechenvorschriften und inferiert den Resultatstyp einer Rechenvorschrift. Das KIEL-System hingegen unterstützt keine Polymorphie und erfordert für jede Rechenvorschrift eine explizite Angabe des Resultatstyps.

Die Deklaration einer Rechenvorschrift `f` mit `n` formalen Parametern x_i vom Typ m_i, dem Ergebnistyp `m` und dem Rumpf `r` hat im KIEL-System folgende Form

$$\texttt{fun f}(x_1 : m_1, \ldots, x_n : m_n) : m = r;.$$

Dabei sind auch Rechenvorschriften ohne formale Parameter zulässig. Der Rumpf `r` ist ein Ausdruck im Sinne von 2.2 und 2.4 und muss den Typ `m` haben. In ihm dürfen zusätzlich die als formale Parameter deklarierten Variablenbezeichner und Aufrufe der Rechenvorschrift selbst verwendet werden.

Als konkretes Beispiel für die Deklaration einer Rechenvorschrift sei hier die Rechenvorschrift `b_i_s` (kurz für `buchstabe_in_string`) angegeben. Die nachfolgende Rechenvorschrift bestimmt, wie oft ein Buchstabe in einer Zeichenreihe enthalten ist:

```
fun b_i_s( buchstabe : string, s : string ) : int =
    if size(s) = 0
       then 0
       else b_i_s(buchstabe, substring(s,1,size(s)-1)) +
            (if substring(s,0,1) = buchstabe
               then 1
               else 0);
```

Der Aufruf einer Funktion oder einer Rechenvorschrift `f` mit `k` formalen Parametern ist ein Ausdruck und hat die Form

$$\texttt{f}(e_1, e_2, \ldots, e_k),$$

wobei alle e_i wieder selbst Ausdrücke sind, deren Typen den Typen der formalen Parameter der Funktion entsprechen. Der Typ dieses Ausdrucks ergibt sich aus der Deklaration der Rechenvorschrift.

Neben der einfachen Rekursion wie im Beispiel unterstützt das KIEL-System auch eine verschränkte Rekursion, bei der sich mindestens zwei Funktionen eventuell wechselseitig aufrufen. Hierzu muss ein System von Rechenvorschriften deklariert werden, das bei

einer Menge von n Rechenvorschriften folgende Form hat:

$$\begin{aligned}
&\texttt{fun}\ \mathtt{f_1}(\mathtt{x_{11}} : \mathtt{m_{11}}, \ldots, \mathtt{x_{1k_1}} : \mathtt{m_{1k_1}}) : \mathtt{m_1} = \mathtt{r_1} \\
&\texttt{and}\ \mathtt{f_2}(\mathtt{x_{21}} : \mathtt{m_{21}}, \ldots, \mathtt{x_{2k_2}} : \mathtt{m_{2k_2}}) : \mathtt{m_2} = \mathtt{r_2} \\
&\quad \ldots \\
&\texttt{and}\ \mathtt{f_n}(\mathtt{x_{n1}} : \mathtt{m_{n1}}, \ldots, \mathtt{x_{nk_2}} : \mathtt{m_{nk_n}}) : \mathtt{m_n} = \mathtt{r_n};
\end{aligned}$$

In diesem Fall dürfen in $\mathtt{r_1}$ bis $\mathtt{r_n}$ jeweils Aufrufe aller im System deklarierten Rechenvorschriften verwendet werden. Archetypisches Beispiel für die verschränkte Rekursion sind die Tests auf Gerade- und Ungeradesein, wie sie nachfolgend angegeben werden:

```
fun ist_gerade( i : int ) : bool =
    if i = 0
    then true
    else ist_ungerade(i-1)

and ist_ungerade( i : int ) : bool =
    if i = 0
    then false
    else ist_gerade(i-1);
```

Die verschränkte Rekursion ist hier an den gegenseitigen Aufrufen der beiden Rechenvorschriften zu erkennen.

2.5 Datentypen und Rechenvorschriften mit Mustererkennung

Im Rahmen einer zweiten Diplomarbeit [2] wurden umfangreiche Erweiterungen am KIEL-System vorgenommen. Durch die Ergänzung von zusammengesetzten Datenstrukturen und Rechenvorschriften mit Mustererkennung wurde es möglich, Algorithmen zu programmieren, die auf komplexen Datenstrukturen wie Matrizen oder Binärbäumen arbeiten.

2.5.1 datatype-Deklarationen

Mit Hilfe der neu eingeführten `datatype`-Deklaration kann der Anwender neue zusammengesetzte Typen erzeugen. Die Deklaration eines neuen Typs `m` hat im KIEL-System

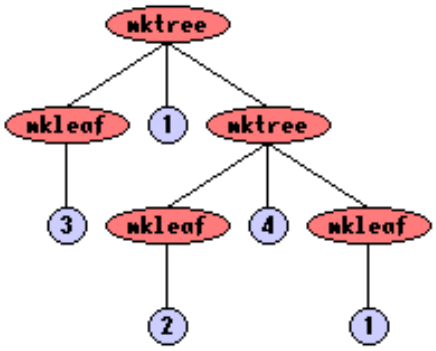


Abbildung 2.1: TBinTree

die nachfolgend angegebene Form:

$$\begin{aligned} \texttt{datatype m} = & \texttt{con}_1 \texttt{ of } \texttt{n}_{11} * \ldots * \texttt{n}_{1l_1} \mid \\ & \texttt{con}_2 \texttt{ of } \texttt{n}_{21} * \ldots * \texttt{n}_{2l_2} \mid \\ & \ldots \\ & \texttt{con}_k \texttt{ of } \texttt{n}_{k1} * \ldots * \texttt{n}_{kl_k}; \end{aligned}$$

Dabei sind $\texttt{con}_1$ bis $\texttt{con}_k$ die Bezeichner für die sogenannten Typkonstruktoren und alle aufgeführten $\texttt{n}_{ij}$ bereits eingeführte Typen oder der neue Typ $\texttt{m}$. Die Typkonstruktoren $\texttt{con}_i$ haben die Funktionalität

$$\texttt{n}_{i1} * \texttt{n}_{i2} * \ldots * \texttt{n}_{il_i} \rightarrow \texttt{m}.$$

Eine Instanz des Typs $\texttt{m}$ wird durch den Aufruf eines Typkonstruktors erzeugt, der die Form $\texttt{con}_i(\texttt{e}_{i1}, \ldots \texttt{e}_{l_i})$ hat. Hierbei sind die Argumente $\texttt{e}_{ij}$ jeweils Ausdrücke des Typs $\texttt{n}_{ij}$.

Als praktisches Beispiel soll ein Typ TBinTree für knotenmarkierte Bäume vorgestellt werden. Er wird durch folgende Deklaration eingeführt:

```
datatype TBinTree = mkleaf of int |
                    mktree of TBinTree * int * TBinTree;
```

Ein Baum kann nun durch Aufrufe der Typkonstruktoren `mkleaf` und `mktree` konstruiert werden. Um zum Beispiel den in Abb. 2.1 dargestellten Baum zu erzeugen, muss folgender Ausdruck verwendet werden:

```
mktree(mkleaf(3),1,mktree(mkleaf(2),4,mkleaf(1)))
```

Analog zu Systemen von Rechenvorschriften können auch Systeme von Datentypen deklariert werden. Diese werden wie folgt angegeben:

$$
\begin{aligned}
\texttt{datatype } \mathtt{m_1}=\mathtt{con_{11}} \texttt{ of } \mathtt{n_{111}} * \ldots * \mathtt{n_{11l_{11}}} \mid \\
\ldots \\
\mathtt{con_{1k}} \texttt{ of } \mathtt{n1_{1k1}} * \ldots * \mathtt{n_{1kl_{1k}}} \mid \\
\ldots \\
\texttt{and } \mathtt{m_p}=\mathtt{con_{p1}} \texttt{ of } \mathtt{n_{p11}} * \ldots * \mathtt{n_{p1l_{p1}}} \mid \\
\ldots \\
\mathtt{con_{pk}} \texttt{ of } \mathtt{n1_{pk1}} * \ldots * \mathtt{n_{pkl_{pk}}};
\end{aligned}
$$

Es gibt außer den Typkonstruktoren keine Operatoren auf diesen zusammengesetzten Datenstrukturen. Um auf die einzelnen Komponenten der Datenstruktur zugreifen zu können, muss der Anwender daher Rechenvorschriften mit Mustererkennung benutzen.

2.5.2 Rechenvorschriften mit Mustererkennung

Muster werden aus Konstanten, Variablenbezeichnern, Typkonstruktoren (inklusive der Standard-Typkonstruktoren :: und [...] für Listen) und dem Platzhaltersymbol _ aufgebaut. Aus diesen Grundbausteinen werden Muster ähnlich wie Ausdrücke zusammengesetzt. So sind die folgenden Beispiele gültige Muster:

`mktree(l, _, r)`	Typkonstruktor, 2 Variablen, 1 Platzhalter
`mkleaf(x)`	Typkonstruktor, 1 Variable
`mktree(mkleaf(l), 1, mkleaf(r))`	Typkonstruktoren, 2 Variablen, 1 Konstante
`erstes :: restliste`	Typkonstruktor (Liste), 2 Variablen
`nil`	Konstante (leere Liste)

Folgende Beispiele sind hingegen keine gültigen Muster:

`erstes ::`	Operator :: benötigt zweiten Operanden
`mktree(l, 3.4, r)`	Typkonstruktor erwartet int und nicht real

Ebenso wie bei Ausdrücken haben alle Muster im KIEL-System einen eindeutigen Typ. Eine genaue Definition aller erlaubten Muster ist in [2] nachzulesen.

Muster können in der Definition von Rechenvorschriften anstelle der Variablen in der Liste der formalen Parameter verwendet werden. Im erweiterten KIEL-System lassen sich für eine Rechenvorschrift mehrere Berechnungsvorschriften angegeben, die sich je-

weils durch die verwendeten Muster in der Liste der Parameter und den Rumpf unterscheiden.

Eine Rechenvorschrift mit Mustererkennung mit $\mathtt{n}$ Berechnungsvorschriften oder Fällen wird wie folgt deklariert:

$$\begin{aligned}&\mathtt{fun\ f(muster_{11} : t_1, \ldots, muster_{k1} : t_k)\ :\ t = r_1\ |}\\&\mathtt{\quad f(muster_{12} : t_1, \ldots, muster_{k2} : t_k)\ :\ t = r_2\ |}\\&\quad\ldots\\&\mathtt{\quad f(muster_{1n} : t_1, \ldots, muster_{kn} : t_k)\ :\ t = r_n;}\end{aligned}$$

Bei einem Aufruf der Rechenvorschrift in der Form $\mathtt{f(e_1, e_2, \ldots e_k)}$ wird im Rahmen der Auswertung die erste Berechnungsvorschrift $\mathtt{r_j}$ gewählt, für die alle Muster $\mathtt{muster_{ij}}$ jeweils zum Argument $\mathtt{e_i}$ passen. Dabei erfolgt die Überprüfung der Muster strikt in der Reihenfolge der Aufschreibung – sowie alle $\mathtt{k}$ Muster eines Falles auf die konkreten Argumente passen, wird die Suche abgebrochen und die passende Berechnungsvorschrift gewählt.

Anschaulich betrachtet passt ein Ausdruck zu einem Muster, wenn er entsprechend dem Muster aufgebaut ist: Wenn im Muster ein Typkonstruktor benutzt wird, dann muss der Ausdruck auch durch diesen Typkonstruktor aufgebaut sein. Wenn ein Muster eine Konstante enthält, muss auch der Ausdruck an der selben Stelle diese Konstante enthalten. Wenn das Muster das Symbol _ enthält, dann kann der Ausdruck an dieser Stelle einen beliebigen Teilausdruck enthalten. Wenn ein Muster eine Variable enthält, dann wird – wenn der gesamte Ausdruck zum Muster passt – die Variable an den entsprechenden Teil des Ausdrucks gebunden.

Um dies zu verdeutlichen, werden nachfolgend einige Beispiele für Muster und Ausdrücke angegeben und jeweils angemerkt, ob Muster und Ausdruck zusammen passen:

$\mathtt{mktree(l, \_, r)}$	passt zu	$\mathtt{mktree(mkleaf(2), 1, mkleaf(4))}$
$\mathtt{mktree(l, \_, r)}$	passt nicht zu	$\mathtt{mkleaf(2)}$
$\mathtt{mktree(mkleaf(l), 1, mkleaf(r))}$	passt zu	$\mathtt{mktree(mkleaf(2), 1, mkleaf(4))}$
$\mathtt{mktree(mkleaf(l), 1, mkleaf(r))}$	passt nicht zu	$\mathtt{mktree(mkleaf(2), 4, mkleaf(4))}$

Eine formale Definition der Überprüfung von Mustern und Ausdrücken kann man in [2] nachlesen.

2.5.3 Beispiel

Mit Hilfe dieser Rechenvorschriften mit Mustererkennung kann man auf die durch eine datatype-Deklaration eingeführten komplexen Datenstrukturen zugreifen. Dabei wird einfach für jeden Typkonstruktor ein entsprechendes Muster verwendet. Eine Rechenvorschrift, die die Höhe eines wie oben deklarierten kontenmarkierten Baumes berechnet, könnte so aussehen:

```
fun bintreeheight(mkleaf(_)          : TBinTree) : int =
    0
|   bintreeheight(mktree(l, _, r)    : TBinTree) : int =
    1+max(bintreeheight(l),bintreeheight(r));
```

Das erste Muster passt auf alle Bäume, die aus einem einzelnen Blatt bestehen, wobei die Markierung des Knotens irrelevant ist. Für diese wird eine Höhe von 0 zurückgegeben. Das zweite Muster passt auf alle Bäume, die an der Wurzel unabhängig von der Markierung einen linken und einen rechten Teilbaum haben. Falls dieses Muster passt, wird die Variable `l` an den linken Teilbaum und die Variable `r` an den rechten Teilbaum gebunden. Die Gesamthöhe des Baumes ergibt sich dann aus der maximalen Höhe der beiden Teilbäume plus `1`.

Mit diesen beiden Erweiterungen des KIEL-Systems ist die Ausdrucksfähigkeit des KIEL-Systems stark angestiegen. Mit ihrer Hilfe können nun Programme geschrieben werden, die auch auf komplexen Datenstrukturen operieren.

2.6 Produkttypen und lokale Wertedeklarationen

Neben den datatype-Deklarationen besteht in Standard ML die Möglichkeit, mit Hilfe des Typkonstruktors $*$ aus bereits definierten Typen neue, zusammengesetzte Typen zu erzeugen. Im Rahmen der vorliegenden Arbeit wurde das KIEL-System um die Unterstützung dieses Typkonstruktors erweitert.

Sind t_1 bis t_k bereits eingeführte Typen, dann ist

$$t_1 * ... * t_k$$

ein neuer so genannter Produkttyp. Ein Ausdruck dieses Typs wird auch Tupel genannt und hat im KIEL-System die Form

$$(e_1, e_2, ..., e_k).$$

Dabei sind die Komponenten e_i jeweils Ausdrücke des Typs t_i. Um auf diese einzelnen Komponenten eines Tupels zurückzugreifen, kann der Anwender eine Projektion oder eine lokale Wertedeklaration verwenden.

Projektionen sind eine Menge von Basisoperationen #1, #2, ... #n, die ein Tupel auf die entsprechende Komponente abbilden. $\#k(e_1, e_2, ..., e_n)$ ist also im KIEL-System ein Ausdruck vom Typ t_k und bildet das Tupel auf die k-te Komponente e_k ab.

Eine *lokale Wertedeklaration* für k Variablen v_1 bis v_k ist im erweiterten KIEL-System ein Ausdruck und hat die Form

$$\texttt{let val } (v_1 : t_1, \ldots, v_k : t_k) = \texttt{e in r end},$$

wobei die t_i jeweils die Typen der neu deklarierten Variablen und e ein Ausdruck vom Typ $t_1 * t_2 * ... * t_k$ ist. Der Rumpf r ist ein Ausdruck eines beliebigen Typs, in dem die neu deklarierten Variablen benutzt werden dürfen. Der Typ der lokalen Wertedeklaration ist der Typ des Rumpfes r.

Für die in 2.6 eingeführte Datenstruktur TBinTree wäre es sinnvoll, zu testen, ob ein Baum ein Suchbaum ist.

Ein Baum mit der Wurzelmarkierung x und den beiden Teilbäumen l und r ist genau dann ein Suchbaum, wenn l und r Suchbäume sind, alle Knotenmarkierungen aus l gleich oder kleiner als x sind und alle Knotenmarkierungen aus r gleich oder größer als x sind. Eine naive Implementierung dieser Funktion mit dem im vorangegangenen Abschnitt geschilderten Sprachumfang ist:

```
fun min_im_baum(mkleaf(x)          : TBinTree) :int =
    x
|   min_im_baum(mktree(l, x, r)    : TBinTree) : int =
    min(min_im_baum(l), min(min_im_baum(r),x));

fun max_im_baum(mkleaf(x)          : TBinTree) :int =
    x
|   max_im_baum(mktree(l, x, r)    : TBinTree) : int =
    max(max_im_baum(l), max(max_im_baum(r),x));

fun ist_suchbaum_naiv(mkleaf(_)         : TBinTree) : bool =
    true
|   ist_suchbaum_naiv(mktree(l, x, r)   : TBinTree) : bool =
    ist_suchbaum_naiv(l) andalso
    ist_suchbaum_naiv(r) andalso
    max_im_baum(l) <= x andalso
```

```
    x <= min_im_baum(r);
```

Wenn man die Auswertung dieses Programms genau betrachtet, stellt man fest, dass es bei dieser Implementierung ein Problem gibt:

Die Funktion `ist_suchbaum_naiv()` steigt rekursiv immer tiefer in den zu testenden Baum herab, bis sie bei den Blättern angekommen ist. In jedem dieser Schritte muss außerdem die maximale Knotenmarkierung im linken und die minimale Knotenmarkierung im rechten Teilbaum berechnet werden. Dies geschieht durch Aufrufe von zwei weiteren Funktionen, die selbst wieder rekursiv im Baum bis zu den Blättern hinabsteigen. Dabei werden die maximale und minimale Knotenmarkierung für viele Teilbäume unnötig mehrfach berechnet.

Mit Hilfe von Produkttypen und lokalen Wertedeklarationen kann man diese Probleme elegant umgehen und ein effizienteres Programm schreiben. Durch die Verwendung von Produkttypen können beim Suchbaumtest neben dem Ergebnis der Prüfung auch die minimale und maximale Knotenmarkierung im Baum zurückgegeben werden. Damit entfallen die vielfach unnötigen Aufrufe von `min_im_baum()` und `max_im_baum()`, weil alle Werte in der selben Rekursion berechnet werden.

Nachstehend wird eine entsprechende Version des Programms angegeben:

```
fun ist_sb(mkleaf(x)         : TBinTree) : bool * int * int =
    (true,x,x)
|   ist_sb(mktree(l, x, r)   : TBinTree) : bool * int * int =
    let val (lsb: bool, lmax:int, lmin:int) = ist_sb(l) in
        let val (rsb: bool, rmax:int, rmin:int) = ist_sb(r) in
           ( lsb andalso
             rsb andalso
             lmax <= x andalso x <= rmax
           , max(lmax,max(x,rmax))
           , min(lmin,min(x,rmin))
           )
       end
    end;
fun ist_suchbaum(t : TBinTree) : bool =
    let val (sb:bool, tmax: int, tmin: int) = ist_sb(t) in
        sb
    end;
```

Um die Steigerung der Effizienz zu zeigen, soll der Baum betrachtet werden, der durch den folgenden Ausdruck erzeugt wird (siehe Abb. 2.2 auf der nächsten Seite):

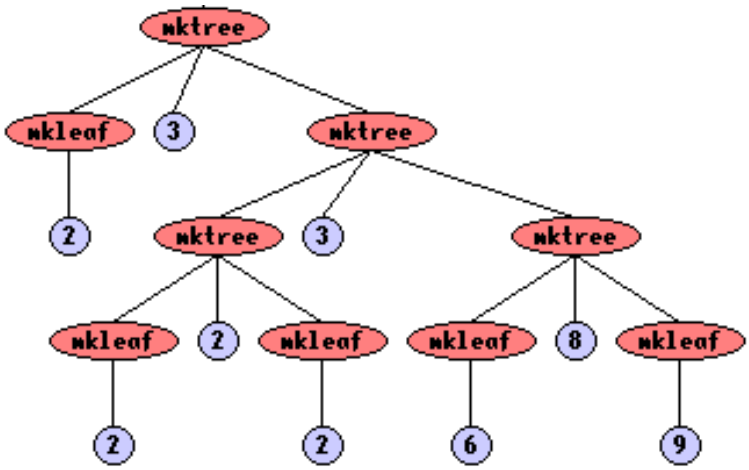


Abbildung 2.2: TBinTree – aber kein Suchbaum

```
mktree(
 mkleaf(2),
 3,
 mktree(
  mktree(mkleaf(2), 2, mkleaf(2)),
  3,
  mktree(mkleaf(6), 8, mkleaf(9))
 )
)
```

Wird die Suchbaumeigenschaft mit Hilfe der Rechenvorschrift `ist_suchbaum_naiv` geprüft, werden bei der Auswertung insgesamt 27 Aufrufe von Rechenvorschriften ausgeführt. Bei einem Test mit `ist_suchbaum` werden hingegen lediglich 10 Aufrufe von Rechenvorschriften ausgeführt. Je größer die Tiefe eines Suchbaumes, desto größer ist der Vorteil der effizienteren Implementierung mit Produkttypen und lokaler Wertedeklaration.

2.7 Auswertung

Ausdrücke werden im KIEL-System gemäß einer Termersetzungssemantik basierend auf einem Termersetzungssystem ausgewertet. Im Folgenden soll die Funktionsweise dieses Termersetzungssystem skizziert werden. Eine formale Definition kann man in [4] nachlesen.

Im KIEL-Systems besteht das Termersetzungssystem aus

- der Signatur $\Sigma = (S, K, F)$ des KIEL-Systems (wie in 2.3 beschrieben),
- einer Variablenmenge X, wobei jedem $x \in X$ ein Typ $s \in S$ aus der Signatur Σ zugewiesen ist,
- einer Menge R von Regeln der Form $l \rightarrow r$, wobei l und r Terme aus der Menge der Σ-Terme über den Variablen aus X sind, beide den gleichen Typ haben und alle Variablen aus r auch in l vorkommen.

Mit einem Termersetzungssystem kann ein Term t durch die Anwendung von Regeln aus R in einen anderen Term t' überführt werden.

Eine Regel $l \rightarrow r$ kann dabei genau dann auf einem Term t angewendet werden, wenn die linke Regelseite l ein Teilterm des Terms t ist. Durch die Anwendung ergibt sich ein neuer Term t', indem eben dieser Teilterm l in t durch die rechte Regelseite r ersetzt wird.

Falls die Regel $l \rightarrow r$ die Variablen $x_1, ..., x_n$ enthält, so steht sie stellvertretend für alle möglichen Regeln, die entstehen, indem man in $l \rightarrow r$ jedes x_i durch einen beliebigen Ausdruck e_i des gleichen Typs ersetzt. Im Sinne dieser nicht explizit aufgeführten Regeln kann die Regel $l \rightarrow r$ auf alle passenden Terme angewandt werden. Dies ist formal als Variablensubstitution in [4] definiert.

In dieser Arbeit wird die Überführung eines Terms t in einen Term t' durch Anwendung einer Termersetzungsregel in der Form $t \Rightarrow t'$ geschrieben. Um die Lesbarkeit zu verbessern, wird dabei der in t ersetzte Teilterm fett gedruckt.

Das KIEL-System wertet einen Ausdrucks t durch schrittweise Anwendung der Regeln des Termersetzungssystems aus. Dabei entsteht eine Sequenz $t_0, ..., t_n$ von Ausdrücken, die wie folgt gebildet werden:

- Setze $t_0 = t$ und
- berechne t_{i+1} aus t_i durch Anwendung einer passenden Regel aus R.

Wenn diese Sequenz endlich ist, also irgendwann keine Termersetzungsregeln mehr anwendbar sind, dann terminiert die Auswertung und der letzte Term t_n ist das Ergebnis. Wenn die Kette unendlich ist, dann ist das Ergebnis der Auswertung nicht definiert.

Die Regeln im Termersetzungssystem von KIEL lassen sich in zwei Gruppen unterteilen: die *Vereinfachungsregeln* und die *Expansionsregeln*.

2.7.1 Vereinfachung

Vereinfachungsregeln werden auf Basisoperatoren und -funktionen sowie Verzweigungen angewandt, wie sie in 2.2 vorgestellt wurden.

Bei der Anwendung einer Vereinfachungsregel wird der Aufruf des Basisoperators beziehungsweise der Basisfunktion durch den vom KIEL-System bestimmten Wert ersetzt.

Diese Termersetzungsregeln sind für die Basisoperatoren üblicherweise nicht explizit spezifiziert. Sie lassen sich aber sehr einfach aufzählen, da nur eine kleine Anzahl von Operatoren existiert und ihre Operanden Elemente endlicher Mengen sind.

Für den Operator + auf der Sorte `int` gilt zum Beispiel diese zwar sehr große, aber endliche Menge an Regeln:

$$\begin{aligned}
\texttt{0} + \texttt{0} &\rightarrow \texttt{0} \\
\texttt{0} + \texttt{1} &\rightarrow \texttt{1} \\
&\dots \\
\texttt{0} + \texttt{1073741823} &\rightarrow \texttt{1073741823} \\
\texttt{1} + \texttt{0} &\rightarrow \texttt{1} \\
\texttt{1} + \texttt{1} &\rightarrow \texttt{2} \\
\texttt{1} + \texttt{1073741822} &\rightarrow \texttt{1073741823} \\
\texttt{2} + \texttt{0} &\rightarrow \texttt{2} \\
&\dots
\end{aligned}$$

Entsprechend kann man auch für die andere Basisoperatoren und Sorten die Termersetzungsregeln aufschreiben. Für die Basisoperation `size` auf Konstanten der Sorte `string` gelten folgende Regeln:

$$\begin{aligned}
\texttt{size}(\texttt{""}) &\rightarrow \texttt{0} \\
\texttt{size}(\texttt{"a"}) &\rightarrow \texttt{1} \\
&\dots \\
\texttt{size}(\texttt{"Z"}) &\rightarrow \texttt{1} \\
\texttt{size}(\texttt{"aa"}) &\rightarrow \texttt{2} \\
\texttt{size}(\texttt{"ab"}) &\rightarrow \texttt{2} \\
&\dots
\end{aligned}$$

Einige einfache Beispiele für die Anwendungen der Termersetzungsregeln für die Basisoperatoren und -funktionen werden nachfolgend angegeben:

$$\begin{aligned}
2 + \mathbf{4} * \mathbf{6} &\Rightarrow 2 + 24 \\
\mathbf{size}(\text{"}\mathbf{ri}\text{"}) + 5 &\Rightarrow 2 + 5 \\
\mathbf{2} + \mathbf{5} &\Rightarrow 7
\end{aligned}$$

Neben diesen einfachen Termersetzungsregeln für die Basisoperatoren und -funktionen, werden auch für das Vereinfachen von Verzweigungen zwei Termersetzungsregeln wie folgt festgelegt:

$$\texttt{if true then e else f} \;\rightarrow\; \texttt{e}$$
$$\texttt{if false then e else f} \;\rightarrow\; \texttt{f}$$

Dabei sind `e` und `f` zwei Variablen eines beliebigen, aber gleichen Typs. Einige Beispiele für die Anwendung dieser beiden Termersetzungsregeln sind nachfolgend angegeben:

$$\textbf{if } \texttt{true} \textbf{ then } 3 \textbf{ else } 2 \;\Rightarrow\; 3$$
$$\textbf{if } \texttt{false} \textbf{ then } 4 \textbf{ else } 10 \;\Rightarrow\; 10$$

Mit Hilfe dieser Termersetzungsregeln lassen sich bereits einfache Ausdrücke ohne lokale Wertedeklarationen und Aufrufe von Rechenvorschriften auswerten.

2.7.2 Expansion von Rechenvorschriften

Um auch Ausdrücke mit Aufrufen von Rechenvorschriften auswerten zu können, müssen entsprechende Termersetzungsregeln für die sogenannte Expansion von Rechenvorschriften festgelegt werden.

2.7.2.1 Termersetzungsregeln

Bei einer Expansion des Aufrufs einer Rechenvorschrift wird dieser Aufruf durch den Rumpf der jeweiligen Rechenvorschrift ersetzt, wobei im Rumpf auftretende Variablensymbole durch die passenden Argumente des Aufrufs ersetzt werden.

Sei `f` eine Rechenvorschrift mit `n` formalen Parametern, die durch eine Deklaration in der Form

$$\texttt{fun f}(\texttt{x}_1 : \texttt{m}_1, ..., \texttt{x}_\texttt{n} : \texttt{m}_\texttt{n}) : \texttt{m} = \texttt{r};$$

eingeführt wurde. Die dazugehörige Termersetzungsregel ergibt sich direkt aus dieser Deklaration. Wenn $\texttt{x}_1, ..., \texttt{x}_\texttt{n}$ Variablen der Typen $\texttt{m}_1, ..., \texttt{m}_\texttt{n}$ sind, dann gilt im Termersetzungssystem die nachstehend angegebene Termersetzungsregel:

$$\texttt{f}(\texttt{x}_1, ..., \texttt{x}_\texttt{n}) \rightarrow \texttt{r}$$

Diese Regel lässt sich wegen der Variablensubstitution (siehe 2.7) auf alle Ausdrücke der Form $\texttt{f}(\texttt{x}_1, ..., \texttt{x}_\texttt{n})$ anwenden, in denen die $\texttt{x}_\texttt{i}$ beliebige Teilterme des entsprechenden

Typs sind. Ersetzt wird der Ausdruck durch den Rumpf r, wobei mittels der Variablensubstitution die Variablen $\mathtt{x_i}$ durch die entsprechenden Teilterme ersetzt werden.

Für eine Rechenvorschrift `f` mit Mustererkennung in der Form

$$\begin{aligned}\mathtt{fun\ f}(&\mathtt{muster_{11}} : \mathtt{t_1}, \ldots, \mathtt{muster_{k1}} : \mathtt{t_k}) : \mathtt{t} = \mathtt{r_1} \mid \\ &\mathtt{f}(\mathtt{muster_{12}} : \mathtt{t_1}, \ldots, \mathtt{muster_{k2}} : \mathtt{t_k}) : \mathtt{t} = \mathtt{r_2} \mid \\ &\ldots \\ &\mathtt{f}(\mathtt{muster_{1n}} : \mathtt{t_1}, \ldots, \mathtt{muster_{kn}} : \mathtt{t_k}) : \mathtt{t} = \mathtt{r_n};\end{aligned}$$

ergeben sich die Termersetzungsregeln analog. Wenn alle in den Mustern benutzten Variablen in der Variablenmenge `X` des Termersetzungssystems entsprechend eingeführt wurden, dann wird jeder Fall der Deklaration durch eine Regel berücksichtigt, die sich aus den Mustern und der Berechnungsvorschrift ergibt.

Für die oben angeführte Rechenvorschrift `f` müssen folgende Regeln im Termersetzungssystem gelten:

$$\begin{aligned}\mathtt{f}(\mathtt{muster_{11}}, \ldots, \mathtt{muster_{k1}}) &\rightarrow \mathtt{r_1} \\ \mathtt{f}(\mathtt{muster_{11}}, \ldots, \mathtt{muster_{k2}}) &\rightarrow \mathtt{r_2} \\ &\ldots \\ \mathtt{f}(\mathtt{muster_{1n}}, \ldots, \mathtt{muster_{kn}}) &\rightarrow \mathtt{r_n}\end{aligned}$$

Standard ML sieht vor, dass bei der Auswertung einer Rechenvorschrift mit Mustererkennung die verschiedenen Fälle in der Reihenfolge der Aufschreibung überprüft werden. Sobald in einem Fall alle Muster auf die Argumente passen, wird dieser Fall ausgewertet und die Suche bricht ab.

In einem herkömmlichen Termersetzungssystem gibt es keine Vorrangregeln für einzelne Regeln. Daher muss im Termersetzungssystems für das KIEL-System eine entsprechende Erweiterung festgelegt werden: Termersetzungsregeln für Rechenvorschriften mit Mustererkennung müssen in der beschriebenen Reihenfolge geprüft und angewandt werden.

Bei Systemen von Rechenvorschriften wird analog verfahren: Zu jeder einzelnen Rechenvorschrift im System gehören wie oben beschrieben entsprechend eine oder – im Fall eine Rechenvorschrift mit Mustererkennung – mehrere Regeln im Termersetzungssystem.

2.7.2.2 Beispiel: Expansion von Rechenvorschriften

Zur Veranschaulichung der Expansion soll die Rechenvorschrift `b_i_s` aus 2.4.2 auf Seite 16 dienen. Diese Rechenvorschrift bestimmt, wie oft ein Buchstabe in ein Zeichenreihe enthalten ist. Sie wird wie folgt deklariert:

```
fun b_i_s( buchstabe : string, s : string ) : int =
    if size(s) = 0
       then 0
       else b_i_s(buchstabe, substring(s,1,size(s)-1)) +
            (if substring(s,0,1) = buchstabe
               then 1
               else 0);
```

Es soll nun der nachfolgend angegebene Term betrachtet werden, in dem ein Aufruf dieser Rechenvorschrift vorkommt:

```
3 + b_i_s("a", "bb" ^ "da") * 8
```

Im Expansionsschritt wird der Aufruf der Rechenvorschrift durch den Rumpf der Rechenvorschrift ersetzt, wobei vorher in diesem das Variablensymbol `buchstabe` durch den Term `"a"` und das Variablensymbol `s` durch den Term `"bb"^"da"` ersetzt wurde:

```
3 + if size("bb" ^ "da") = 0
       then 0
       else b_i_s(
              "a",
              substring("bb" ^ "da",1,size("bb" ^ "da")-1)
            ) +
            (if substring("bb" ^ "da",0,1) = "a"
               then 1
               else 0)
* 8
```

Der entstandene Term kann dann durch die Anwendung anderer Termersetzungsregeln weiter ausgewertet werden.

2.7.3 Expansion von lokalen Wertedeklarationen

Im KIEL-System werden lokale Wertedeklarationen bei der Auswertung ähnlich wie Aufrufe von Rechenvorschriften behandelt. Weil sich eine lokalen Wertedeklaration auch als Anwendung einer anonymen Funktion (β-Reduktion) auffassen lässt, ist es möglich, die entsprechenden Termersetzungsregeln von den Termersetzungsregeln für β-Reduktionen aus der operationellen Semantik von Standard ML herzuleiten.

2.7.3.1 Termersetzungsregeln

Eine lokale Wertedeklaration mit n Variablen x_i vom Typ m_i als linker und einem Tupel $(v_1, ..., v_n) : m_1 * ... * m_n$ als rechter Seite der Deklaration sowie einem Rumpf r wird durch die Termersetzungsregel

$$\texttt{let val } (x_1 : m_1, ..., x_n : m_n) = (v_1, ..., v_n) \texttt{ in r end} \rightarrow \quad r'$$

expandiert. Dabei entsteht die rechte Seite der Regel r' aus dem Rumpf r der lokalen Wertedeklaration, indem für jedes x_i sämtliche Vorkommnisse in r durch die entsprechende Komponente v_i ersetzt werden.

Ist die rechte Seite der Deklaration hingegen noch nicht ausgewertet, also kein Tupel, sondern ein beliebiger Ausdruck f vom Typ $m_1 * ... * m_n$, wird die lokale Wertedeklaration durch die Termersetzungsregel:

$$\texttt{let val } (x_1 : m_1, ..., x_n : m_n) = f \texttt{ in r end} \quad \rightarrow \quad r'$$

expandiert. Hier entsteht die rechte Seite der Regel r' aus dem Rumpf r der lokalen Wertedeklaration, indem für jedes x_i sämtliche Vorkommnisse in r durch die Projektion von f auf die i-te Komponente ersetzt werden.

2.7.3.2 β-Reduktion

Es mag nicht gleich einleuchten, warum lokale Wertedeklarationen mit ähnlichen Regeln behandelt werden wie Rechenvorschriften. Der Hintergrund ist jedoch einfach:

Eine lokale Wertedeklaration lässt sich immer auch als β-Reduktion, also als Anwendung einer passenden λ-Abstraktion, auffassen. Für die Auswertung dieser β-Reduktion gelten dann ähnliche Termersetzungsregeln wie bei der Auswertung von Aufrufen herkömmlicher Rechenvorschriften.

Zur Veranschaulichung soll die lokale Wertedeklaration

$$\texttt{let val } (x_1 : m_1, ..., x_n : m_m) = (v_1, ..., v_n) \texttt{ in r end}$$

betrachtet werden. Diese kann man auch als Anwendung der entsprechenden λ-Abstraktion $\texttt{fn}(x_1 : m_1, ..., x_n : m_n) => r$ (hier in ML-Notation angegeben: x_1 bis x_n werden auf r abgebildet) auf die Argumente $v_1, ..., v_n$ interpretieren.

Das KIEL-System kennt allerdings keine Funktionstypen und damit auch keine λ-Abstraktionen. Ihre operationelle Semantik entspricht in Standard ML der in 2.7.2 vorgestellten Termersetzungssemantik für die Auswertung von Aufrufen von Rechenvorschriften.

Nach diesen Termersetzungsregeln wird eine solche Anwendung einer λ-Abstraktion ausgewertet, indem sie durch einen Term $\mathtt{r}'$ ersetzt wird, der aus dem Rumpf $\mathtt{r}$ der λ-Abstraktion ensteht, indem alle Vorkommnisse der Variablen $\mathtt{x}_1, ..., \mathtt{x}_n$ aus den formalen Parameter durch die jeweiligen Argumente des Aufrufs (hier $\mathtt{v}_1, ..., \mathtt{v}_n$) ersetzt werden.

Aus dieser Termersetzungsregel folgen direkt die in 2.7.3.1 angegebenen Regeln für lokale Wertedeklarationen.

2.7.3.3 Beispiel: Einfache lokale Wertedeklaration

Am Beispiel einer Funktion `Max(s : int list) : int` mit einer lokalen Wertedeklaration soll aufgezeigt werden, wie eine lokale Wertedeklaration in einen Aufruf einer äquivalenten Rechenvorschrift umgeschrieben werden kann. Die Rechenvorschrift `Max` sei wie nachfolgend angegeben deklariert:

```
fun Max(s : int list) : int =
    if length(s)=1 then hd(s)
                   else let val (y:int) = Max(tl(s))
                        in  max(hd(s),y)
                        end;
```

Die lokale Wertedeklaration in dieser Rechenvorschrift kann man auch wie folgt mit Hilfe einer λ-Abstraktion ausdrücken, die auf den rechten Teil der Deklaration angewandt wird:

$$(\mathtt{fn}(\mathtt{y} : \mathtt{int}) => \mathtt{max}(\mathtt{hd}(\mathtt{s}), \mathtt{y}))(\mathtt{Max}(\mathtt{tl}(\mathtt{s}))$$

In Standard ML werden λ-Abstraktionen unterstützt und man könnte obige Rechenvorschrift also auch einfach ohne lokale Wertedeklaration formulieren:

```
fun Max(s : int list) : int =
    if length(s)=1 then hd(s)
                   else (fn(y:int) => max(hd(s),y))(Max(tl(s)));
```

Weil das KIEL-System keine λ-Abstraktionen kennt, muss man sich mit einer zusätzlichen Hilfsrechenvorschrift `F(y:int) : int` außerhalb der Rechenvorschrift `Max` behelfen. Damit die Variable `s` im Rumpf von `F` verwendet werden kann, muss allerdings der Funktion `F` noch ein weiterer Parameter hinzugefügt werden. So kann der in der Rechenvorschrift `Max` gültige Wert für `s` an die Hilfsrechenvorschrift `F` übergeben werden.

Insgesamt ergibt sich also ein System von zwei Rechenvorschriften:

```
fun F(y:int, s: int list) : int =
    max(hd(s),y)
and Max(s : int list) : int =
    if length(s)=1 then hd(s)
                   else F(Max(tl(s),s)
```

2.7.3.4 Beispiel: Tupel und mehrere Variablen

Betrachten wir nun eine Rechenvorschrift `rmv_biggest`, bei der in einer lokalen Wertedeklaration mehrere Variablen eingeführt werden. Jeder Variable soll eine Komponente eines Tupels zugewiesen werden. Die Rechenvorschrift ist wie folgt festgelegt:

```
fun rmv_biggest(s : int list) : int list * int =
    if length(s)=1 then
       (s,hd(s))
    else
      let val (snmx : int list, mx : int) = rmv_biggest(tl(s)) in
         if hd(s) > mx then (tl(s),hd(s))
                       else (hd(s) :: snmx, mx)
      end;
```

`rmv_biggest` bestimmt aus einer übergebenen Liste von ganzen Zahlen zum einen das größte Element und zum anderen die Liste ohne dieses größte Element.

Wir betrachten die lokale Wertedeklaration im Rumpf der Rechenvorschrift. Wenn sie vom KIEL-System ausgewertet werden soll, können zwei Fälle auftreten: Entweder wurde die rechte Seite der Wertedeklaration bereits ausgewertet oder noch nicht.

Falls die rechte Seite der Wertedeklaration zum Beispiel bereits zu $([3,4,5],9)$ ausgewertet wurde, muss ein Ausdruck in der Form

$$\begin{array}{ll}\texttt{let} & \texttt{val}\,(\texttt{snmx} : \texttt{int list}, \texttt{mx} : \texttt{int}) = ([3,4,5],9)\,\texttt{in} \\ & \texttt{if}\,\texttt{hd}(\texttt{s}) > \texttt{mx}\,\texttt{then}\,\ldots\,\texttt{else}\,\ldots \\ & \texttt{end}\end{array}$$

ausgewertet werden. Dieser Ausdruck kann einfach mit Hilfe einer λ-Abstraktion als eine β-Reduktion

$$\begin{array}{ll}(\texttt{fn} & (\texttt{snmx} : \texttt{int list}, \texttt{mx} : \texttt{int}) => \\ & \texttt{if}\,\texttt{hd}(\texttt{s}) > \texttt{mx}\,\texttt{then}\,\ldots\,\texttt{else}\,\ldots\,)\quad ([3,4,5],9)\end{array}$$

betrachtet werden. Im Sinne dieser Anwendung einer λ-Abstraktion wird die lokale Wertedeklaration nach der in 2.7.2 eingeführten Termersetzungsregel wie folgt expandiert:

$$\texttt{if hd(s)} > 9 \texttt{ then (tl(s), hd(s)) else (hd(s) :: [3, 4, 5], 9)}$$

Im zweiten Fall ist es komplizierter. Wenn nämlich die rechte Seite der Wertedeklaration noch nicht ausgewertet wurde, dann hat der Ausdruck die Form

$$\begin{array}{ll} \texttt{let} & \texttt{val (snmx : int list, mx : int)} = \texttt{rmv\_biggest(tl(s))} \texttt{ in} \\ & \texttt{if hd(s)} > \texttt{mx then ... else ...} \\ & \texttt{end.} \end{array}$$

Wie gehabt, ergibt sich daraus die Anwendung der nachfolgend angegebenen λ-Abstraktion:

$$\begin{array}{ll} \texttt{(fn} & \texttt{(snmx : int list, mx : int)} => \\ & \texttt{if hd(s)} > \texttt{mx then ... else ...)} \quad \texttt{(rmv\_biggest(tl(s))} \end{array}$$

Bei der Expansion dieser β-Reduktion tritt allerdings ein Problem auf: Die Vorkommnisse der Variablen `snmx` und `mx` im Rumpf der λ-Abstraktion müssen durch die Komponenten des Arguments `rmv_biggest(tl(s))` ersetzt werden. Weil dieser Ausdruck noch nicht ausgewertet ist, existieren aber noch keine einzelnen Komponenten, die den Variablen zugewiesen werden könnten.

Daher führt das KIEL-System hier eine Projektion $\#\texttt{k} : \texttt{m}_1 * ... * \texttt{m}_k * ... * \texttt{m}_n \rightarrow \texttt{m}_k$ ein, die ein Tupel auf seine `k`-te Komponente abbildet. Mit Hilfe der Projektion kann dann die Expansion erfolgen:

Die Variable `snmx` wird an den Ausdruck $\#1$`(rmv_biggest(tl(s))` und die Variable `mx` an den Ausdruck $\#2$`(rmv_biggest(tl(s)))` gebunden. Somit kann den Variablen die passende Komponente zugewiesen werden, ohne dass die rechte Seite der Deklaration `rmv_biggest(tl(s))` vorher ausgewertet sein muss.

Nach der Expansion ergibt sich dann folgender Term:

$$\begin{array}{lll} \texttt{if hd(s)} > \#2\texttt{(rmv\_biggest(tl(s))} & \texttt{then} & \texttt{(tl(s), hd(s))} \\ & \texttt{else} & \texttt{(hd(s)} :: \#1\texttt{(rmv\_biggest(tl(s))),} \\ & & \#2\texttt{(rmv\_biggest(tl(s))))} \end{array}$$

Wie man beobachten kann, entspricht dies genau einer Anwendung der in 2.7.3.1 angegeben Termersetzungsregel.

Mit Hilfe dieser Projektionen kann die Rechenvorschrift auch – wie im vorherigen Beispiel – ohne die lokale Wertedeklaration formuliert werden. Dabei wird analog zum dortigen Vorgehen eine Hilfsrechenvorschrift anstelle der λ-Abstraktion eingeführt. So ergibt sich die folgende Implementierung:

```
fun F(s_nmx: int list, mx: int) : int =
    if hd(s) > mx then (tl(s),hd(s))
                  else (hd(s) :: snmx, mx)
and fun remove_biggest(s : int list) : int list * int =
    if length(s)=1 then
       (s,hd(s))
    else
       F(#1(remove_biggest(tl(s))),#2(remove_biggest(tl(s))))
    end;
```

2.7.4 Auswertungsstrategien

Ein Term t wird im Rahmen einer Termersetzungssemantik schrittweise über eine Sequenz von Termen t_0 bis t_n ausgewertet, wobei der Term t_{i+1} jeweils aus t_i durch Anwendung einer Termersetzungsregel entsteht.

Bei der Auswertung eines Terms gibt es oft mehrere Regeln, die in einem Schritt anwendbar sind. Betrachtet man zum Beispiel die einfache Rechenvorschrift

$$\texttt{fun G(x : int)} = \texttt{x} * 2$$

und den Ausdruck

$$\texttt{G(G}(3 + 4)),$$

dann könnten im nächsten Schritt mehrere Regeln aus dem Termersetzungssystem des KIEL-Systems angewandt werden:

$$\begin{aligned}
\texttt{G(G}(\mathbf{3} + \mathbf{4})) &\Rightarrow^{(+)} \texttt{G(G}(7)) \\
\texttt{G}(\mathbf{G}(3 + 4)) &\Rightarrow^{(RV)} \texttt{G}((3 + 4) * 2) \\
\mathbf{G}(\texttt{G}(3 + 4)) &\Rightarrow^{(RV)} \texttt{G}(3 + 4) * 2
\end{aligned}$$

Eine allgemeine Termersetzungssemantik macht keine Vorgaben darüber, welche dieser Regel gewählt werden muss. In der Praxis legen sogenannte *Auswertungsstrategien* fest, welche Regel an welcher Stelle im Term angewendet wird. Die Stellen in einem Term, an denen Termersetzungsregeln angewandt werden können, werden auch *Reducible Expressions* (Redex) genannt.

Die von den meisten Programmiersprachen verwendete Auswertungsstrategie ist die sogenannte *Leftmost-Innermost-Substitution*.

Dabei wird von allen inneren Redeces (in denen also kein weiterer Redex vorkommt), der im aufgeschriebenen Term am weitesten links gelegene Redex ersetzt. Falls nicht alle Teilterme dieses Redex definiert sind, wird die Auswertung abgebrochen und das Resultat der gesamten Auswertung ist undefiniert. Die Leftmost-Innermost-Substitution entspricht der bekannten Call-by-Value Semantik.

Daneben existieren auch andere Auswertungsstrategien, wobei vor allem die *Leftmost-Outermost-Substitution* interessant ist. Hierbei wird der Redex ersetzt, der im aufgeschriebenen Term am weitesten links gelegen ist – dies entspricht einer Call-by-Name Semantik.

Bei der *vollständigen Substitution* werden sämtliche Redeces auf einen Schlag ersetzt.

Das KIEL-System kann Ausdrücke wahlweise nach einer der drei Auswertungsstrategien auswerten. Weil aber bei der vollständigen Substitution sehr schnell riesige Ausdrücke entstehen, muss diese Auswertungsstrategie beim Start des KIEL-System explizit über die Kommandozeile aktiviert werden.

Die Unterschiede zwischen den Auswertungsstrategien sollen anhand folgender einfacher Funktion `F` demonstriert werden:

```
fun F(x:int) :int =
    if x = 1 then 1
             else F(F(x-1));
```

Wenn diese Funktion mit einem Argument > 0 aufgerufen wird, dann zieht der Aufruf zwei rekursive Aufrufe nach sich.

Wir beginnen mit einem Ausdruck `F(2)` und betrachten in Abb. 2.3 auf der nächsten Seite zunächst eine Auswertung des Ausdrucks nach der Leftmost-Innermost Strategie.

In jedem Schritt wird genau die Termersetzungsregel angewandt, die sich auf den Redex bezieht, der von aller inneren Redeces möglicht weit links liegt.

Die Auswertung desselben Ausdrucks nach einer Leftmost-Outermost Strategie ist in der Abb. 2.4 auf Seite 37 dargestellt. In jedem Schritt wird die passende Termersetzungsregel angewandt, die sich auf den am weitesten links gelegenen Redex bezieht.

Man erkennt auf einen Blick, dass die Auswertung bei den beiden Strategien unterschiedlich verläuft, das Ergebnis aber gleich ist.

In diesem Fall war die Auswertung mit Hilfe der Leftmost-Innermost-Substitution effizienter: Es werden weniger Schritte als bei der Leftmost-Outermost-Substitution gebraucht.

$$
\begin{aligned}
\mathbf{F}(2) &\Rightarrow \texttt{if } \mathbf{2} = \mathbf{1} \texttt{ then } 1 \\
&\qquad\qquad \texttt{else } \mathsf{F}(\mathsf{F}(2-1)) \\
&\Rightarrow \textbf{if } \texttt{false } \textbf{then } 1 \\
&\qquad\qquad \textbf{else } \texttt{else } \mathsf{F}(\mathsf{F}(2-1)) \\
&\Rightarrow \mathsf{F}(\mathsf{F}(\mathbf{2}-\mathbf{1})) \\
&\Rightarrow \mathsf{F}(\mathbf{F}(1)) \\
&\Rightarrow \mathsf{F}(\texttt{if } \mathbf{1} = \mathbf{1} \texttt{ then } 1 \\
&\qquad\qquad \texttt{else } \mathsf{F}(\mathsf{F}(1-1))) \\
&\Rightarrow \mathsf{F}(\textbf{if } \texttt{true } \textbf{then } 1 \\
&\qquad\qquad \textbf{else } \mathsf{F}(\mathsf{F}(1-1))) \\
&\Rightarrow \mathbf{F}(1) \\
&\Rightarrow \texttt{if } \mathbf{1} = \mathbf{1} \texttt{ then } 1 \\
&\qquad\qquad \texttt{else } \mathsf{F}(\mathsf{F}(1-1))) \\
&\Rightarrow \textbf{if } \texttt{true } \textbf{then } 1 \\
&\qquad\qquad \textbf{else } \mathsf{F}(\mathsf{F}(1-1))) \\
&\Rightarrow 1
\end{aligned}
$$

Abbildung 2.3: Auswertung von `F(2)` nach der Leftmost-Innermost Strategie

Das ist aber nicht zu verallgemeinern, bei anderen Rechenvorschriften kann die Auswertung mit Hilfe der Leftmost-Outermost-Substitution effizienter sein. In Kapitel 5.3.1 wird sogar ein Beispiel für eine Rechenvorschrift angegeben, die je nach gewählter Auswertungsstrategie unterschiedliche Ergebnisse liefert.

$$
\begin{aligned}
\mathbf{F}(2) &\Rightarrow \texttt{if } \mathbf{2} = \mathbf{1} \texttt{ then } 1 \\
&\qquad\qquad \texttt{else } \mathrm{F}(\mathrm{F}(2-1)) \\
&\Rightarrow \textbf{if } \texttt{false } \textbf{then } 1 \\
&\qquad\qquad \textbf{else } \mathrm{F}(\mathrm{F}(2-1)) \\
&\Rightarrow \mathbf{F}(\mathrm{F}(2-1)) \\
&\Rightarrow \texttt{if } \mathbf{F}(2-1) = 1 \texttt{ then } 1 \\
&\qquad\qquad\qquad \texttt{else } \mathrm{F}(\mathrm{F}(\mathrm{F}(2-1)-1)) \\
&\Rightarrow \texttt{if } (\texttt{if } \mathbf{2}-\mathbf{1} = \mathbf{1} \texttt{ then } 1 \\
&\qquad\qquad\qquad \texttt{else } \mathrm{F}(\mathrm{F}((2-1)-1)))) = 1 \texttt{ then } 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else } \mathrm{F}(\mathrm{F}(\mathrm{F}(2-1)-1)) \\
&\ldots \\
&\Rightarrow \texttt{if } (\textbf{if } \texttt{true } \textbf{then } 1 \\
&\qquad\qquad\qquad \textbf{else } \mathrm{F}(\mathrm{F}((2-1)-1)))) = 1 \texttt{ then } 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else } \mathrm{F}(\mathrm{F}(\mathrm{F}(2-1)-1)) \\
&\Rightarrow \texttt{if } \mathbf{1} = \mathbf{1} \texttt{ then } 1 \\
&\qquad\qquad \texttt{else } \mathrm{F}(\mathrm{F}(\mathrm{F}(2-1)-1)) \\
&\Rightarrow \textbf{if } \texttt{true } \textbf{then } 1 \\
&\qquad\qquad\quad \textbf{else } \mathrm{F}(\mathrm{F}(\mathrm{F}(2-1)-1)) \\
&\Rightarrow 1
\end{aligned}
$$

Abbildung 2.4: Auswertung von `F(2)` nach der Leftmost-Outermost Strategie

3 Aufgabenstellung

Das KIEL-System unterstützte bislang den in 2.4 und 2.5 geschilderten Sprachumfang. Damit konnten bereits viele Algorithmen umgesetzt werden. Leider wurden Nutzer durch die fehlende Unterstützung einiger Sprachelemente häufig zu ineffizienten oder unnötig komplizierten Implementierungen gezwungen.

Bei Beschränkung auf die unterstützten Sprachelemente müssen einzelne Ausdrücke häufig mehrfach ausgewertet werden und Programme werden oftmals unübersichtlich, wenn komplexe Terme wiederholt innerhalb von einer Rechenvorschriftsdeklaration auftreten.

Wie in 2.6 geschildert, lassen sich diese Probleme durch lokale Wertedeklaration vermeiden. In einer lokalen Wertedeklaration eingeführte Variablen können komplexe Terme abkürzen oder mehrfach in einem Rumpf verwendet werden – müssen aber nur einmal ausgewertet werden.

Um sinnvoll mit lokalen Wertedeklarationen arbeiten zu können, ist außerdem eine Unterstützung von Produkttypen hilfreich. Mit beiden Sprachelementen zusammen kann der Nutzer Rechenvorschriften erstellen und nutzen, die gleichzeitig mehrere Werte berechnen.

Weil das KIEL-System bei der Auswertung nicht nur die Leftmost-Innermost-Substitution unterstützt, kann es vorkommen, dass bei der Expansion einer lokalen Wertedeklaration der zugewiesene Wert noch nicht ausgewertet ist. Daher ist es nötig, eine Projektion von einem Tupel auf eine bestimmte Komponente dieses Tupels als neue Operation zuzulassen. Diese kann dann bei der Expansion der lokalen Wertedeklaration allen Variablen die passenden Komponenten der (eventuell noch nicht ausgewerteten) rechten Seite der Deklaration zuweisen.

Die Hauptaufgabe dieser Diplomarbeit ist daher, das KIEL-System um die vorgestellten Sprachelemente – Produkttypen, Projektionen und lokale Wertedeklarationen – zu erweitern.

Das KIEL-System wird seit Jahren in der Grundausbildung von Studenten der Informatik an der Christian-Albrechts-Universität zu Kiel eingesetzt. Bei dieser intensiven Nutzung sind in der Vergangenheit Fehler bei der Auswertung von Ausdrücken in der alten

Version des KIEL-Systems aufgefallen. Diese Fehler müssen beseitigt werden, um die Nutzung des Programms in der Lehre weiter uneingeschränkt zu ermöglichen.

Ebenfalls wurde von Nutzern des KIEL-Systems der Wunsch geäußert, dass sich das Programm die Position und Größe der einzelnen Fenster des KIEL-Systems merkt und bei einem Neustart der Anwendung wieder herstellt. Eine entsprechende Erweiterung des Programms soll erfolgen.

Darüber hinaus ist beabsichtigt, das KIEL-System um eine Funktion zu erweitern, die ermöglicht, einen Aufrufgraphen für ein vom Benutzer erstelltes Programm zu erzeugen.

4 Benutzeroberfläche

Dieses Kapitel kann als Einführung in die Bedienung der KIEL-Systems verwendet werden. Es beschreibt die Benutzerschnittstelle des Programms.

Die Bedienung des Systems erfolgt komfortabel mit der Maus über eine grafische Benutzeroberfläche. In mehreren Fenstern werden alle relevanten Informationen angezeigt sowie Möglichkeiten zur Interaktion gegeben. Eine übliche Ansicht wird in Abb. 4.1 auf der nächsten Seite gezeigt.

Das KIEL-System wird von der Kommandozeile aus mit dem Aufruf

```
# kiel <RETURN>
```

gestartet. Danach öffnet sich die grafische Benutzeroberfläche. Die wichtigsten Fenster der Anwendung sind:

Hauptfenster Kontrolle der anderen Fenster, Aufruf von Programmfunktionen

Ausdrucksfenster Liste von bereits eingegeben Ausdrücken

Rechenvorschriftenfenster Liste der definierten Rechenvorschriften

Undo-Redo-Fenster Bereits ausgeführte Auswertungsschritte rückgängig machen

Auswertungsfenster Einstellungen zur Auswertungsstrategie

Baumfenster Anzeige des aktuellen Ausdrucks als Baum, Steuern der Auswertung

Die Funktion der verschiedenen Fenster wird im Folgenden einzeln erläutert.

4.1 Hauptfenster

Das Hauptfenster (Abb. 4.2 auf der nächsten Seite) ist während der gesamten Programmausführung sichtbar. Es stellt die Schaltzentrale des KIEL-Systems dar. Das Fenster ist in fünf Bereiche untergliedert.

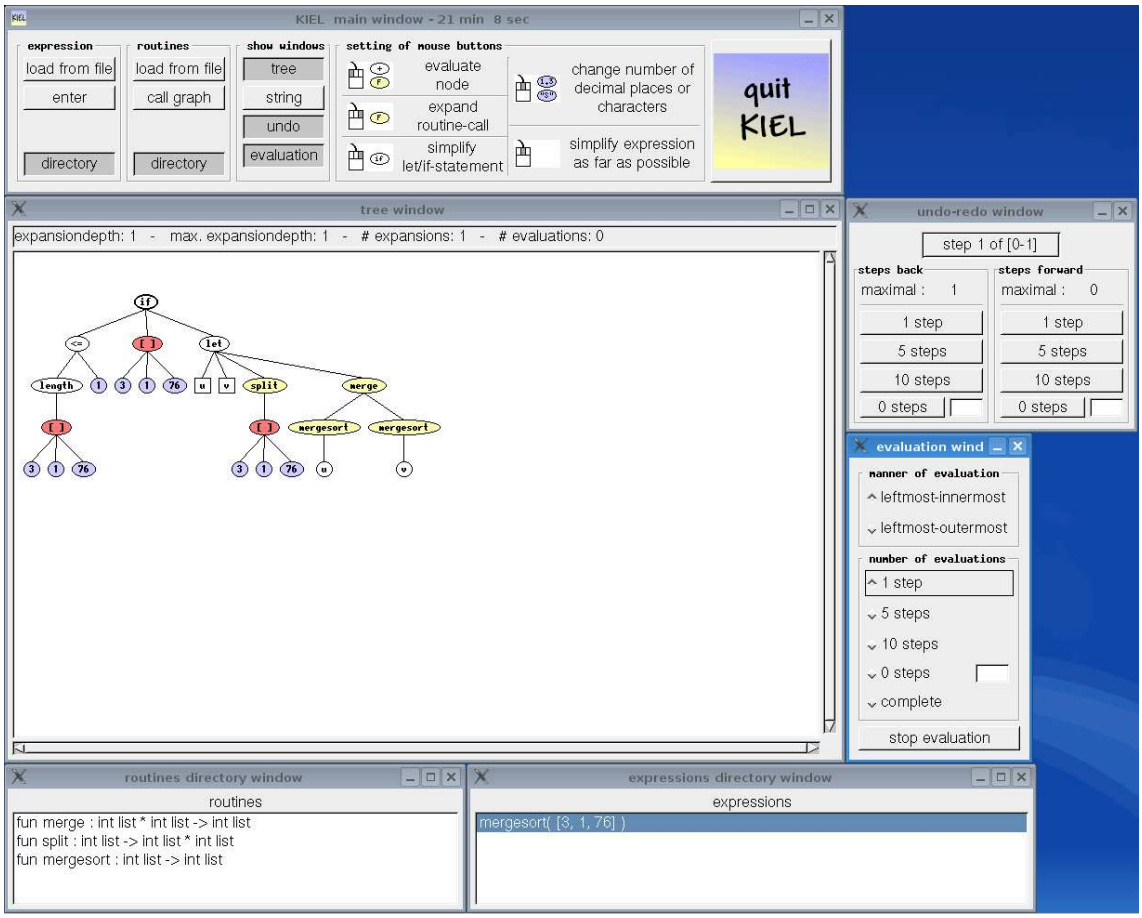


Abbildung 4.1: Gesamtansicht des KIEL-Systems

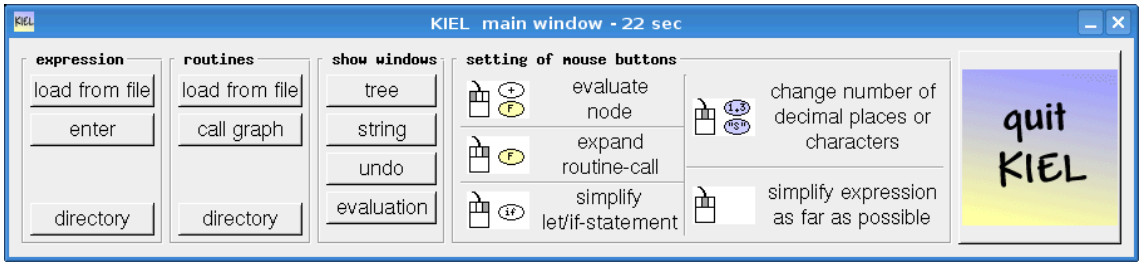


Abbildung 4.2: Hauptfenster

Ausdrücke

Hier sind drei Schaltflächen zusammengefasst, mit denen der Anwender die Fenster für die Verwaltung von Ausdrücken steuern kann.

Nach einem Klick auf *Load From File* öffnet sich ein Fenster, in dem eine Ausdrucksdatei zum Laden gewählt werden kann. Mit *Enter* öffnet der Nutzer ein Fenster, in dem er direkt einen Ausdruck eingeben kann. Durch wiederholtes Klicken auf *Directory* kann das Fenster mit der Liste der Ausdrücke geöffnet und wieder geschlossen werden.

Rechenvorschriften

In diesem Bereich sind Schaltflächen zusammengefasst, mit denen der Anwender die Fenster für die Verwaltung von Rechenvorschriften steuern kann.

Mit einem Klick auf *Load From File* kann der Anwender ein Fenster öffnen, in dem er eine Datei mit Rechenvorschriften auswählen kann, die dann vom KIEL-System geladen wird. Durch einen Klick auf *Call-Graph* wird das KIEL-System angewiesen, einen Aufrufgraph für die geladenen Rechenvorschriften anzuzeigen. Durch wiederholtes Klicken auf *Directory* kann das Fenster mit der Liste der geladenen Rechenvorschriften geöffnet und wieder geschlossen werden.

Fenster

Die vier Schaltflächen *Baum*, *Zeichenreihe*, *Undo* und *Auswertung* öffnen und schließen die entsprechenden Fenster.

Informationsbereich

Damit der Anwender sich die Bedeutung der verschiedenen Mausklicks im Baumfenster leichter merken kann, wird eine kleine Erinnerungshilfe angezeigt.

Beenden

Eine große Schaltfläche ist für die Beendigung des KIEL-Systems vorgesehen.

4.2 Ausdrucksfenster

Das Ausdrucksfenster (Abb. 4.3) zeigt alle Ausdrücke an, die vom Benutzer per Hand eingegeben oder aus Dateien geladen wurden. Durch einfachen *Linksklick* auf einen Aus-

Abbildung 4.3: Ausdrucksfenster

druck in der Liste veranlasst der Anwender das KIEL-System, diesen im Baumfenster grafisch darzustellen.

Mit einem *Rechtsklick* auf einen Ausdruck wird der Ausdruck in das Eingabefeld für neue Ausdrücke kopiert, das aus dem Hauptfenster heraus geöffnet werden kann. So kann ein bereits eingegebener Ausdruck leicht abgewandelt werden, ohne ihn komplett neu einzugeben.

4.3 Rechenvorschriftenfenster

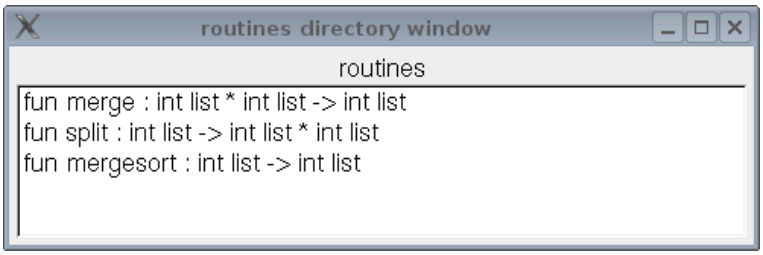


Abbildung 4.4: Rechenvorschriftenfenster

Alle geladenen Rechenvorschriften werden im Rechenvorschriftenfenster (Abb. 4.4) angezeigt. Diese Rechenvorschriften können in Ausdrücken verwendet werden, die der Nutzer lädt oder per Hand eingibt.

Wenn über die Schaltfläche "Load From File" eine Datei mit Rechenvorschriften geladen wurde, werden die im Ausdrucksfenster und im Rechenvorschriftenfenster angezeigten Listen gelöscht. In Rechenvorschriftenfenster werden danach die soeben geladenen Rechenvorschriften angezeigt.

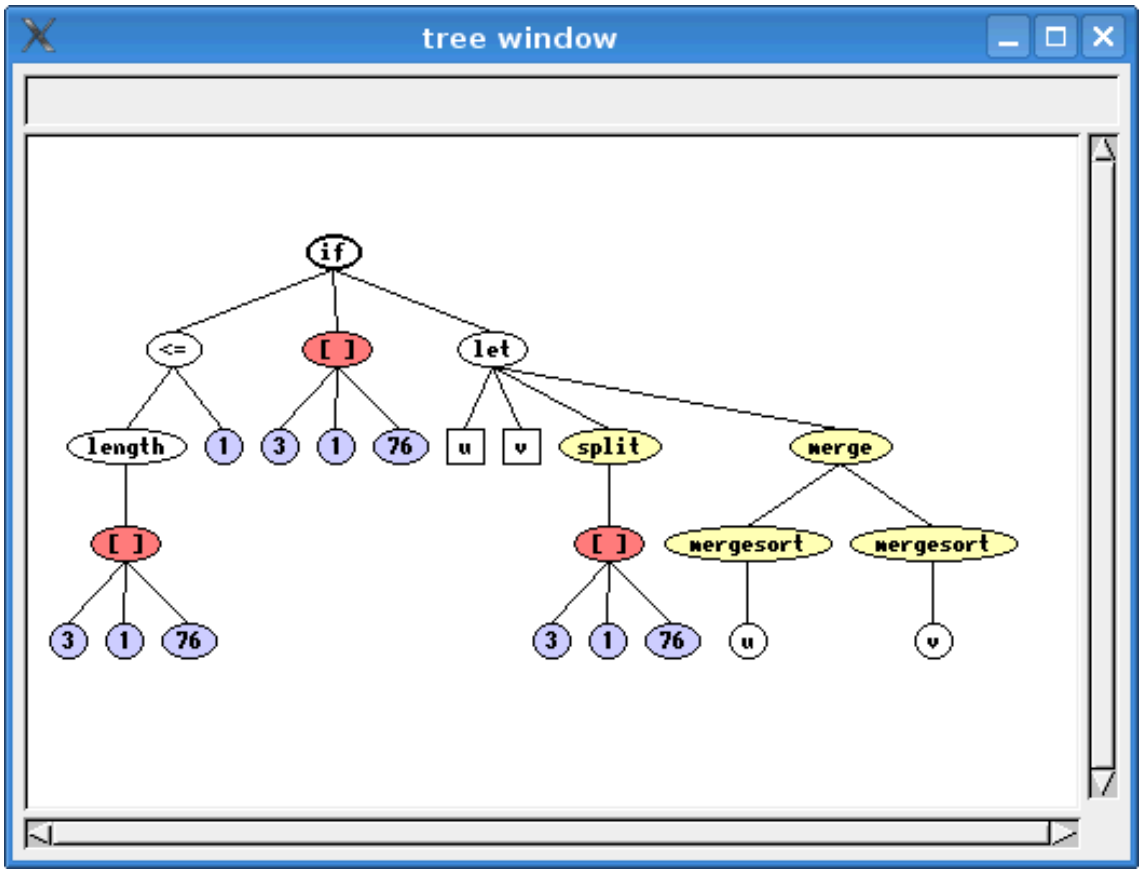


Abbildung 4.5: Baumfenster

4.4 Baumfenster

Im Baumfenster zeigt das KIEL-System den aktuell ausgewählten Ausdruck in Form eines Baumes an. Dabei werden die einzelnen Teile des Ausdrucks als Knoten dargestellt, die mit Kanten verbunden sind, wenn sie miteinander in einer Beziehung stehen.

Oben im Baumfenster befindet sich eine Statuszeile, die bei laufender Auswertung Informationen zum Stand der Abarbeitung anzeigt. Dort können folgende Daten eingesehen werden:

- die aktuelle Expansionstiefe,
- die maximale Expansionstiefe,
- die Anzahl der Expansionsschritte während der aktuellen Auswertung und
- die Anzahl der Vereinfachungsschritte während der aktuellen Auswertung.

Falls der Nutzer bemerkt, dass eine Auswertung zu lange dauert – zum Beispiel weil das Programm wegen einer Endlosschleife nicht terminiert, kann er mit Hilfe des *Ausführungsfensters* die laufende Auswertung abbrechen. Im Baumfenster wird danach der teilausgewertete Ausdruck angezeigt.

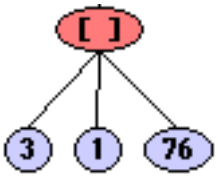
4.4.1 Darstellung der verschiedenen Knotenarten

Mit Hilfe einer unterschiedlicher Darstellungen der Knoten können einzelne Teilausdrücke unterschieden werden. Der zuletzt vom Nutzer bearbeitete Knoten wird zur Hervorhebung fett umrandet dargestellt. Die unterschiedlichen Darstellungen werden nachfolgend einzeln beschrieben:

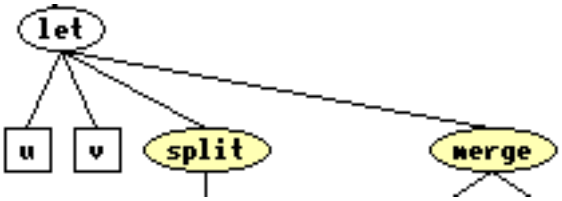
Unausgewertete Knoten werden weiß hinterlegt dargestellt.

Bereits ausgewertete Knoten werden blau hinterlegt dargestellt.

Aufrufe von nutzerdefinierten Rechenvorschriften werden gelb hinterlegt dargestellt.



Listen werden durch einen rot hinterlegten Knoten dargestellt. Die einzelnen Listenelemente sind darunter als Teilbäume angeordnet.



Bei lokalen Wertedeklarationen werden die Knoten für neu deklarierte Variablen eckig dargestellt, weil sie alleine kein auswertbarer Ausdruck sind. Danach folgt ein Teilbaum für die Werte, die den Variablen zugewiesen werden und ein Teilbaum mit dem Rumpf der lokalen Wertedeklaration.

4.4.2 Bedienung des Baumfensters

Im Baumfenster kann der Nutzer die schrittweise Auswertung des Ausdrucks mit einfachen Mausklicks steuern. Eine Zusammenfassung der verfügbaren Funktionen wird im Hauptfenster des KIEL-Systems angezeigt, so dass der Nutzer diese wichtigen Informationen immer im Blick hat.

Durch einen direkten Klick auf einen Knoten kann der Nutzer die Funktionen aktivieren, die sich auf diesen Knoten beziehen:

- **Auswertung eines Knotens** Nach einem *Linksklick* auf einen noch nicht ausgewerteten Knoten versucht das KIEL-System, diesen Knoten, beziehungsweise den unter ihm aufgespannten Teilausdruck, so weit wie möglich auszuwerten. Dabei werden die Einstellungen im Auswertungsfenster berücksichtigt, vor allem die Auswertungsart (Leftmost-Innermost, Rightmost-Outermost, Full) und die maximale Anzahl der Auswertungsschritte.
- **Vereinfachung einer Verzweigung** Bei einem *Rechtsklick* auf einen Knoten mit einer Verzweigung wird diese vereinfacht. Dabei wird zunächst die Bedingung ausgewertet, wobei die im Auswertungsfenster angegebene maximale Anzahl der Auswertungsschritte beachtet wird.
- **Expandieren eines Aufrufs einer Rechenvorschrift** Bei einem *Rechtsklick* auf einen Knoten mit dem Aufruf einer Rechenvorschrift, versucht das KIEL-System, diesen Aufruf zu expandieren.
- **Expansion einer lokalen Wertedeklaration** Mit einem *Rechtsklick* auf einen Knoten mit einer lokalen Wertedeklaration wird die Termersetzung ausgeführt. Dabei werden die Einstellungen im Auswertungsfenster beachtet, vor allem die Auswertungsart (Leftmost-Innermost, Rightmost-Outermost, Full) und die maximale Anzahl der Auswertungsschritte.
- **Einstellungen zur Darstellung** Mit dem *Mausrad* kann für einige Knoten ein Menü geöffnet und bedient werden, in dem zum Beispiel für Zeichenreihen die Anzahl der angezeigten Zeichen oder für Maschinenzahlen die Anzahl der angezeigten Nachkommastellen festegelegt werden.

Durch einen Klick in den freien Bereich des Baumfensters kann der Nutzer Funktionen aktivieren, die sich auf den gesamten dargestellten Ausdruck beziehen:

- **Weitestgehende Vereinfachung des Ausdrucks** Mit einem Klick der *mittleren Maustaste* im freien Bereich wird der Ausdruck soweit wie möglich vereinfacht. Da-

bei werden Basisoperationen ausgeführt und Verzweigungen vereinfacht, aber keine vom Anwender vorgegebenen Rechenvorschriften expandiert. Die im Auswertungsfenster angegebene maximale Anzahl der Auswertungsschritte wird beachtet.

4.5 Undo-Redo-Fenster

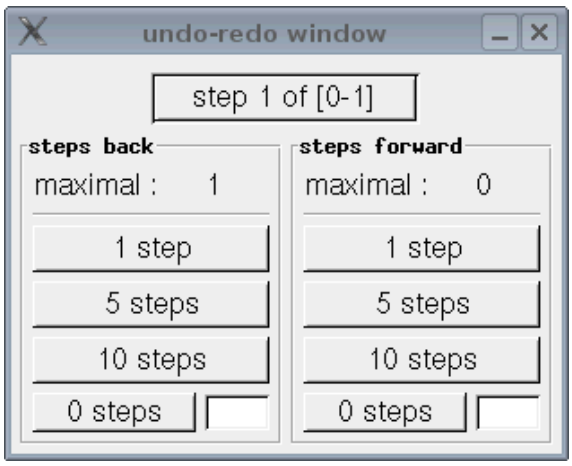


Abbildung 4.6: Undo-Redo-Fenster

Bei der schrittweisen Auswertung eines Ausdrucks wird nach jedem Mausklick des Benutzers der Zustand des Baums gespeichert. Mit Hilfe der Schaltflächen in der linken Hälfte des Undo-Redo-Fensters kann der Anwender beliebig viele Auswertungsschritte wieder zurücknehmen. Wenn er zwischenzeitlich keine neue Auswertungsschritte gestartet hat, kann der Nutzer mit den Schaltflächen in der rechten Hälfte bereits rückgängig gemachte Auswertungsschritte nochmals ausführen.

4.6 Auswertungsfenster

Im Auswertungsfenster nimmt der Anwender die Grundeinstellungen für die interaktive Auswertung der Ausdrücke vor. Das Fenster ist in drei Funktionsbereiche gegliedert. Oben kann die Auswertungsstrategie gewählt werden. Zur Auswahl stehen hier Leftmost-Innermost und Leftmost-Outermost.

In der Mitte kann eingestellt werden, wie viele Auswertungsschritte als Reaktion auf einen Klick der Nutzers im Baumfenster ausgeführt werden. Bei der Wahl von einem Auswertungsschritt kann der Nutzer einen Ausdruck wirklich schrittweise ausführen lassen. Daneben ist auch eine Auswertung von mehreren Schritte auf einmal oder die kompletten Auswertung des gewählten Knoten bzw. Teilausdrucks möglich.

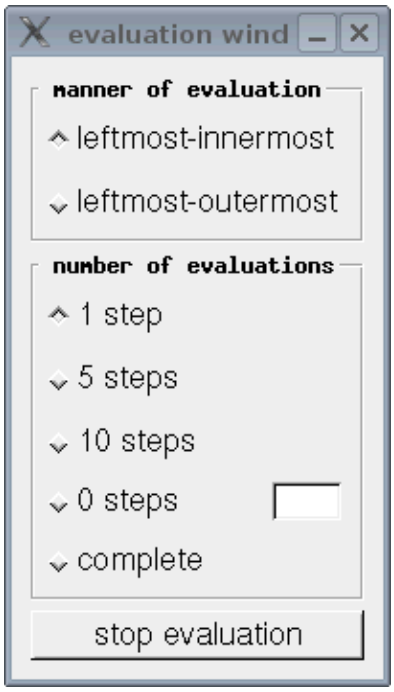


Abbildung 4.7: Auswertungsfenster

Mit einem Klick auf die *Stop Evaluation* Schaltfläche kann eine laufende Auswertung abgebrochen werden.

4.7 Kommandozeilenparameter

Beim Start des KIEL-Systems können mehrere Parameter auf der Kommandozeile mit übergeben werden. Dabei handelt es sich zum einen um Parameter zu Einschalten zusätzlicher Funktionen, zum anderen um das direkte Laden von Programmen und Ausdrücken aus Dateien.

Bei einer Auswertung von Termen mit vollständiger Substitution können sehr große Zwischenterme entstehen, deren Darstellung unübersichtlich ist. Daher sollte diese Auswertungsstrategie nur sehr bewusst eingesetzt werden. Um im Auswertungsfenster die Option für die vollständige Substitution anzuzeigen, muss beim Start der KIEL-Systems der Parameter `--full-eval` übergeben werden.

Mit Hilfe der Parameter `--prg` und `--exp` können direkt beim Start Dateien angeben werden, die als Programm beziehungsweise Ausdruck geladen werden sollen.

Ein Aufruf des KIEL-Systems auf der Kommandozeile kann zum Beispiel so aussehen:

```
kiel --full-eval --prg programs/fibonacci.prg
```

```
--exp programs/fibonacci_ok.prg
```

Damit wird das KIEL-System gestartet, die Schaltfläche für die vollständige Substitution angezeigt und sowohl Rechenvorschriften als auch ein Ausdruck zur Berechnung der Fibonacci-Zahlen geladen.

5 Anwendungsbeispiele für das erweiterte KIEL-System

Das KIEL-System ist vielfältig einsetzbar. Im folgenden Kapitel sollen eine ganze Reihe von Anwendungsbeispielen gegeben werden, die aufzeigen, wie Anwender die Funktionen des KIEL-Systems nutzen können.

Dabei wird zunächst dargestellt, wie sich effiziente Programme mit Hilfe der neu in das KIEL-System eingeführten Sprachelemente – Produkttypen und lokalen Wertedeklarationen – einfacher formulieren lassen. Danach wird in einem zweiten Teil des Kapitels vorgestellt, wie ein vom KIEL-System gezeichneter Aufrufgraph das Verständnis vom Zusammenspiel einzelner Rechenvorschriften verbessern kann.

Im dritten Teil wird dargelegt, wie man das KIEL-System benutzen kann, um die Unterschiede zwischen den Auswertungsstrategien Leftmost-Innermost und Leftmost-Outermost zu verdeutlichen. Der vorletzte Teil des Kapitels erklärt, wie im KIEL-System mit dem Problem der Verschattung von Variablen bei lokalen Wertedeklarationen umgegangen wird.

Abschließend zeigt ein Beispiel aus der Praxis, wie das KIEL-System helfen kann, Fehler in bestehenden Programmen zu finden und zu beheben.

5.1 Optimierung von Algorithmen

In diesem Abschnitt soll aufgezeigt werden, wie mit Hilfe der neu im KIEL-System eingeführten Sprachelemente effizientere Programme geschrieben werden können. Dabei wird anhand des Beispiels der Fibonacci-Zahlen demonstriert, wie eine mathematische Definition auf verschiedene Arten in ein Standard ML Programm umgesetzt werden kann. Dann werden mit Hilfe des KIEL-Systems die Unterschiede bei der Auswertung der verschiedenen Implementierungen aufgezeigt.

Die Fibonacci-Zahlen sind eine bekannte Zahlenfolge, die induktiv definiert ist: Die nullte und erste Fibonacci-Zahl ist 1. Danach lässt sich jede Fibonacci-Zahl in der Folge als die Summe der beiden vorherigen Fibonacci-Zahlen berechnen.

Mathematisch ist die n-te Fibonacci-Zahl über die Funktion $fib : \mathbb{Z} \to \mathbb{Z}$ festgelegt:

$$x \quad \Rightarrow \quad \begin{cases} 1 & : \text{ falls } x \in \{0, 1\} \\ fib(x-1) + fib(x-2) & : \text{ falls } x > 1 \\ \text{undefiniert} & : \text{ sonst.} \end{cases}$$

Es gibt eine Vielzahl von Verbindungen zwischen den Fibonacci-Zahlen und anderen Objekten in der Mathematik: Im Pascalschen Dreieck tauchen sie beispielsweise als Summen von Diagonalen auf. Die Formel von Binet setzt sie in Verbindung mit dem Goldenen Schnitt. Auch in der Kombinatorik erscheinen sie häufig. Und in der Natur kann man sie als Anzahl der Spiralen bei Pflanzenblättern beobachten. Diese Querverbindungen und weitere interessante Eigenschaften der Fibonacci-Zahlen werden ausführlich in [10] vorgestellt.

Um aufzuzeigen, wie die obige mathematische Definition auf unterschiedliche Arten in ein Programm umgesetzt werden kann, werden im Folgenden drei verschiedene Implementierungen vorgestellt und verglichen.

Die drei Versionen sind unter den Namen `fib_naiv`, `fib_effizient` und `fib` in der Datei `fibonacci.prg` in der Distribution des KIEL-Systems enthalten. Diese Datei muss zunächst im KIEL-System durch einen Klick auf die Schaltfläche "Load From File" und anschließende Auswahl des Dateinamens geladen werden. Danach wird im Auswertungsfenster zunächst die Auswertungsart auf Leftmost-Innermost gestellt und dann die Auswertung in Einzelschritten (Number of Evalutions auf 1) angewählt.

Nun können die in den folgenden Abschnitten vorgestellten Implementierungen analysiert werden.

5.1.1 Naive Implementierung Fibonacci-Zahlen

Eine naive Implementierung der Fibonacci-Funktion ergibt sich direkt aus der im letzten Abschnitt angegebenen mathematischen Definition. Eine direkte Umsetzung in Standard ML ist im Folgenden angegeben:

```
fun fib_naiv(n : int) : int =
    if n = 0 orelse n = 1 then 1
                          else fib_naiv(n-1) + fib_naiv(n-2);
```

Die Rechenvorschrift `fib_naiv` unterscheidet in einer Verzweigung die beiden ersten Fälle der mathematischen Definition aus dem letzten Abschnitt: Für die Argumente `0`

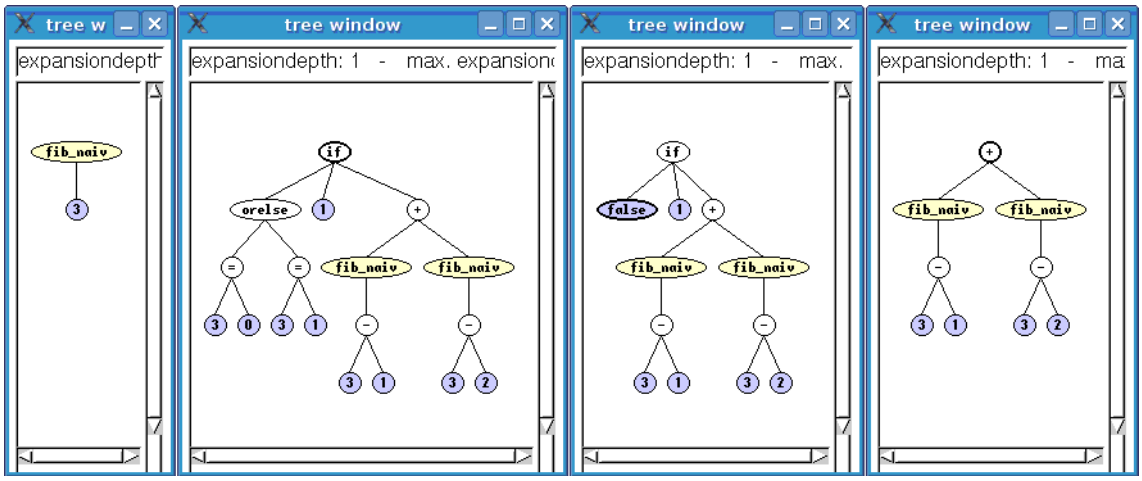


Abbildung 5.1: Auswertung `fib_naiv(3)`, Teil 1

oder `1` ist das Ergebnis `1` fest vorgegeben. Für alle anderen Argumente wird das Ergebnis aus den beiden vorherigen Fibonacci-Zahlen berechnet.

In das Fenster für die Eingabe von Ausdrücken wird zunächst der Term `fib_naiv(3)` eingegeben. Danach wird dieser neue Ausdruck in der Liste der Ausdrücke ausgewählt. Jetzt wird er als Baum im Baumfenster (siehe linkes Fenster in Abb. 5.1) dargestellt.

Die Auswertung wird durch einen Klick mit der linken Maustaste auf den Ursprungsknoten gestartet. Im ersten Schritt wird der Aufruf der Rechenvorschrift `fib_naiv` expandiert und durch den Rumpf der Rechenvorschrift ersetzt. Dabei wird im Rumpf der Variablenbezeichner `n` durch das Argument `3` ersetzt. Die Rechenvorschrift unterscheidet zwei Fälle in einer Verzweigung: Die nullte und erste Fibonacci-Zahl werden explizit als `1` festgelegt. Alle anderen Fibonacci-Zahlen werden als Summe der beiden Vorgänger berechnet.

Auch alle weiteren Schritte werden jeweils durch Linksklick auf den Ursprungsknoten gestartet. Zunächst muss die Abbruchbedingung ausgewertet werden. Das KIEL-System vereinfacht daher den Teilausdruck `3=0 orelse 3=1` zu `false`. Damit kann dann im nächsten Schritt auch die Verzweigung vereinfacht werden: Die Bedingung trifft nicht zu, daher wird der else-Zweig gewählt und wir erhalten den folgenden Ausdruck:

```
fib_ naiv(3-1)+fib_ naiv(3-2).
```

Entsprechend der gewählten Auswertungsstrategie wird nun von den beiden inneren Ausdrücken `3-1` und `3-2` der am weitesten links gelegene Ausdruck zu `2` vereinfacht (siehe Abb. 5.2 auf der nächsten Seite). Hierdurch wird der Aufruf von `fib_naiv(2)` zu einem inneren Ausdruck und kann im folgenden Schritt, wie im mittleren Fenster dargestellt, expandiert werden. Nun muss zunächst wieder die Abbruchbedingung `2=0`

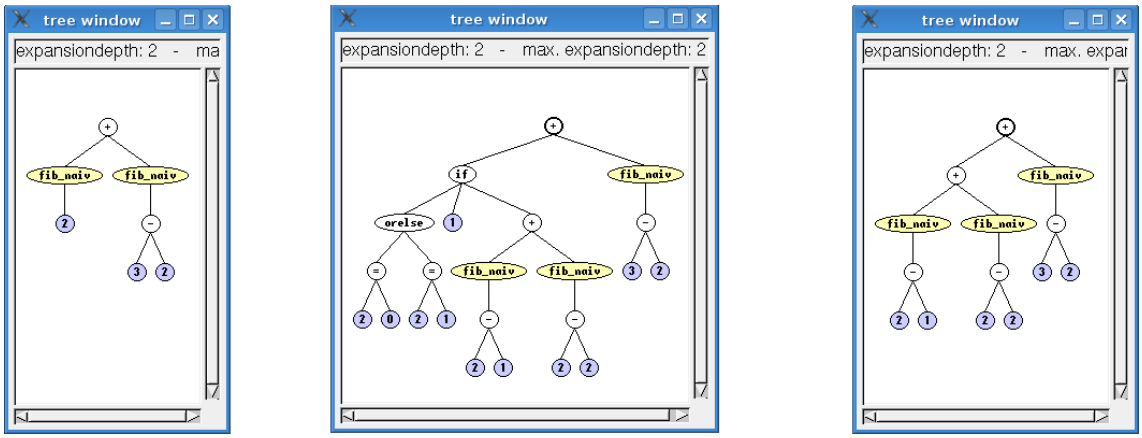


Abbildung 5.2: Auswertung `fib_naiv(3)`, Teil 2

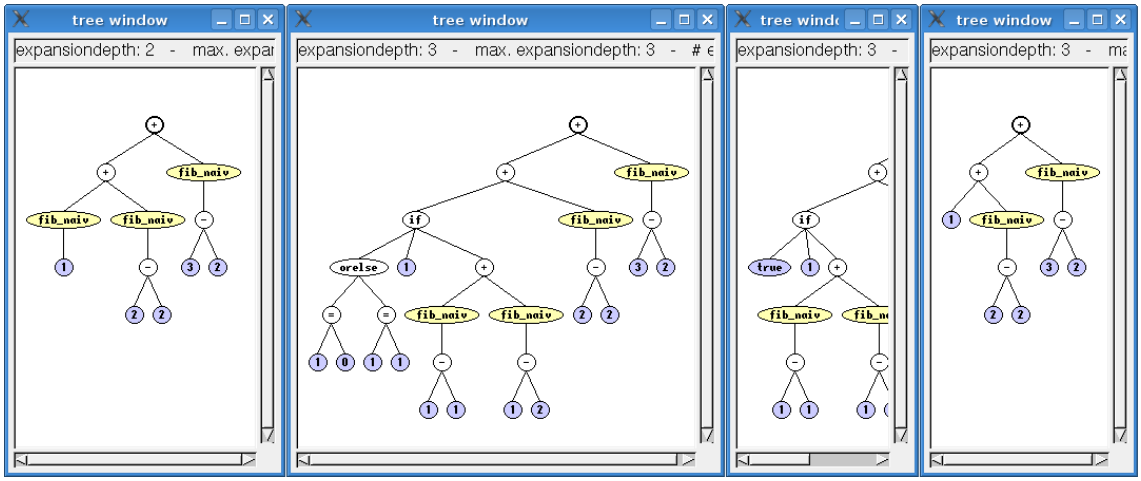


Abbildung 5.3: Auswertung `fib_naiv(3)`, Teil 3

`orelse 2=1` ausgewertet werden. Weil die Rekursion noch nicht abgeschlossen ist, ist das Resultat wieder `false` und die Berechnung von `fib_naiv(2)` wird auf den Ausdruck `fib_naiv(2-1) + fib_naiv(2-2)` zurückgeführt. Insgesamt ist die Auswertung mittlerweile bei folgendem Ausdruck angelangt, der im rechten Fenster in Abb. 5.2 dargestellt wird:

```
fib_ naiv(2-1)+fib_ naiv(2-2)+fib_ naiv(3-2).
```

Von den inneren Ausdrücken wird nun zunächst wieder der am weitesten links gelegene Ausdruck `2-1` zu `1` vereinfacht (siehe Abb. 5.3). Danach kann im nächsten Schritt der Aufruf `fib_naiv(1)` expandiert werden. Weil diesmal die Abbruchbedingung `1=0 orelse 1=1` erreicht ist, bricht die Rekursion hier ab und die Verzweigung kann zur explizit angegebenen ersten Fibonacci-Zahl `1` vereinfacht werden. Insgesamt wurde der ursprüngliche Ausdruck `fib_naiv(3)` mittlerweile zum nachfolgenden Ausdruck aus-

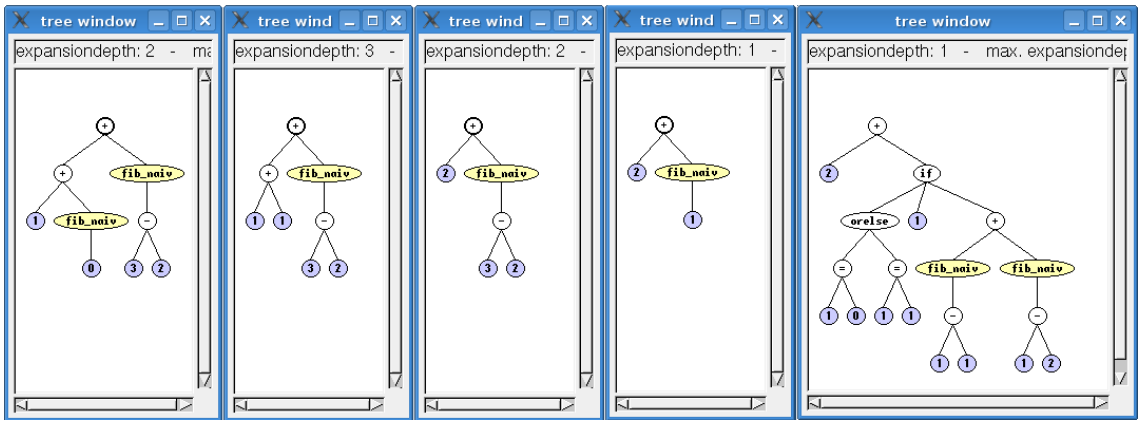


Abbildung 5.4: Auswertung `fib_naiv(3)`, Teil 4

gewertet, der auch im rechten Fenster der Abbildung 5.3 auf der vorherigen Seite dargestellt ist:

```
1+fib_ naiv(2-2)+fib_ naiv(3-2).
```

Nun wird zunächst der innere Ausdruck `2-2` ist zu `0` vereinfacht, wie im linken Fenster in Abb. 5.4 zu sehen ist. Dann wird der Aufruf `fib_naiv(0)` expandiert und ausgewertet. Genau wie nach der Expansion von `fib_naiv(1)` ist auch hier die Abbruchbedingung erfüllt und der Ausdruck wird direkt zu `1` ausgewertet, wie im zweiten Fenster in der Abbildung dargestellt. Nun können im nächsten Schritt die Resultate von `fib_naiv(0)` und `fib_naiv(1)` addiert werden: `1+1` wird zu `2`. Danach sind die Argumente des verbleibenden Aufrufs von `fib_naiv` an der Reihe; der Teilausdruck `3-2` wird zu `1` vereinfacht und es verbleibt als Ausdruck:

```
2+fib_ naiv(1).
```

Nun offenbart sich die Crux der naiven Implementierung, denn der Ausdruck `fib_naiv(1)` wurde bereits in einem früheren Schritt ausgewertet. Die nachfolgenden Berechnungen werden also unnötigerweise zum zweiten Mal ausgeführt.

Entsprechend der Auswertungsstrategie wird der Aufruf der Rechenvorschrift expandiert. In den nächsten beiden Schritten (siehe Abb. 5.5 auf der nächsten Seite) muss zunächst die Abbruchbedingung `1=0 orelse 1=1` der Verzweigung zu `true` ausgewertet werden. Somit kann im folgenden Schritt die Verzweigung auf die explizit angegebene erste Fibonacci-Zahl `1` vereinfacht werden. Die Auswertung des Ausdrucks `fib_naiv(3)` ist damit beim Ausdruck

```
2+1
```

angelangt. Dieser kann nun im letzten Schritt der Auswertung zu `3` vereinfacht werden.

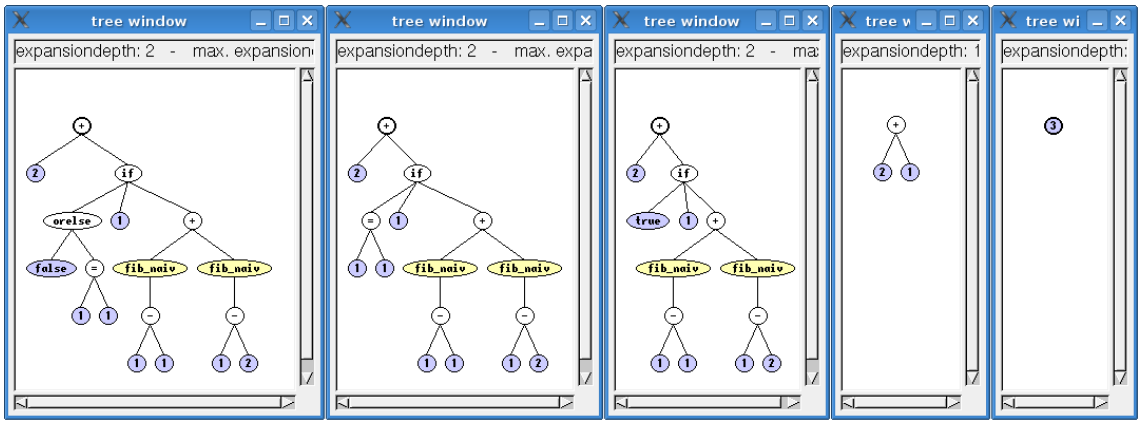


Abbildung 5.5: Auswertung `fib_naiv(3)`, Teil 5

Wie man sehen kann, musste im Laufe dieser Auswertung der Term `fib_naiv(1)` zwei Mal ausgewertet werden.

Wenn mit Hilfe der Rechenvorschrift `fib_naiv` höhere Fibonacci-Zahlen berechnet werden sollen, kommt es zu vielen Doppelauswertungen und die Laufzeit der Berechnung steigt stark an. Genauer gesagt, ergibt sich die Anzahl der notwendigen Expansionen der Rechenvorschriften `fib_naiv` für alle Argumente $n \geq 0$ als $fib(n) * 2 - 1$.

Mit Hilfe einiger Tricks lassen sich diese Doppelauswertungen vermeiden.

5.1.2 Effiziente Implementierung

Eine intelligente Implementierung der Fibonacci-Funktion berücksichtigt die Tatsache, dass sich jede Fibonacci-Zahl aus ihren zwei Vorgängern ergibt, und berechnet die Zahlenkette paarweise: Das Paar aus der nullten und ersten Fibonacci-Zahl steht wegen der mathematischen Definition als $(1, 1)$ fest. Ebenfalls aus der Definition folgt, dass, wenn (x, y) ein Paar der n-ten und der darauf folgenden Fibonacci-Zahl ist, $(y, x + y)$ das Paar aus der $n + 1$-ten und der darauf folgenden Fibonacci-Zahl ist.

Auch bevor das KIEL-System im Rahmen dieser Diplomarbeit um lokale Wertedeklarationen und Produkttypen erweitert wurde, war eine solche effiziente Implementierung möglich und wird in [2] angegeben. Diese Implementierung verwendet den benutzerdefinierten Datentyp `TIntPair`, um ein Paar von Fibonacci-Zahlen darzustellen. Um auf die einzelnen Elemente dieses Paares zurückzugreifen oder den Nachfolger des Paares zu berechnen, sind jeweils eigene Hilfsfunktionen notwendig. Dadurch wird der nachfolgend angegebene Quelltext deutlich unübersichtlicher als bei der naiven Implementierung:

```
datatype TIntPair = mk of int * int;
```

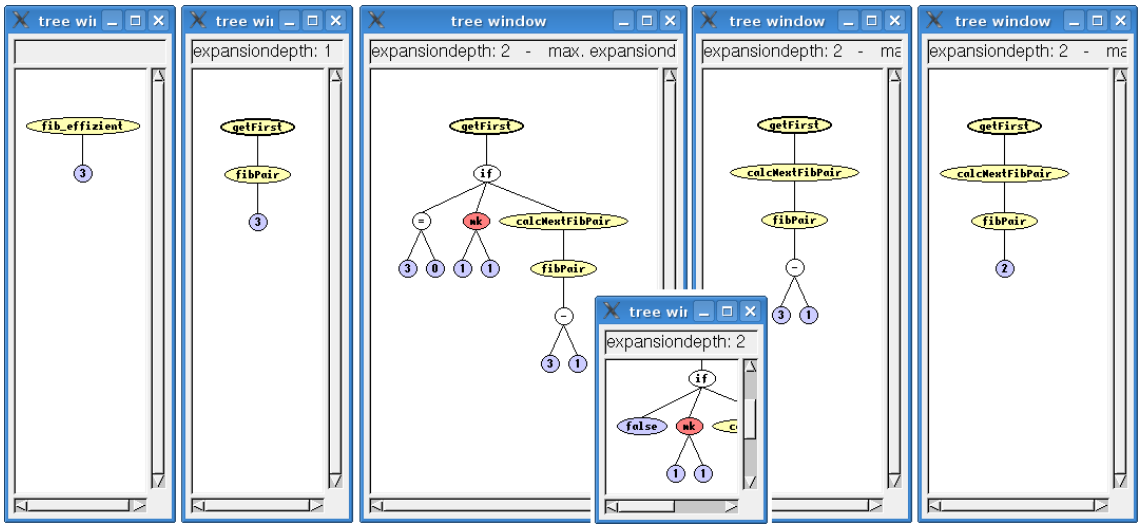


Abbildung 5.6: Auswertung `fib_effizient(3)`, Teil 1

```
fun getFirst( mk(a,b) : TIntPair ) : int =
    a;

fun calcNextFibPair( mk(a,b) : TIntPair ) : TIntPair =
    mk(b,a+b);

fun fibPair( n : int ) : TIntPair =
    if n = 0 then mk(1,1)
             else calcNextFibPair(fibPair(n-1));

fun fib_effizient( n : int ) : int =
    getFirst(fibPair(n));
```

Mit Hilfe dieser Rechenvorschriften können die Fibonacci-Zahlen deutlich schneller berechnet werden – auch wenn dies auf Kosten der Lesbarkeit des Quelltextes geht. Die Auswertung des Ausdrucks `fib_effizient(3)` soll im Folgenden kurz skizziert werden.

Ausgehend vom Ausdruck `fib_effizient(3)` (siehe Abb. 5.6), wird zunächst der Aufruf von `fib_effizient` expandiert und durch den Rumpf der Rechenvorschrift ersetzt: Die dritte Fibonacci-Zahl soll als erste Komponente des dritten Fibonacci-Zahlenpaars berechnet werden. Dieses dritte Fibonacci-Zahlenpaar wird über den Aufruf von `fibPair(3)` bestimmt. Im folgenden Schritt wird nun dieser Aufruf expandiert und durch den Rumpf der Rechenvorschrift ersetzt. Die Rechenvorschrift `fibPair` unterscheidet über eine Verzweigung zwei Fälle: Das nullte Fibonacci-Zahlenpaar ist als Auf-

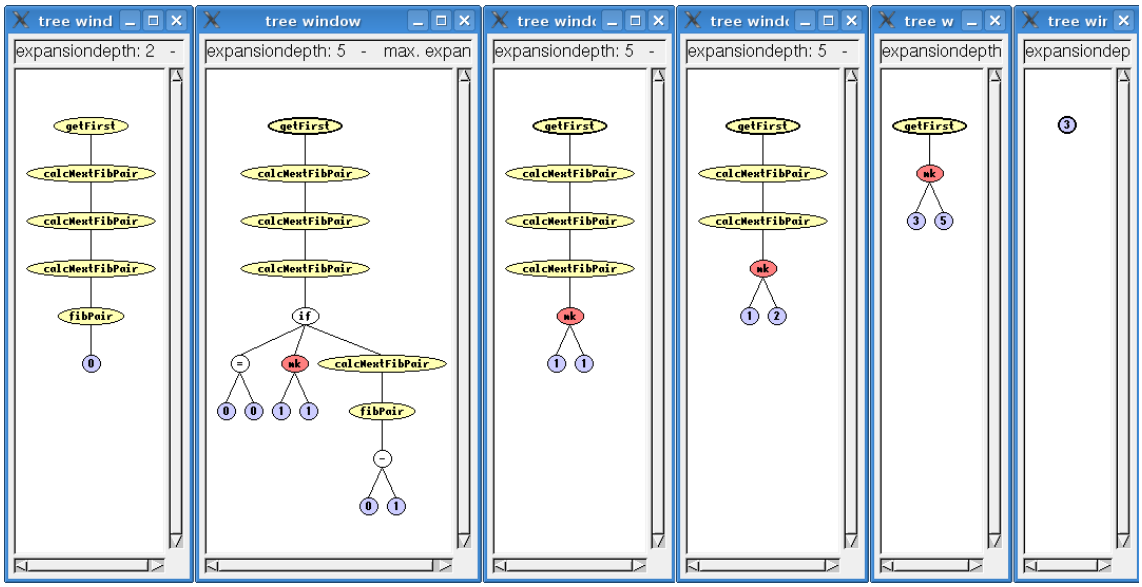


Abbildung 5.7: Auswertung `fib_effizient(3)`, Teil 2

ruf der Typkonstruktors `mk(1,1)` fest vorgegeben. Alle anderen Zahlenpaare werden über die Rechenvorschrift `calcNextFibPair` aus dem vorigen Zahlenpaar berechnet.

Da die Abbruchbedingung `3=0` in der Verzweigung nun zu `false` ausgewertet wird – wie im kleinen Fenster in Abb. 5.6 dargestellt – kann die Verzweigung zu ihrem zweiten Fall vereinfacht und `fibPair(3)` auf `calcNextFibPair(fibPair(3-1))` zurückgeführt werden. In diesem Ausdruck wird nun wiederum `3-1` zu `2` vereinfacht.

In einer Reihe ähnlicher Schritte werden nun immer wieder Aufrufe von `fibPair` expandiert. Solange die Rekursion noch nicht beim nullten Fibonacci-Zahlenpaar angekommen ist, wird `fibPair(n)` immer auf den Ausdruck `calcNextFibPair(fibPair(-n-1))` zurückgeführt, der das n-te Paar aus dem Vorgängerpaar berechnen soll.

Schließlich muss irgendwann der Teilausdruck `fibPair(0)` ausgewertet werden (siehe linkes Fenster in Abb. 5.7). Zunächst wird dieser Aufruf expandiert. Weil jetzt aber die Abbruchbedingung `0=0` der Verzweigung zu `true` vereinfacht wird, kann die Verzweigung dieses Mal auf das explizit angegebene nullte Fibonacci-Zahlenpaar `mk(1,1)` vereinfacht werden und die Auswertung ist beim Ausdruck

```
getFirst(calcNextFibPair(
      calcNextFibPair(calcNextFibPair(mk(1,1)))))
```

angekommen. Ausgehend vom Fibonacci-Zahlenpaar `mk(1,1)`, soll dieser Ausdruck die drei folgenden Zahlenpaare berechnen und schließlich die erste Komponente des Paars extrahieren. Der Ausdruck wird nun schrittweise durch die Expansion des jeweils

innersten Aufrufs von `calcNextFibPair` ausgewertet, bis wir den Ausdruck

```
getFirst(mk(3,5))
```

erreichen. Nach der Expansion dieser Rechenvorschrift mit Mustererkennung terminiert die Auswertung des Ausdrucks `fib_effizient(3)` erfolgreich und das Ergebnis `3` steht fest.

Beim Vergleich der beiden Auswertungen `fib_naiv` und `fib_effizient` fällt sofort auf, dass die Rekursion bei `fib_effizient` gradlinig verläuft und kein Ausdruck mehrfach ausgewertet wird. Bei `fib_naiv` hingegen kommt es zu Mehrfachauswertungen einzelner Ausdrücke.

5.1.3 Implementierung mit Hilfe lokaler Wertedeklarationen

Die Implementierung für die Berechnung der Fibonacci-Zahlen aus dem letzten Abschnitt ist zwar effizienter als die naive Implementierung, aber auch wesentlich komplizierter. Es sind ein neu deklarierter Datentyp und vier Rechenvorschriften notwendig, um die Berechnung zu erledigen.

Mit Hilfe von lokalen Wertedeklarationen jedoch lässt sich der effiziente Algorithmus wesentlich kompakter und übersichtlicher implementieren, wie im folgenden Quelltext zu erkennen ist:

```
fun G (n : int) : int * int =
  if n = 0 then (1,1)
           else let val (x : int, y : int) = G(n-1)
                in  (y,x+y)
           end;

fun fib(n : int) : int =
  let val (x : int, y : int) = G(n)
  in  x
  end;
```

Eine rekursive Hilfsrechenvorschrift `G` übernimmt die Berechnung des n-ten Paares von Fibonacci-Zahlen. Die Rechenvorschrift `fib` selbst hat nur die Aufgabe, `G` aufzurufen und aus dem von `G` berechneten Fibonacci-Zahlenpaar die erste Fibonacci-Zahl zurückzugeben.

Wie bei den vorherigen Implementierungen sollen die notwendigen Auswertungsschritte zur Berechnung der dritten Fibonacci-Zahl skizziert werden.

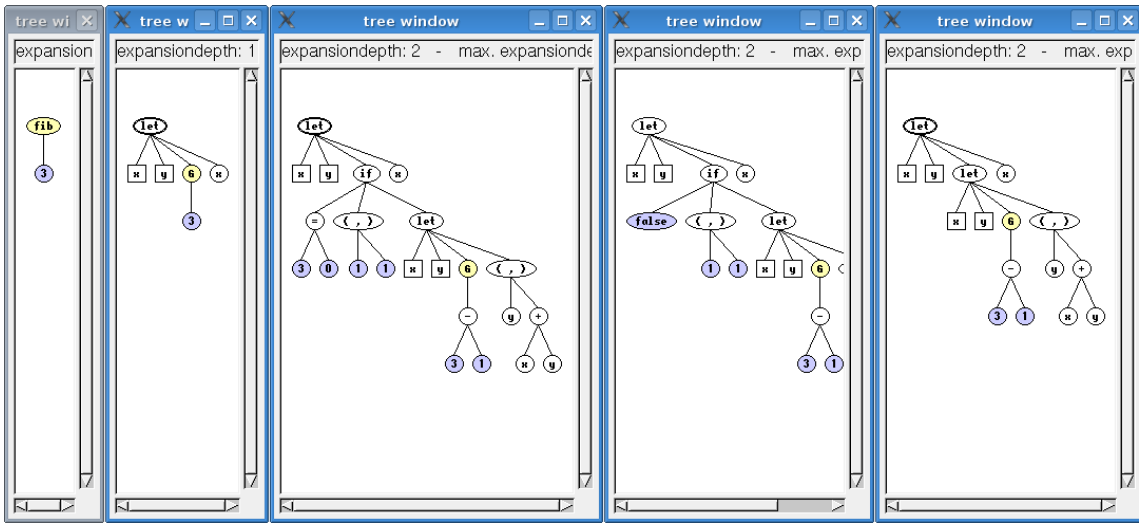


Abbildung 5.8: Auswertung `fib(3)`, Teil 1

Ausgehend vom Ausdruck `fib(3)` (siehe Abb. 5.8), wird im ersten Schritt zunächst der Aufruf der Rechenvorschrift `fib` expandiert. Diese Rechenvorschrift führt die Berechnung von `fib(3)` auf die Berechnung des ersten Elements des dritten Paares von Fibonacci-Zahlen zurück. Das dritte Paar soll durch die Hilfsrechenvorschrift `G` berechnet werden. Im darauf folgenden Schritt wird der Aufruf `G(3)` expandiert. Die Rechenvorschrift `G` unterscheidet über eine Verzweigung zwei Fälle: Wenn das nullte Paar berechnet werden soll, wird das Paar `(1,1)` zurückgegeben, ansonsten wird das n-te Paar aus dem vorherigen Paar nach der Formel $(x_n, y_n) = (x_{n-1}, x_{n-1} + y_{n-1})$ berechnet.

In den nächsten beiden Schritten wird zunächst die Abbruchbedingung `3=0` zu `false` ausgewertet und dann die Verzweigung vereinfacht, so dass die Berechnung von `G(3)` mit Hilfe des vorherigen Paars `G(3-1)` geschieht, wie im rechten Fenster in der Abbildung 5.8 dargestellt.

Im folgenden Schritt (siehe Abb. 5.9 auf der nächsten Seite) wird zunächst der Ausdruck `3-1` zu `2` vereinfacht. Nun kann der Aufruf von `G(2)` expandiert werden. Genau wie nach der letzten Expansion ist die Abbruchbedingung `2=0` noch nicht erreicht und wird im Folgeschritt zu `false` ausgewertet. Also kann die Verzweigung wieder auf ihren else-Zweig vereinfacht werden und die Berechnung von `G(2)` wird mit Hilfe des vorherigen Paars `G(2-1)` vorgenommen.

Analog wird vom KIEL-System auch die Berechnung von `G(2-1)` auf einen Aufruf von `G(0)` zurückgeführt. Im nächsten abgebildeten Fenster (siehe Abb. 5.10 auf Seite 62) wurde dieser Aufruf von `G(0)` gerade expandiert. Wie im kleinen Fenster in der Mitte der Abbildung zu erkennen ist, wird die Abbruchbedingung `0=0` diesmal erfüllt und zu `true` ausgewertet. Damit kann die Verzweigung vereinfacht und durch das nullte Fibonacci-Zahlenpaar `(1,1)` ersetzt werden.

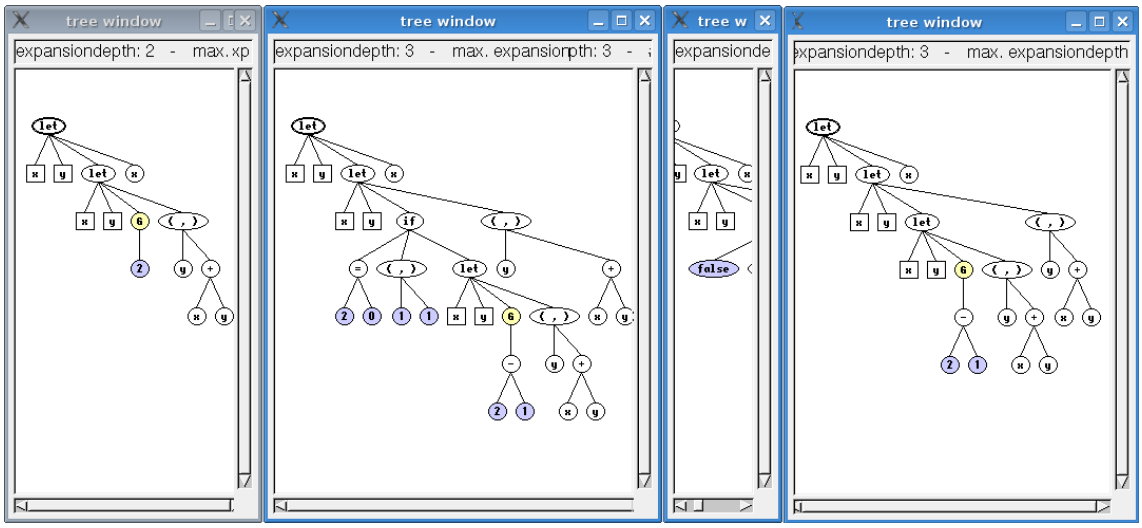


Abbildung 5.9: Auswertung `fib(3)`, Teil 2

Neben der äußersten lokalen Wertdeklaration, die nur dazu dient, die erste Komponente aus dem Fibonacci-Zahlenpaar zu extrahieren, enthält der Ausdruck nun drei geschachtelte lokale Wertedeklarationen. Diese berechnen ein Fibonacci-Zahlenpaar aus seinem Vorgänger, indem sie jeweils den Variablen `x` und `y` die beiden Elemente des letzten Tupels zuweisen und dann nach der oben angegebenen Formel das Nachfolgetupel als `(y,x+y)` berechnen.

Im ersten Schritt wird die innerste lokale Wertedeklaration expandiert (siehe Abb. 5.11 auf der nächsten Seite). Dabei wird `x` an `1` und `y` an `1` gebunden und dann die Variablenbezeichner im Rumpf durch die gebundenen Werte ersetzt. Somit wird die lokale Wertedeklaration durch den Ausdruck `(1,1+1)` ersetzt. Dieser Ausdruck kann nun im nächsten Schritt zu `(1,2)` vereinfacht werden. Analog werden in den nächsten Schritten immer zunächst die innen liegende lokale Wertedeklaration expandiert und dann das entstehende Tupel vereinfacht.

Im vorletzten Fenster in der Abbildung 5.11 wurde schließlich das dritte Fibonacci-Zahlenpaar `(3,5)` bestimmt. Nun muss nur noch die letzte lokale Wertedeklaration expandiert werden, um die `3` als erste Komponente des Tupels herauszugreifen.

Damit wurde die dritte Fibonacci-Zahl berechnet und die Auswertung wird erfolgreich beendet.

5.1.4 Effizienz der Auswertung

Auch zum Vergleich der Effizienz der drei Implementierungen kann man das KIEL-System benutzen. Wenn ein Ausdruck ausgewertet wird, zählt das System die Anzahl

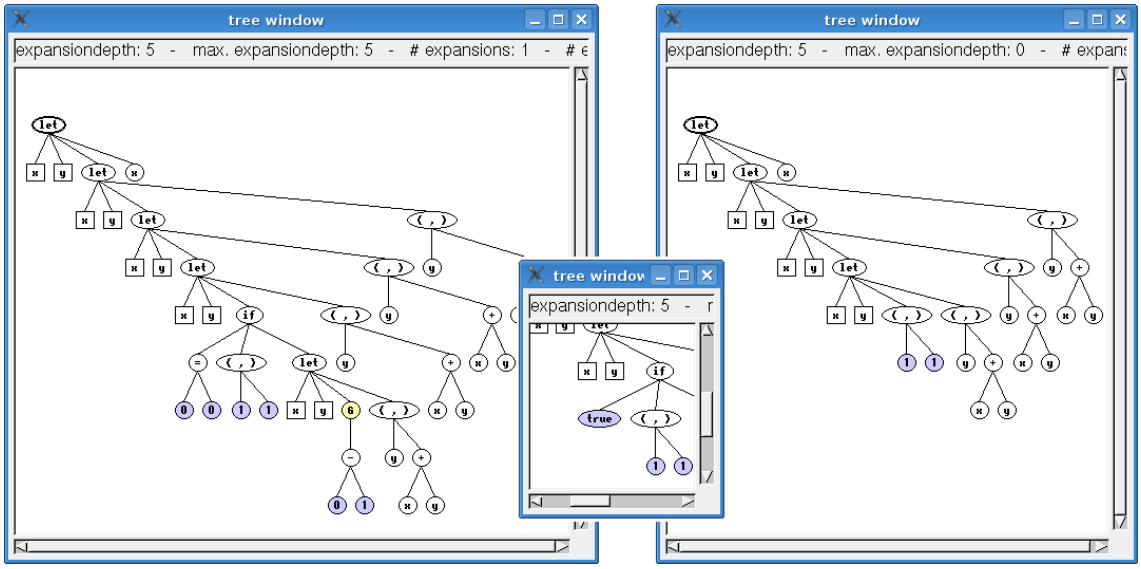


Abbildung 5.10: Auswertung `fib(3)`, Teil 3

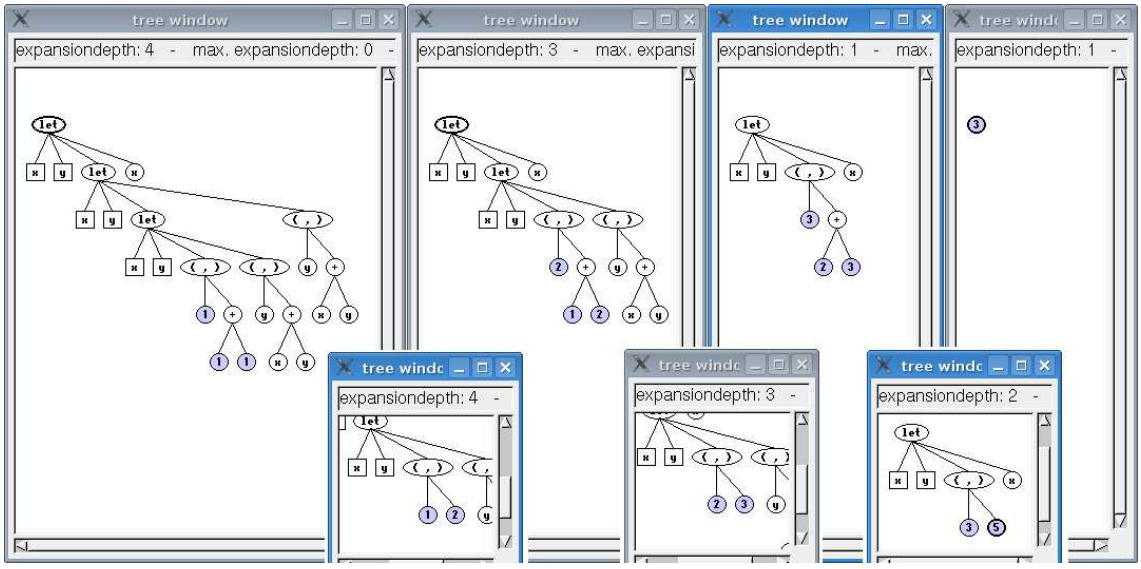


Abbildung 5.11: Auswertung `fib(3)`, Teil 4

der Expansionen von Rechenvorschriften und die Anzahl der Vereinfachungen von Basisoperatoren und -funktionen mit.

Um diese Funktion zu nutzen, muss im Auswertungsfenster zunächst eine komplette Auswertung nach der Leftmost-Innermost-Strategie ausgewählt werden.

Die drei Implementierungen sollen über die erforderlichen Auswertungsschritte bei der Berechnung der 10. und der 15. Fibonacci-Zahl verglichen werden. Zunächst müssen die Ausdrücke `fib_naiv(10)`, `fib_effizient(10)`, `fib(10)` sowie die Ausdrücke `fib_naiv(15)`, `fib_effizient(15)` und `fib(15)` in das Ausdruckseingabefenster eingegeben werden. Dann stehen sie im Ausdrucksfenster zur Auswahl bereit.

Jeder Ausdruck wird nun einzeln angewählt, so dass er im Baumfenster dargestellt wird. Durch Klick auf den Wurzelknoten wird dann die komplette Auswertung gestartet. Nach Abschluss der Auswertung kann im oberen Teil des Baumfensters die Anzahl der Expansions- und Vereinfachungsschritte abgelesen werden.

Für die Berechnung der 10. Fibonacci-Zahl liegen die drei Implementierungen noch relativ dicht beieinander und geben das Ergebnis nach kurzer Zeit zurück. Im Einzelnen ist die Anzahl der benötigten Schritte in der folgenden Tabelle aufgeführt:

	Auswertungsschritte	Expansionen
`fib_naiv(10)`	938	177
`fib_effizient(10)`	42	23
`fib(10)`	53	12

Tabelle 5.1: Erforderliche Schritte zur Berechnung der 10. Fibonacci-Zahl

Bei der Berechnung der 15. Fibonacci-Zahl sind die Unterschiede bereits wesentlich deutlicher zu erkennen. Die benötigte Anzahl von Schritten der drei Implementierungen ist der nachgestellten Tabelle zu entnehmen:

	Auswertungsschritte	Expansionen
`fib_naiv(15)`	10473	1973
`fib_effizient(15)`	62	33
`fib(15)`	78	17

Tabelle 5.2: Erforderliche Schritte zur Berechnung der 15. Fibonacci-Zahl

Offensichtlich sind die beiden Implementierungen `fib_effizient` und `fib` wesentlich effizienter als die ursprüngliche Version `fib_naiv`.

Die Rechenvorschrift `fib` mit lokalen Wertedeklarationen benötigt bei ihrer Auswertung immer die geringste Anzahl von Schritten. Von `fib_effizient` werden aber höchstens

doppelt so viele Expansionsschritte benötigt.

Das ist logisch – beide Versionen implementieren schließlich den gleichen Algorithmus für die Berechnung der gewünschten Fibonacci-Zahl. Bei `fib_effizient` müssen allerdings Hilfsrechenvorschriften die Arbeit erledigen, die in der kompakten Version `fib` durch lokale Wertedeklarationen geleistet wird.

5.2 Verständnis komplexer Programme

Das KIEL-System kann nicht nur Ausdrücke auswerten. Komplexe Programme oder Algorithmen, bei denen sich viele Rechenvorschriften gegenseitig aufrufen, sind gerade für ungeübte Nutzer oftmals schwer zu überblicken. Abhilfe kann das KIEL-System durch die Anzeige eines sogenannten Aufrufgraphen schaffen.

Im letzten Abschnitt wurden drei mögliche Implementierungen von Algorithmen für die Berechnung von Fibonacci-Zahlen angegeben: eine naive und ineffiziente sowie zwei effiziente Implementierungen. Von den effizienten Varianten ist eine mit Hilfe lokaler Wertedeklarationen umgesetzt worden, die andere mit Hilfe von nutzerdefinierten Datentypen und einer ganzen Reihe von Hilfsrechenvorschriften. Am Beispiel dieser drei Implementierungen soll der Nutzen eines Aufrufgraphen vorgestellt werden.

Zunächst muss ein Programm mit allen Rechenvorschriften im KIEL-System eingelesen werden. Der Quelltext aller drei Implementierungen ist als Datei `fibonacci.prg` in der Distribution des KIEL-Systems enthalten. Diese Datei kann nach Klick auf die Schaltfläche “Load From File” im Rechenvorschriftenbereich des Hauptfensters ausgewählt und geladen werden. Nun wird durch Klick auf die Schaltfläche “Call Graph” im gleichen Bereich ein Aufrufgraph angefordert.

Zur Anzeige des Aufrufgraphen wird vom KIEL-System das Programm `gv` gestartet. Der in Abb. 5.12 sichtbare Aufrufgraph stellt die drei Implementierungen nebeneinander dar. Dabei wird jede Rechenvorschrift durch einen Knoten und jeder Aufruf einer Rechenvorschrift aus einen Rechenvorschrift heraus durch einen Pfeil dargestellt.

Links wird die aus einer einzigen Rechenvorschrift `fib_naiv` bestehende naive Implementierung dargestellt. Wie man sieht, ist dies eine rekursive Rechenvorschrift, die sich selbst an zwei Stellen aufruft.

In der Mitte wird die effiziente Implementierung dargestellt, wie sie bereits vor der Erweiterung des KIEL-Systems im Rahmen dieser Diplomarbeit möglich war. Deutlich sind hier die drei Hilfsrechenvorschriften `getFirst`, `fibPair` und `calcNextFibPair` sowie der Typkonstruktor `mk` eingezeichnet. Der Nutzer kann sehen, dass die Rechenvorschrift `fib_effizient` die Hilfsrechenvorschriften `getFirst` und `fibPair` aufruft.

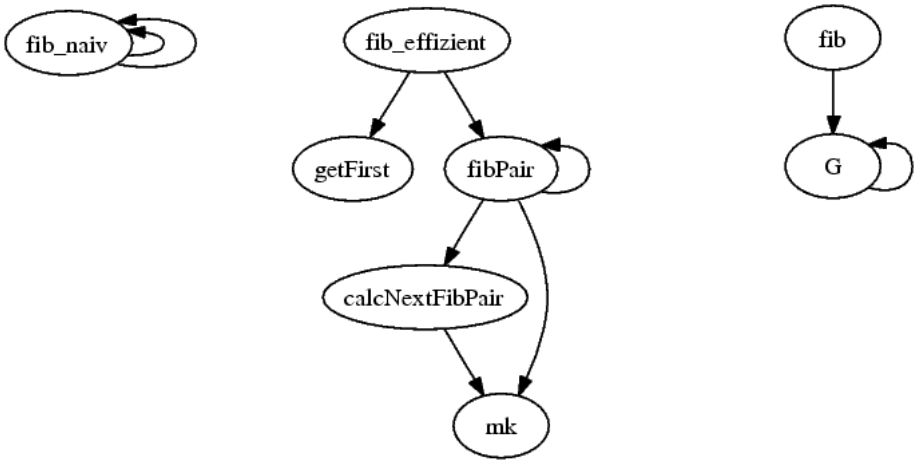


Abbildung 5.12: Aufrufgraph für die drei verschiedenen Fibonacci-Implementierungen

Die Rechenvorschrift `fibPair` ist als Kern der Rekursion erkennbar, da ein entsprechender Pfeil von der Rechenvorschrift auf sich selbst verweist. Daneben ruft `fibPair` sowohl den Typkonstruktor `mk` als auch die Hilfsrechenvorschrift `calcNextFibPair` auf. Diese Hilfsrechenvorschrift wiederum ruft ebenfalls den Typkonstruktor `mk` auf.

Rechts wird die Implementierung mit Hilfe lokaler Wertedeklarationen dargestellt. Man erkennt, dass die rekursive Berechnung durch die Hilfsrechenvorschrift `G` erfolgt, die nur einmal von der Rechenvorschrift `fib` aufgerufen wird.

Durch ein Betrachten des Aufrufgraphen kann ein Nutzer also bereits viel über ein Programm erfahren, ohne in den Quelltext schauen zu müssen. Ein späteres Studium desselben fällt auf diesem Wege leichter, weil die Zusammenhänge der verschiedenen Rechenvorschriften bereits deutlich geworden sind.

5.3 Unterschiede der Auswertungsstrategien

Wie in Abschnitt 2.7.3.1 ausführlich geschildert, macht ein einfaches Termersetzungssystem keine Vorgaben darüber, auf welchen reduzierbaren Teilausdruck (Reducible Expression, Redex) in einem Auswertungsschritt eine Termersetzungsregel angewendet werden muss. Erst durch die Wahl einer Auswertungsstrategie wird festgelegt, in welcher Reihenfolge ein Term ausgewertet wird.

Als Auswertungsstrategien sind dabei die Leftmost-Innermost-Substitution und die Leftmost-Outermost-Substitution am gebräuchlichsten. Bei der Leftmost-Innermost-Substitution wird von allen inneren Redeces (die also keinen weiteren Redex mehr enthalten) der im aufgeschriebenen Term am weitesten links gelegene Redex reduziert. Bei der

Leftmost-Outermost-Substitution hingegen wird der am weitesten links gelegene Redex reduziert – auch wenn er weitere reduzierbare Redeces enthält. Daneben existiert die Full-Substitution, bei der in einem Schlag alle Redeces reduziert werden.

Das KIEL-System kann Ausdrücke wahlweise nach einer dieser drei Auswertungsstrategien auswerten. Weil durch die schrittweise Auswertung im KIEL-System die Reihenfolge der Auswertungen sichtbar gemacht werden kann, eignet es sich hervorragend dafür, die Unterschiede zwischen den verschiedenen Auswertungsstrategien aufzuzeigen.

5.3.1 Morris-Funktion

Wie in 2.7.4 beschrieben, erscheint die Termauswertung nach der Leftmost-Innermost-Strategie auf den ersten Blick effizienter als die Termauswertung nach der Leftmost-Outermost-Strategie, weil Mehrfachauswertungen von Argumenten vermieden werden. Intuitiv könnte man auch erwarten, dass das Ergebnis einer Auswertung unabhängig von der gewählten Auswertungsstrategie ist. Beides muss allerdings nicht immer der Fall sein.

Als man sich in den 60er Jahren des 20. Jahrhunderts erstmals intensiv mit derartigen semantischen Problemen beschäftigte, fanden Mathematiker Funktionen, für die die oben angeführte Intuition nicht korrekt ist. Das bekannteste Beispiel ist die Morris-Funktion F, die auf J. H. Morris zurückgeht und in [15] vorgestellt wird. Sie ist mathematisch über die Funktion $F : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ definiert:

$$(x, y) \longmapsto \begin{cases} 1 & \text{falls } x \leq 0 \\ F(x-1, F(x-y, y) & \text{sonst} \end{cases} .$$

Obige Definition lässt sich einfach in eine entsprechende ML Rechenvorschrift übertragen, die nachfolgend wiedergegeben wird:

```
fun F(x:int, y:int) : int =
   if x <= 0 then 1
             else F(x-1,F(x-y,y));
```

Diese Rechenvorschrift ist in der Distribution des KIEL-Systems als Programm in der Datei `morris.prg` enthalten. Diese Datei muss zunächst geladen werden.

Nur soll die Auswertung des Ausdrucks `F(1,0)` betrachtet werden. Dazu wird dieser Ausdruck zunächst in das Ausdruckseingabefenster eingegeben und dann im Ausdrucksfenster angewählt. Danach zeigt das KIEL-System den Ausdruck als Baum im Baumfenster an.

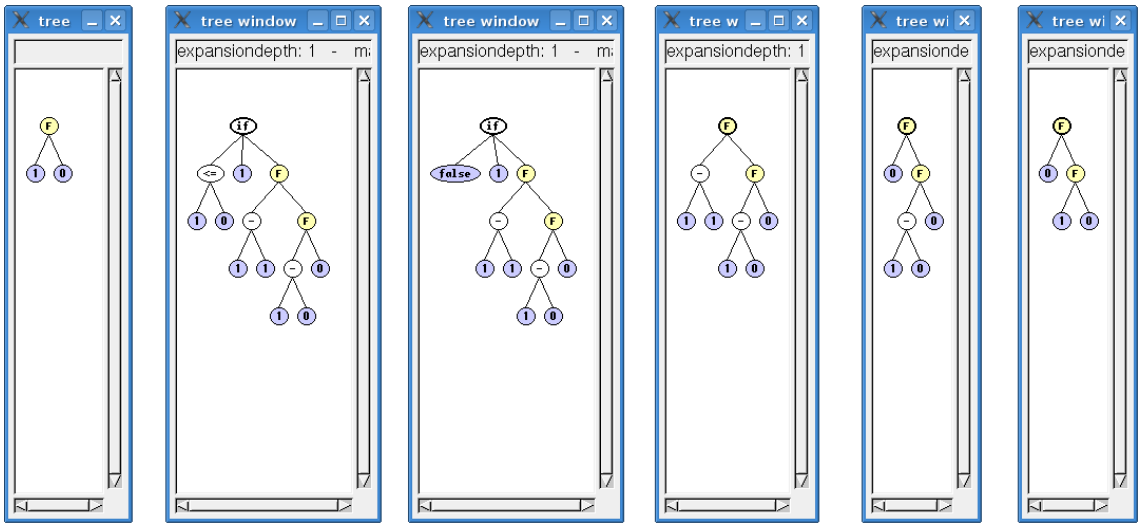


Abbildung 5.13: Auswertung `F(1,0)` nach Leftmost-Innermost-Strategie,

5.3.2 Leftmost-Innermost-Auswertung

Nun kann die schrittweise Auswertung beginnen. Zunächst soll die Auswertung mit Hilfe der Leftmost-Innermost-Substitution erfolgen. Dafür wählt man diese Auswertungsart im Auswertungsfenster an.

Im linken Fenster in Abb. 5.13 wird zunächst der ursprüngliche Ausdruck als Baum dargestellt. Im ersten Schritt wird der Aufruf `F(1,0)` expandiert. Dabei wird er durch den Rumpf der Rechenvorschrift `F` ersetzt, wobei die Variablensymbole `x` und `y` durch `1` beziehungsweise `0` ersetzt werden. Die inneren Redeces im so entstandenen Ausdruck sind `1<=0`, `1-1` sowie `1-0`.

Von ihnen wird im zweiten Schritt der am weitesten links gelegene Ausdruck `1<=0` zu `false` vereinfacht. Nun ist der Bedingungsausdruck der Verzweigung ausgewertet und die Verzweigung wird damit zu einem Redex. Weil sie am weitesten links gelegen ist, wird sie im dritten Schritt vereinfacht. Nun verbleiben die inneren Redeces `1-1` und `1-0`, die in den nächsten beiden Schritten von links ausgehend vereinfacht werden.

Bislang wurde der Ausdruck `F(1,0)` zu `F(0,F(1,0))` ausgewertet (siehe rechtes Fenster in Abb. 5.13). Man erkennt, dass im letzten Ausdruck der anfängliche Ausdruck enthalten ist, und zwar als einziger innerer Redex – das KIEL-System würde also der Auswertungsstrategie folgend wieder versuchen, den Teilausdruck `F(1,0)` auszuwerten und dabei unweigerlich abermals bei `F(0,F(1,0))` landen.

Die Rekursion terminiert also niemals und der Ausdruck ist bei Auswertung nach der Leftmost-Innermost-Strategie undefiniert.

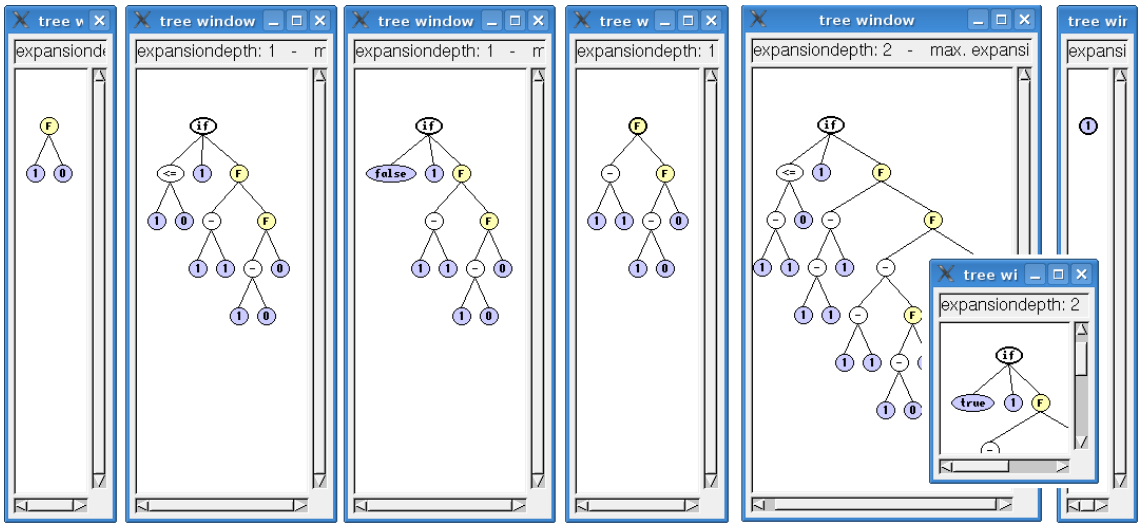


Abbildung 5.14: Auswertung `F(1,0)` nach Leftmost-Outermost-Strategie,

5.3.3 Leftmost-Outermost-Auswertung

Mit Hilfe des Undo/Redo-Fensters wird das Baumfenster nun wieder in den Urzustand `F(1,0)` gebracht. Dann wählt man im Auswertungsfenster die Auswertungsstrategie Leftmost-Outermost an.

Im linken Fenster in Abb. 5.14 ist die Baumdarstellung des ursprünglichen Ausdrucks `F(1,0)` zu sehen. Da nur ein Redex vorhanden ist, wird auch hier zunächst der Aufruf von `F` expandiert. Im zweiten Schritt wird von den Redeces `1<=0`, `1-1` und `1-0` der am weitesten links gelegene vereinfacht: Die Bedingung der Verzweigung `1<=0` wird durch `false` ersetzt. Im dritten Schritt kann nun die Verzweigung vereinfacht werden.

Bislang unterscheiden sich die beiden Auswertungsstrategien nicht. Im aktuellen Ausdruck `F(1-1,F(1-0,0))` allerdings wird es interessant: Sowohl der gesamte Ausdruck `F(1-1,F(1-0,0))` als auch die Teilausdrücke `1-1`, `F(1-0,0)` und `1-0` sind durchweg Redeces, die vereinfacht oder expandiert werden könnten.

Bei der zuvor aufgezeigten Auswertung nach der Leftmost-Innermost Strategie dürfen nur die inneren Redeces berücksichtigt werden, also genau diejenigen, die keine weitere Redeces enthalten. Dies sind nur `1-1` und `1-0`. Von diesen wurde nach der Leftmost-Innermost-Strategie der am weitesten links gelegene Redex `1-1` gewählt.

Bei der Auswertung nach der Leftmost-Outermost-Strategie besteht diese Beschränkung nicht: Von allen Redeces wird der am weitesten links gelegene Redex ausgewählt. In diesem Fall ist dies der äußerste Aufruf von `F`. Das Ergebnis der nun folgenden Expansion ist im fünften Fenster in 5.14 zu sehen. In diesem großen Ausdruck ist der am weitesten links gelegene Redex in der Bedingung `1-1<=0` der Verzweigung zu finden.

Zunächst wird der Teilausdruck `1-1` zu `0` und dann die restliche Bedingung `0<=0` zu `true` vereinfacht. Damit kann auch die Verzweigung vereinfacht werden und das Ergebnis der Auswertung ist `1`.

5.3.4 Effizienz der Strategien

Mit Hilfe des KIEL-Systems kann die Anzahl der für die Auswertung eines Ausdrucks benötigten Expansions- und Vereinfachungsschritte gezählt werden. Dies soll genutzt werden, um die Effizienz der Leftmost-Innermost- und der Leftmost-Outermost-Strategie bei der Auswertung verschiedener Aufrufe der Morris-Funktion `F` zu vergleichen.

Zunächst müssen die gewünschten Ausdrücke `F(5,1)`, `F(5,5)`, `F(5,10)`, `F(10,1)` sowie `F(10,5)` und `F(10,10)` im Ausdruckseingabefenster eingegeben werden. Dann stehen sie im Ausdrucksfenster zur Auswahl bereit.

Jeder Ausdruck wird nun einzeln angewählt, so dass er im Baumfenster dargestellt wird. Durch Klick auf den Wurzelknoten wird dann die komplette Auswertung gestartet. Nach Abschluss der Auswertung kann im oberen Teil des Baumfensters die Anzahl der Expansions- und Vereinfachungsschritte abgelesen werden. Dieser Prozess wird einmal für die Leftmost-Outermost- und einmal für die Leftmost-Innermost-Strategie durchgeführt. Die Anzahl der benötigten Schritte wird in der folgenden Tabelle dargestellt:

	Leftmost-Innermost		Leftmost-Outermost	
	Expansionen	Vereinfachungen	Expansionen	Vereinfachungen
`F(5,1)`	63	188	6	27
`F(5,5)`	33	98	6	27
`F(5,10)`	33	98	6	27
`F(10,1)`	2047	6140	11	77
`F(10,5)`	1057	3170	11	77
`F(10,10)`	1025	3074	11	77

Tabelle 5.3: Erforderliche Schritte für die Auswertung von `F`

Wie man erkennt, erfolgt die Auswertung von `F` nach der Leftmost-Outermost-Strategie deutlich effizienter als nach der Leftmost-Innermost-Strategie. Die Anzahl der erforderlichen Auswertungsschritte ist bei der Leftmost-Outermost-Strategie nur vom ersten Argument abhängig.

5.4 Verschattung bei lokalen Wertedeklarationen

Durch die Einführung von lokalen Wertedeklaration werden Nutzer des KIEL-Systems mit einem neuen Problem konfrontiert: Weil jede lokale Wertedeklaration ein ganz normaler Ausdruck ist, können im Rumpf von lokalen Wertedeklarationen oder Rechenvorschriften weitere lokale Wertedeklarationen enthalten sein. Wenn nun in diesen geschachtelten Wertedeklarationen einem bereits eingeführten Variablensymbol Werte zugewiesen werden, stellt sich die Frage, wie derartige Ausdrücke ausgewertet werden sollen.

Im KIEL-System ist für eine Variable `x`, die entweder in einer lokalen Wertedeklaration mit Rumpf `t` oder als formaler Parameter einer Rechenvorschrift mit dem Rumpf `t` eingeführt wurde, der sogenannte Gültigkeitsbereich (auch Sichtbarkeitsbereich) als Zeichenreihe dadurch gegeben, dass man im Rumpf `t` alle lokalen Wertedeklarationen, in denen `x` nochmals deklariert wird, mit Ausnahme der rechten Seite der jeweiligen Deklaration entfernt.

In Standard ML müssen daneben noch die Fälle, dass `x` als Rechenvorschrift, als Typ mit Konstruktor oder als Typabkürzung deklariert wird, unterschieden werden. Im KIEL-System jedoch werden nur lokale Wertedeklarationen unterstützt und diese Unterscheidung entfällt.

Ein einfaches Programm, dessen Quelltext nachfolgend angegeben wird, soll zur Verdeutlichung dieses Prinzips dienen:

```
fun f(x:int) : int =
    let val (x:int) = x+1 in
        let val (x:int) = x*2 in
          x+x
        end
    end;
```

Die Rechenvorschrift `f` hat einen formalen Parameter mit dem Bezeichner `x`. Daneben werden in zwei geschachtelten lokalen Wertedeklarationen der Variablen `x` Werte zugewiesen. Ein Programm mit dieser Rechenvorschrift ist in der Datei `verschattung.prg` in der Distribution des KIEL-Systems enthalten.

Nachdem das Programm geladen wurde, stellt man im Auswertungsfenster die Auswertungsstrategie auf Leftmost-Innermost. Nun muss der Ausdruck `f(2)` eingegeben und im Ausdrucksfenster angewählt werden.

Danach wird im Baumfenster der ursprüngliche Ausdruck als Baum angezeigt, wie im

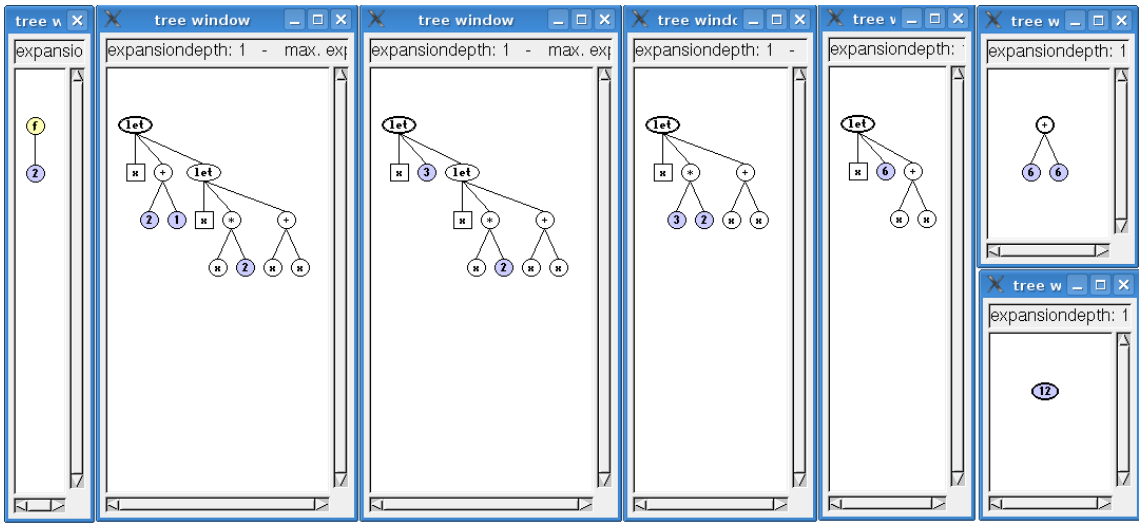


Abbildung 5.15: Verschattung von Variablen bei lokalen Wertedeklarationen

linken Fenster in Abb. 5.15 auf der nächsten Seite dargestellt. Im ersten Auswertungsschritt muss der Aufruf von `f(2)` expandiert werden. Bei diese Expansion wird die Variable `x` an das Argument `2` gebunden und dann der Aufruf durch den Rumpf der Rechenvorschrift ersetzt. Der Gültigkeitsbereich dieser Variable ist allerdings nicht der gesamte Rumpf

$$\texttt{let val x : int = x + 1 in let val x : int = x * 2 in x + x end end},$$

in dem das Symbol `x` insgesamt sechs Mal vorkommt. Weil bereits in der äußeren lokalen Wertedeklaration eine Variable `x` deklariert wird, gehört sie – bis auf die rechte Seite der Deklaration – nicht zum Gültigkeitsbereich. Damit liegt nur ein `x` im Gültigkeitsbereich der gebundenen Variable und wird bei der Expansion der Rechenvorschrift ersetzt. Im folgenden Ausdruck ist dieses `x` durch einen Pfeil markiert:

$$\begin{array}{l}\texttt{let val x : int = x + 1 in let val x : int = x * 2 in x + x end end.}\\ \qquad\qquad\qquad\quad\ \uparrow\end{array}$$

Der Ausdruck nach erfolgter Expansion wird im zweiten Fenster der Abb. 5.15 dargestellt. Im zweiten Auswertungsschritt wird dann der innere Ausdruck `2+1` zu `3` vereinfacht.

Im nächsten Schritt soll nun die äußere lokale Wertedeklaration expandiert werden. Die rechte Seite der Wertedeklaration wurde im letzten Schritt zu `3` ausgewertet – dieser Wert soll nun der Variable `x` zugewiesen werden. Im Rumpf der äußeren lokalen Wertedeklaration

$$\texttt{let val x : int = x * 2 in x + x end}$$

kommt das Variablensymbol `x` viermal vor. Allerdings liegen nicht alle Vorkommnisse im Gültigkeitsbereich der äußeren lokalen Wertedeklaration. In der inneren lokalen Wertedeklaration wird `x` ebenfalls deklariert. Daher gehört dieser Abschnitt – abgesehen von der rechten Seite der Wertedeklaration – nicht zum Gültigkeitsbereich. Damit kommt das Variablensymbol `x` nur einmal im Gültigkeitsbereich vor, nämlich im rechten Teil der inneren Wertedeklaration. Im folgenden Ausdruck ist dieses `x` mit einem Pfeil markiert:

```
let val x : int =  x  * 2 in x + x end.
                   ↑
```

Deshalb wird bei der Expansion der lokalen Wertedeklaration nur dieses eine `x` durch den Wert `3` ersetzt, die anderen `x` bleiben unangetastet. Im vierten Fenster in der Abb. 5.15 auf der vorherigen Seite wird das Resultat dieser Expansion dargestellt.

Im nächsten Auswertungsschritt wird der innere Ausdruck `3*2` zu `6` vereinfacht. Danach kann die verbleibende lokale Wertedeklaration expandiert werden. Auch hier wird dem Variablensymbol `x` ein Wert zugewiesen. Diesmal enthält der Rumpf der lokalen Deklaration allerdings keine weiteren lokalen Wertedeklarationen. Daher werden bei der Expansion sämtliche Vorkommnisse von `x` im Rumpf durch den Wert `6` ersetzt. Das Ergebnis dieser Expansion kann man in der Abbildung 5.15 im Fenster rechts oben begutachten.

Nun wird im letzten Schritt nur noch der Ausdruck `6+6` zu `12` vereinfacht und die Auswertung ist abgeschlossen.

5.5 Fehlersuche

Ein weiterer Nutzen des KIEL-Systems ist die einfache Suche nach Fehlern in Programmen. Das KIEL-System kann als sogenannter Debugger für ML Programme benutzt werden. Dabei ist die schrittweise Auswertung von Ausdrücken ein wichtiges Hilfsmittel.

Als *Debugger* wird eine Software bezeichnet, die dazu dient, andere Programme zu testen und Fehler in ihnen zu finden. Der Debugger kontrolliert die Ausführung eines anderen Programms: Falls ein Fehler auftritt, kann der Nutzer den Zustand des Gesamtsystems inspizieren und zum Beispiel die aktuelle Belegung von Variablen und die aktuell ausgeführte Programmzeile einsehen. Fortschrittliche Debugger ermöglichen darüber hinaus die schrittweise Ausführung von Programmen: Jede Anweisung kann einzeln ausgeführt und der Zustand des Speichers vorher und hinterher eingesehen werden.

In diesem Sinne kann auch das KIEL-System als Debugger dienen – schließlich ist die schrittweise Auswertung von Ausdrücken und Programmen der erklärte Hauptzweck der Anwendung.

```
fun merge (nil: int list, t: int list) : int list =
    t
  | merge (s:int list, nil: int list) : int list =
   s
  | merge (x::xs: int list, y::yt: int list) : int list =
    if x<=y then x :: merge(xs,y::yt)
            else y :: merge(x::xs,yt);


fun split (nil: int list) : int list * int list =
    (nil,nil)
  | split ([x]: int list) : int list * int list =
    ([x], nil)
  | split (x::y::ys: int list) : int list * int list =
    let val (u: int list, v: int list) = split(ys)
    in  (x::u, y::v)
    end;

fun sort (nil: int list) : int list =
    nil
  | sort (s: int list) : int list =
    let val (u: int list, v: int list) = split(s)
    in  merge(sort(u),sort(v))
    end;
```

Abbildung 5.16: fehlerhafte Implementierung von Mergesort

Um aufzuzeigen, wie das KIEL-System bei der Fehlersuche helfen kann, wird in [5] ein einfaches, aber fehlerhaftes Programm als Beispiel angegeben.

Der in Abb. 5.16 dargestellte Quelltext soll den Sortieralgorithmus Mergesort für eine Liste von ganzen Zahlen implementieren.

Mergesort sortiert eine Liste von Elementen nach dem Divide-And-Conquer-Prinzip: Die Liste wird in zwei Listen mit einer geringeren Anzahl von Elementen aufgeteilt, die jeweils durch einen rekursiven Aufruf der Mergesort-Routine sortiert werden. Diese zwei sortierten Listen werden dann zum Ergebnis – einer sortierten Liste mit allen Elementen – zusammengefügt.

Im angegebenen Quelltext vereint die Rechenvorschrift `merge` zwei vorsortierte Listen zu einer einzigen sortierten Liste. Die Rechenvorschrift `split` teilt eine Liste in zwei etwa gleichgroße Listen auf. Die Rechenvorschrift `sort` schließlich sortiert eine Liste, indem sie sie zunächst mit Hilfe von `split` in zwei Listen aufteilt, diese durch rekursive Aufrufe ihrer selbst sortiert und dann die vorsortierten Listen durch `merge` wieder zu einer sortierten Liste zusammenfügt.

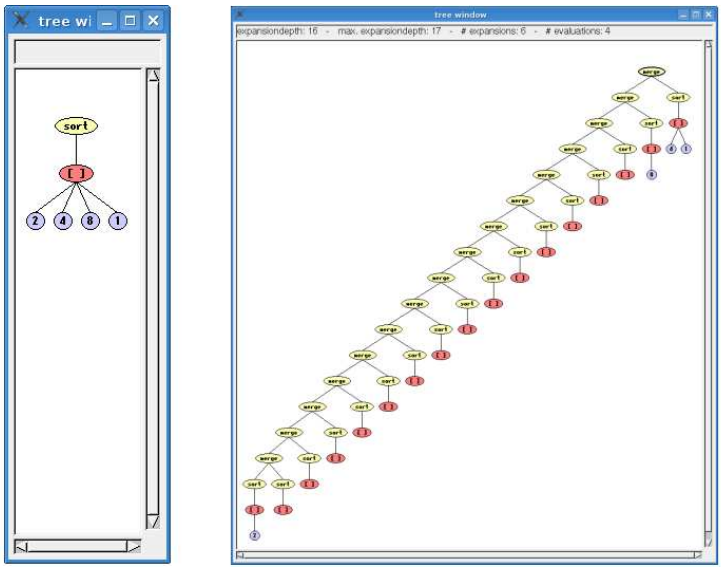


Abbildung 5.17: Auswertung von `sort[2,4,8,1]` terminiert nicht

Um das angegebene Programm mit einer einfachen Liste von ganzen Zahlen zu testen, muss zunächst eine Programmdatei mit den Rechenvorschriften geladen werden. Der Quelltext ist in der Distribution des KIEL-Systems als `mergesort_fehlerhaft.prg` enthalten. Sodann kann im Auswertungsfenster die Leftmost-Innermost-Strategie ausgewählt werden. Dort wird auch festgelegt, dass keine schrittweise, sondern eine komplette Auswertung erfolgen soll. Über das Ausdruckseingabefenster wird nun der Ausdruck

```
sort([2,4,8,1])
```

eingegeben und nach Auswahl im Ausdrucksfenster im Baumfenster dargestellt (linkes Fenster in Abb. 5.17).

Durch einen Klick auf den Ursprungsknoten kann nun die Auswertung gestartet werden. Man erwartet, innerhalb weniger Sekunden das Ergebnis `[1,2,4,8]` zu erhalten. Dies ist aber nicht der Fall: Es wird kein Ergebnis angezeigt. Offensichtlich terminiert die Auswertung nicht. Daher muss die Berechnung durch einen Klick auf die entsprechende Schaltfläche im Auswertungsfenster abgebrochen werden. Im Baumfenster wird nun ein Baum angezeigt (siehe rechtes Fenster in Abb. 5.17), der den unendlich wachsenden Ausdruck

```
merge(merge(merge(...,sort(nil)),sort(nil)),sort(nil))
```

darstellt. Um den Grund zu finden, warum die Auswertung nicht terminiert, kann man die Funktion des KIEL-Systems nutzen, eine Auswertung schrittweise durchzuführen.

Mit Hilfe des Undo/Redo-Fensters kann der Ausdruck wieder in den Urzustand zu-

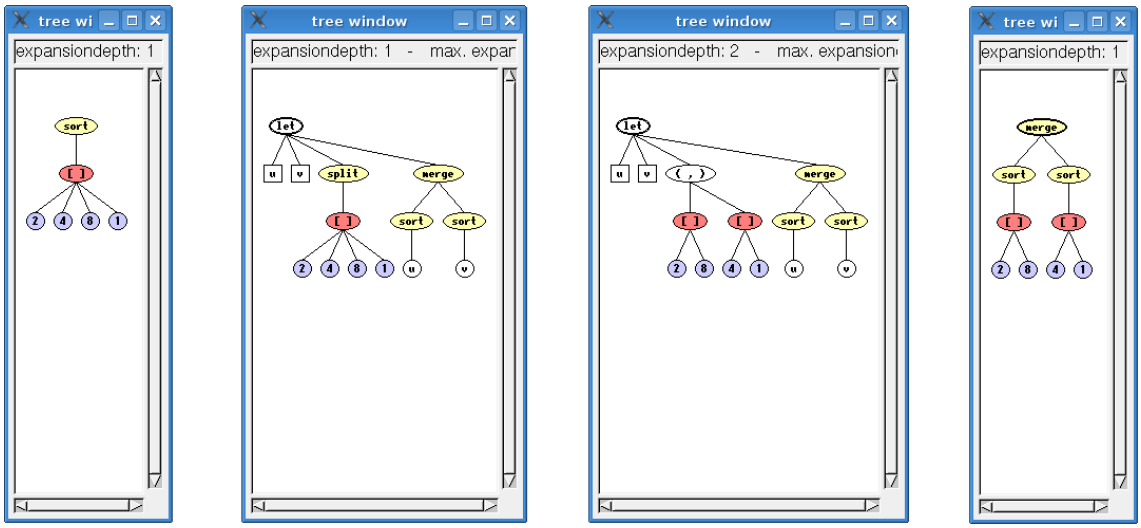


Abbildung 5.18: schrittweise Auswertung von `sort[2,4,8,1]`, Teil 1

rück gebracht werden (linkes Fenster in Abb. 5.18). Nun muss im Auswertungsfenster die schrittweise Auswertung von Ausdrücken angewählt werden. Durch Linksklick auf den Ursprungsknoten im Baumfenster kann jetzt jeweils die Auswertung des nächsten Schritts gestartet und beobachtet werden.

Im ersten Schritt wird der Aufruf von `sort` expandiert und der resultierende Ausdruck entspricht dem Rumpf der Rechenvorschrift. Nun muss zunächst der Aufruf von `split` mit dem Argument `[2,4,8,1]` ausgewertet werden – die einzelnen Schritte dieser Auswertung werden hier allerdings nicht aufgezeigt. Wie im dritten Fenster dargestellt, wird die Liste von `split` in die beiden Listen `[2,8]` und `[4,1]` aufgeteilt.

Nun kann die lokalen Wertedeklaration expandiert werden und es verbleibt die Aufgabe, die beiden Listen `[2,8]` und `[4,1]` zu sortieren und die Ergebnisse zu einer Liste zu vereinen (rechtes Fenster in Abb. 5.18).

Weil die Auswertung nach der Leftmost-Innermost-Strategie erfolgt, muss im nächsten Schritt zunächst die Liste `[2,8]` sortiert werden.

Wie im linken Fenster in Abb. 5.19 auf der nächsten Seite zu sehen ist, wurde das Sortieren der Liste `[2,8]` nach Expansion von `sort` und Aufteilen in zwei Listen korrekterweise auf das Sortieren des Listen `[2]` und `[8]` sowie das anschließende Vereinen der sortierten Listen zurückgeführt.

Im nächsten Schritt muss also die Liste `[2]` sortiert werden. Auch hier sollen die einzelnen Schritte der Expansion von `sort` und des Aufteilens der Liste durch `split` übersprungen werden. Wie im nächsten Fenster dargestellt, kann man dann allerdings feststellen, dass der Teilausdruck `sort([2])` durch `merge( sort([2]), sort([]))` ersetzt wurde. Die einelementige Liste wurde also in die zwei Listen `[2]` und `[]` ge-

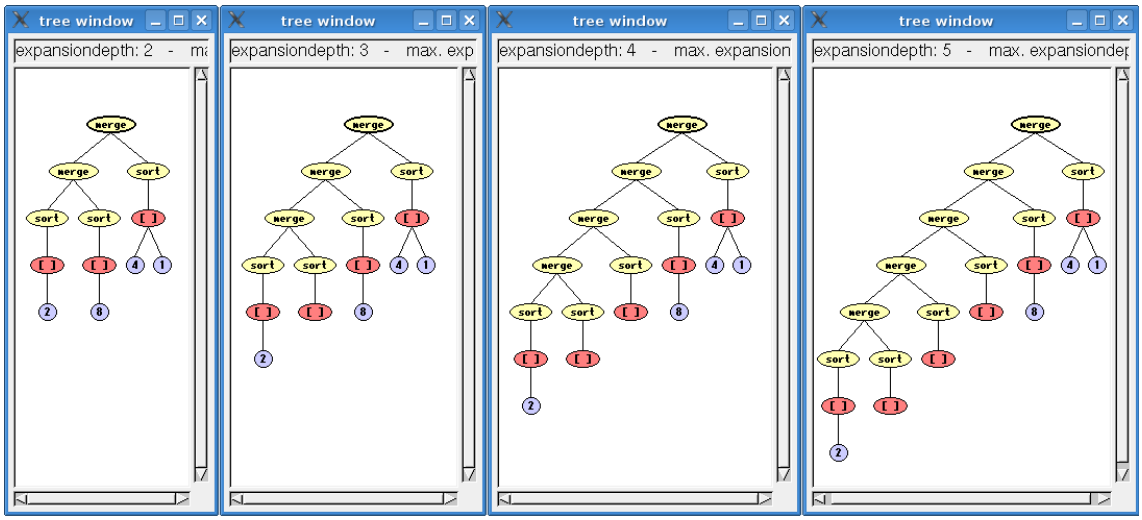


Abbildung 5.19: schrittweise Auswertung von `sort[2,4,8,1]`, Teil 2

teilt, die nun wieder einzeln sortiert und dann zusammengefügt werden sollen. Dies führt offensichtlich zu sich immer wiederholenden Aufrufen von `sort([2])` und die Auswertung terminiert nie, wie man in den restlichen beiden Bildern sehen kann.

Damit konnte der Fehler eingegrenzt werden: Die Rechenvorschrift `sort` kann offensichtlich keine einelementigen Listen sortieren, da jede nichtleere Liste `s` in zwei Listen `u=s` und `v=nil` geteilt wird. Weil `u` und `v` wieder sortiert werden, führt dies bei `u` zu einer niemals endenden Rekursion.

Dieser Fehler ist einfach zu beheben, weil jede einelementige Liste bereits sortiert ist und von `sort` direkt als Ergebnis zurückgegeben werden kann.

Es gibt zwei Möglichkeiten, dies umzusetzen: Der Fall einer einelementigen Liste kann über die Mustererkennung abgefangen werden, wie im Quelltext in Abb. 5.20 zu sehen. Die zweite Möglichkeit besteht darin, im zweiten Fall der Rechenvorschrift `sort` eine Verzweigung einzufügen, die den Fall der einelementigen Liste berücksichtigt, wie in Abb. 5.21 dargestellt.

```
fun sort (nil: int list) : int list =
    nil
  | sort ([x]: int list ) : int list =
    [x]
  | sort (s: int list) : int list =
    let val (u: int list, v: int list) = split(s)
    in  merge(sort(u),sort(v))
    end;
```

Abbildung 5.20: korrigierte Fassung von `sort` (Mustererkennung)

```
fun sort (nil: int list) : int list =
    nil
  | sort (s: int list) : int list =
    if length(s) = 1 then s
                     else let val (u: int list, v: int list) =
                                                        split(s)
                          in  merge(sort(u),sort(v))
                     end;
```

Abbildung 5.21: korrigierte Fassung von `sort` (Verzweigung)

6 Implementierungsdetails

Nachdem bislang die Grundlagen, die Bedienung sowie einige beispielhafte Anwendungen des erweiterten KIEL-System vorgestellt wurden, soll in diesem Kapitel ein Einblick in die Struktur des KIEL-Systems gegeben werden.

Zunächst wird der Aufbau des gesamten Systems und seine Unterteilung in verschiedene Komponenten dargestellt. Die sechs Bereiche Hauptprogramm, interne Datenstrukturen, Typenüberprüfung, Auswertung, Darstellung und Sonstiges werden vorgestellt und ihre wichtigsten Funktionen erläutert.

Dann wird detailliert auf die im Rahmen dieser Arbeit vorgenommenen Erweiterungen eingegangen. Dabei werden für alle Erweiterungen zunächst die neu eingeführten internen Datenstrukturen und dann die erforderlichen Änderungen an den bestehenden Bereichen des KIEL-Systems beschrieben.

6.1 Komponenten

Auch wenn das KIEL-System ein verhältnismäßig kleines Softwareprojekt ist und aktuell deutlich unter 40.000 Zeilen Quelltext[1] enthält, ist es trotzdem nötig, das gesamte System in einzelne Komponenten mit spezifischen Aufgaben zu unterteilen, um die Übersicht zu wahren.

Insgesamt wurde das Projekt in die sechs Bereiche H*auptprogramm, interne Datenstrukturen, Typüberprüfung, Auswertung, Darstellung* und *Sonstiges* aufgeteilt.

In jedem Bereich ist der Quelltext noch einmal feiner in mehrere Dateien aufgeteilt. Alle Dateien eines Bereichs sind in einem eigenen Unterverzeichnis zusammengefasst.

In diesem Abschnitt werden nachfolgend alle Bereiche und ihre Funktionsweise kurz skizziert.

6.1.1 Hauptprogramm (Main)

Das Hauptprogramm hat lediglich die Aufgabe, die Kommandozeilenparameter einzulesen und die grafische Benutzeroberfläche zu starten. Entsprechend konnte auch die ge-

[1] genau: 31496 Zeilen, berechnet mit dem Unix-Tool wc und dem Aufruf "wc */*.c */*.h */*.y */*.l"

samte Funktionalität dieses Bereichs in nur einer einzigen Quelltextdatei `kiel_main.c` implementiert werden.

Daneben werden allerdings noch zwei Konfigurationsdateien in diesem Bereich abgelegt. In diesen beiden Dateien werden Einstellungen für die grafische Benutzeroberfläche (zum Beispiel Größe und Position der Fenster) vorgegeben.

Folgende Dateien gehören zum Hauptprogramm:

- `kiel_main.c`
- `kiel.rc`
- `kiel.rc.gtk`

6.1.2 Interne Datenstrukturen (Syntaxtree)

Das KIEL-System verarbeitet Ausdrücke, die als Text eingegeben werden. Diese werden nach einer lexikalischen und syntaktischen Analyse in einer effizienten Baumstruktur abgespeichert. Alle Auswertungen und Überprüfungen eines Ausdrucks werden an dieser Baumstruktur ausgeführt und nicht an den anfangs vorliegenden Zeichenreihen.

Alle entsprechenden Datenstrukturen und Routinen, die zu Manipulation dieser Datenstrukturen dienen, sind im Bereich Syntaxtree zusammengefasst.

Zunächst soll die grundlegende Repräsentation des Syntaxbaums im Speicher dargestellt werden. Dann wird der Parser erklärt, der diesen Baum aus der textuellen Angabe eines Ausdrucks oder Programms erzeugt.

6.1.2.1 Syntaxbaum

Der Syntaxbaum ist im Hauptspeicher als verknüpfte Menge von Datenstrukturen des in der Datei `syntaxtree.h` definierten Typs `Expr_node_struct` abgelegt.

Die Verknüpfungen des Baumes spiegeln die Struktur des Terms in kanonischer Form wider. Dabei wird jeder Grundbaustein eines Terms (Konstanten, Operatoren, Verzweigungen, Aufrufe von Rechenvorschriften, lokale Wertedeklarationen, ...) durch einen Knoten repräsentiert. Jedem Knoten ist eine Knotenart (Basisoperation, Verzweigung, Konstante, ...) und ein Typ aus dem KIEL-Typsystem (`real`, `int`, `string list list`, ...) zugewiesen.

Jeder Knoten im Syntaxbaum hat maximal drei direkte Nachfolgeknoten, die einzeln ansprechbar sind und die je nach Knotenart unterschiedliche Bedeutung haben.

Konstanten oder Variablensymbole werden durch Knoten ohne Nachfolger repräsentiert.

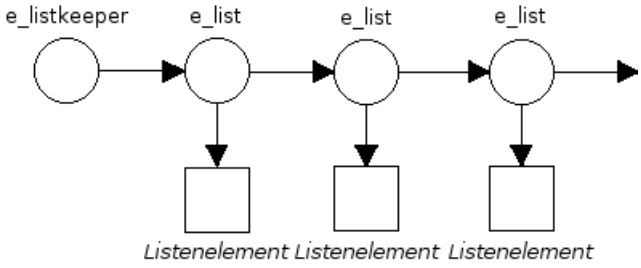


Abbildung 6.1: interne Struktur einer Liste

Aufrufe von Basisoperatoren und -funktionen werden durch Knoten repräsentiert, in deren Nachfolgeknoten die Operanden oder Argumente direkt hinterlegt sind.

Bei Verzweigungen wird im ersten Nachfolgeknoten der Teilbaum für die Bedingung abgelegt, in den beiden anderen die Teilbäume für die zwei Fälle der Verzweigung.

Weil Listen unbegrenzt viele Elemente haben dürfen, können nicht alle diese Elemente direkt mit dem Listen-Knoten verknüpft sein. Stattdessen werden Listen als verkettete Liste von Knoten verwaltet:

Jede Liste wird intern als ein Knoten der Knotenart `e_listkeeper` dargestellt, an den hintereinander für jedes Listenelement ein Knoten der Knotenart `e_list` in Form einer verketteten Liste angehängt ist. Jeder `e_list` Knoten verweist dabei auf einen Knoten mit dem jeweiligen Element der Liste. Wenn es noch weitere Listenelemente gibt, ist er zusätzlich auch mit dem nachfolgenden `e_list` Knoten verknüpft (siehe Abb. 6.1).

Diese Art von verketteten Listen wird auch bei Tupeln, Argumentlisten und ähnlichen Strukturen genutzt: Bei lokalen Wertedeklarationen enthält zum Beispiel der erste Nachfolger die Liste der deklarierten Variablen als verkettete Liste, der zweite Nachfolger den Teilbaum, dessen Wert an die Variablen gebunden werden soll, und der dritte Nachfolger den Rumpf.

Bei Aufrufen von Rechenvorschriften werden die Argumente des Aufrufs als verkettete Liste im ersten Nachfolgeknoten abgelegt.

Die Routinen für den Aufbau und die Verwaltung des Syntaxbaumes sind in folgenden Dateien zusammengefasst:

- `prg.h`
- `expr.h`
- `symboltable.[ch]`
- `syntaxtree.[ch]`

Der interne Syntaxbaum darf nicht mit dem Baum verwechselt werden, der dem Anwender des KIEL-Systems präsentiert wird. Zwar ist jeder Knoten im dargestellten Baum auch ein Knoten im internen Syntaxbaum, dieser interne Baum enthält aber noch zusätzliche Hilfsknoten.

So sind zum Beispiel die Knoten der Art `e_list` nicht sichtbar. Für die Darstellung eines Knotens werden zusätzliche Informationen wie die Position am Bildschirm in einem speziell dafür vorgesehenen Teil der entsprechenden Datenstruktur abgespeichert.

6.1.2.2 Parser

Der Parser des KIEL-Systems wurde mit Hilfe der beiden Tools *lex* und *yacc* [11] umgesetzt. Diese beiden Programme erzeugen aus Konfigurationsdateien Quelltext in der Programmiersprache C für das Scannen und Parsen von Textdateien. Mit *lex* wird ein Scanner erzeugt, der den eingelesenen Text in einzelne lexikalische Elemente unterteilt. Mit *yacc* wird ein Parser erzeugt, der diese Elemente syntaktisch erkennt und in die im letzten Abschnitt eingeführte interne Baumstruktur überführt. Während der syntaktischen Analyse wird auch die Sichtbarkeit von Symbolen überprüft.

Weil sich das Verarbeiten von Programmen und einzelnen Ausdrücken nur leicht unterscheidet, existiert nur ein Satz von Konfigurationsdateien für lex und yacc. Mit Hilfe des Tools *sed* [12] wird aus der Konfigurationsdatei für das Verarbeiten von Programmen automatisch eine Konfigurationsdatei für das Verarbeiten von Ausdrücken erstellt.

Die Routinen für das Verarbeiten von Programmen und Ausdrücken sind in folgenden Dateien zusammengefasst:

- `prg_scanner.l`
- `prg_parser.y`
- `st_io.[ch]`
- `yyval_types.h`

6.1.3 Typüberprüfung (Typechecker)

Nach der lexikalischen und syntaktischen Analyse führt das KIEL-System eine Typüberprüfung durch. Diese stellt sicher, dass die Operanden der Basisoperatoren sowie die Argumente von Basisfunktionen, Konstruktoren und Rechenvorschriften einen passenden Typ haben.

Die Routinen für die Typüberprüfung von Programmen und Ausdrücken sind in folgenden Dateien zusammengefasst:

- `typechecker.[ch]`
- `tc_body.[ch]`
- `tc_util.[ch]`

Die Überprüfung von Programmen erfolgt in zwei Schritten: Zunächst wird für jede deklarierte Rechenvorschriften die Liste der Parameter überprüft und die Typen der auftretenden Variablen werden bestimmt. Danach wird der Rumpf der Rechenvorschrift auf Konsistenz und Korrektheit der Typen überprüft. Bei der Typüberprüfung einzelner Ausdrücke entfällt die Überprüfung der Parameter.

Der Kern der Typüberprüfung ist die rekursive Funktion `checkBody` in der Datei `tc_body.c`. Diese gibt den Typ eines Knotens zurück und überprüft dabei auch, ob die Typen der Nachfolgeknoten zu den erwarteten Typen kompatibel sind.

Dazu wird je nach Knotenart ein unterschiedliches Verfahren verwendet. Im Allgemeinen werden zunächst mit einem rekursiven Aufruf von `checkBody` die Typen der Nachfolgeknoten bestimmt.

Dann wird mit Hilfe der Funktion `isFoundTypeCompatibleToExpected` aus der Datei `tc_util.c` geprüft, ob die Typen zu den erwarteten Typen passen. Wenn zum Beispiel ein Knoten geprüft wird, der für die Anwendung der Basisoperation "`::`" zur Verknüpfung von Listen steht, dann wird mit Hilfe der Funktion `isFoundTypeCompatibleToExpected` getestet, ob die beiden Argumente zueinander passende Listentypen haben.

Im nächsten Schritt wird der Typ des Knoten selbst bestimmt.

Bei Basisoperatoren und -funktionen mit feststehender Funktionalität ist der Typ klar. So ist der Typ eines Knotens für die Anwendung des Basisoperators "`/`" für die Division von zwei Maschinenzahlen offensichtlich `real`.

Bei anderen Knoten ist es nicht so einfach. Für Verzweigungen der Form `if b then e else f` hat der Knoten einen Typ, der sowohl zu `e` als auch zu `f` kompatibel sein muss. Falls `e` und `f` als Typ die gleiche Basissorte haben, dann ist diese Sorte auch der Typ des Knotens. Falls `e` und `f` aber Ausdrücke eines komplexen Typs sind (Listen oder Tupel), dann muss ein Typ gefunden werden, der zu beiden Typen kompatibel ist. Diese Aufgabe erledigt die Funktion `getCompatibleType` aus der Datei `tc_util.c`.

6.1.4 Auswertung (Evaluator)

Die Funktionen zur Auswertung von Ausdrücken sind im Verzeichnis Evaluator zusammengefasst. Sie sind auf folgende Dateien verteilt:

- `evaluator.[ch]`
- `eval_simplify.[ch]`
- `eval_expand.[ch]`
- `pattern_matcher.[ch]`
- `eval_global.[ch]`
- `ml_limits.[ch]`

Dabei werden in `evaluator.c` die drei verschiedenen Auswertungsstrategien Leftmost-Innermost, Leftmost-Outermost und Full implementiert. Diese Auswertungsstrategien rufen Funktionen in `eval_simplify.c` und `eval_expand.c` auf, die Teilbäume vereinfachen beziehungsweise Aufrufe von Rechenvorschriften oder lokale Wertedeklarationen expandieren.

6.1.5 Darstellung (GTK)

Die grafische Benutzeroberfläche ist mit Hilfe des GTK-Toolkits implementiert. Diese Bibliothek war zum Zeitpunkt der Entwicklung der ersten Version des KIEL-Systems die am weitesten verbreitete frei verfügbare Bibliothek ihrer Art. Sie bildet auch die Basis der beliebten Desktop-Umgebung Gnome.

Da jedes Fenster der Benutzeroberfläche in einer einzelnen Datei im Verzeichnis `GTK` abgelegt ist, soll hier von einer Auflistung sämtlicher Dateien abgesehen werden.

Hervorzuheben sind allerdings die zwei Dateien `expr_node_misc.c` und `expr_node_misc.h`. Hier wird die Datenstruktur definiert, die die für die Darstellung eines Knotens am Bildschirm notwendigen Informationen – zum Beispiel die aktuelle Position – enthält.

In den Dateien `drawable_expression.c` und `drawable_expression.h` sind die Routinen zusammengefasst, die Knoten am Bildschirm positionieren und auf einer GTK-Zeichenfläche anzeigen.

6.2 Erweiterungen des unterstützten Sprachumfangs

Der vom KIEL-System unterstützte Sprachumfang wurde im Rahmen dieser Diplomarbeit vor allem um Produkttypen und lokale Wertedeklarationen erweitert.

Im folgenden Abschnitt sollen zunächst für Produkttypen und dann für lokale Wertedeklarationen die nötigen Änderungen im Quelltext des KIEL-Systems beschrieben werden. Dabei werden jeweils nacheinander die in den soeben vorgestellten Komponenten des KIEL-Systems vorgenommenen Änderungen einzeln abgehandelt.

6.2.1 Produkttypen

Um das KIEL-System um Produkttypen zu erweitern, mussten in den fünf Bereichen Syntaxtree, Parser, Typechecker, Evaluator und Darstellung Veränderungen vorgenommen werden.

Die erforderlichen Anpassungen werden nachfolgend beschrieben.

6.2.1.1 Syntaxtree / Interne Repräsentation

Ein Tupel wird vom KIEL-System ähnlich behandelt wie eine Liste. Diese wird als verkettete Liste von Knoten der Knotenart `e_list` implementiert.

Für Tupel wird genau das gleiche Prinzip angewendet, nur dass der oberste Knoten wie in Abb. 6.2 dargestellt die Knotenart `e_product` statt `e_listkeeper` hat. Diese Knotenart wird in der Header-Datei `syntaxtree.h` neu eingeführt.

Durch die teilweise Übernahme der internen Repräsentation können viele Routinen für die Darstellung und Auswertung von Listen auch für die Darstellung und Auswertung von Tupel genutzt werden.

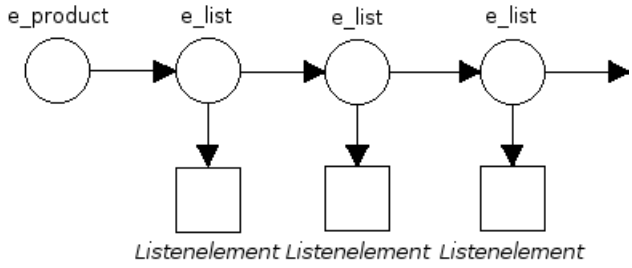


Abbildung 6.2: interne Struktur eines Tupels

6.2.1.2 Parser

Wie bereits beschrieben, ist der Parser des KIEL-Systems mit Hilfe von *lex* und *yacc* umgesetzt worden. Daher war es möglich, ihn um die Erkennung der neuen Syntax zu erweitern.

```
product_type : type { ... }
             | type MULT product_type { ... }
             ;
```

Abbildung 6.3: Produktionen für das Nonterminal `product_type`

```
type : UNIT      { ... }
     | BOOL      { ... }
     | INT       { ... }
     | REAL      { ... }
     | STRING    { ... }
     | ID        { ... }
     | type LIST { ... }
     | product_type {
        // TUPEL-TYP ERKANNT
        ...
     }
     ;
```

Abbildung 6.4: neue Produktion für das Nonterminal `type`

Der erweiterte Parser muss zwei Anforderungen erfüllen: Zum einen muss er Terme erkennen, die ein Ausdruck eines Produkttypen sind. Zum anderen muss er auch Annotationen (also die explizite Nennung eines Typs) erkennen, wie sie zum Beispiel bei der Deklaration von Variablen oder Rechenvorschriften verwendet werden.

Zunächst zum Erkennen einer Annotation; dazu wird, wie in Abb. 6.3 dargestellt, in `prg_parser.y` ein Nonterminal `product_type` als mit $*$ verbundene Konkatenation von bereits definierten Typen eingeführt.

Die Produktion für das Nonterminal `type` wurde erweitert (siehe Abb. 6.4), so dass neben den Basissorten `unit`, `bool`, `int`, `real`, `string` sowie Listen und nutzerdefinierten Typen auch das soeben neu eingeführte `product_type` einen Typ festlegen kann. Wenn ein `product_type` der Form $t_1 * ... * t_n$ an der Stelle einer Annotation auftritt, wird der Textbezeichner $t_1 * ... * t_n$ als Symbol in die Symboltabelle eingetragen und verweist dann auf die genaue Definition des Produkttypen.

Weiterhin muss der Parser Ausdrücke der Form $(e_1, ..., e_n)$ eines Produkttypen $t_1 * ... * t_n$ erkennen, wobei die e_i jeweils Ausdrücke des Typen t_i sind.

Für die Listentypen existiert bereits ein Nonterminal `expr_list`, das durch Komma getrennte Listen von Ausdrücken erkennt. Da es sich bei Ausdrücken eines Produkttypen auch um durch Komma getrennte Ausdrücke handelt, kann bei der Erkennung von Tupeln darauf zurückgegriffen werden.

In Abb. 6.5 wird die entsprechend erweiterte neue Produktion für das Nonterminal `expr`

```
expr : LET BRA_L letvars_listBRA_R EQUAL expr IN expr END {...}
     | infix_expr        { ... }
     | expr ANDALSO expr { ... }
     | expr ORELSE expr  { ... }
     | IF expr THEN expr ELSE expr %prec PREC_ELSE { ... }
     | BRA_L expr_list BRA_R {
         // TUPEL-AUSDRUCK ERKANNT
     }
     ;
```

Abbildung 6.5: neue Produktion für die Erkennung von Tupeln

dargestellt.

Eine Besonderheit ist, dass Produkttypen nicht explizit deklariert werden müssen. Beim Erkennen eines Tupels $(e_1, e_2, ..., e_n)$ vom Typ $t_1 * t_2 * ... * t_n$ wird implizit ein entsprechender Produkttyp angelegt – wenn er nicht bereits vorher eingeführt wurde.

6.2.1.3 Typechecker

Im Rahmen der Analyse nach dem Parsen eines Ausdrucks prüft das KIEL-System die Typen aller Knoten und inferiert die Typen aller Knoten, die noch keinen festen Typ haben.

Um sinnvoll mit den neuen Produkttypen umzugehen, mussten im Typechecker an einigen Stellen Erweiterungen vorgenommen werden.

6.2.1.4 Typzuweisung

Jedem Knoten im KIEL-System muss ein Typ zugewiesen sein. Diese Zuweisung nimmt auch bei Tupeln der Typechecker vor. Wenn dieser ein Tupel überprüft, bestimmt er zunächst aus den Typen der einzelnen Komponenten den Typ des gesamten Tupels als Zeichenreihe.

Dann sucht der Typechecker in der Symboltabelle, ob bereits ein passender Eintrag für den gefundenen Typ des Tupels existiert.

Wenn noch kein Eintrag für den neuen Produkttyp existiert, fügt der Typechecker den Typ und das dazugehörige Symbol in die entsprechenden internen Tabellen ein.

6.2.1.5 Kompatibilität von Typen prüfen

In der Routine `isFoundTypeCompatibleToExpected` prüft das KIEL-System, ob zwei Typen zueinander kompatibel sind. Diese Routine musste um die Unterstützung von

Produkttypen erweitert werden.

Im Fall der Basissorten überprüft die Routine bislang lediglich, ob die beiden Typen gleich sind. Komplizierter ist die Überprüfung bei Listentypen, da hier unter anderem die Sonderfälle von leeren Listen – die zu allen Listentypen kompatibel sind – berücksichtigt werden müssen.

Weil ein Tupel als Komponenten sowohl Ausdrücke der Basistypen als auch Ausdrücke eines Listentypen enthalten kann, muss die Erweiterung für Produkttypen auf die bisherigen Routine aufbauen und rekursiv vorgehen:

Falls die zu prüfenden Produkttypen $\texttt{e} = \texttt{e}_1 * ... * \texttt{e}_\texttt{n}$ und $\texttt{f} = \texttt{f}_1 * ... * \texttt{f}_\texttt{m}$ sind, wird zunächst geprüft, ob die Anzahl der Tupelelemente gleich ist. Dann erfolgt ein Test, ob alle Typenpaare jeweils zueinander kompatibel sind, also ob für alle $1 \leq \texttt{i} \leq \texttt{n}$ der Aufruf `isFoundTypeCompatibleToExpected`$(\texttt{e}_\texttt{i}, \texttt{f}_\texttt{i})$ den Wert `true` zurück liefert.

6.2.1.6 Kompatiblen Typen finden

Das KIEL-System benutzt an verschiedenen Stellen die Routine `getCompatibleType`, um einen Typ zu finden, der zu zwei vorgegebenen Typen kompatibel ist.

Die Routine wird zum Beispiel aufgerufen, um den Typen eines Terms wie `IF b THEN e ELSE f` zu bestimmen. Wenn `e` vom Typ $\texttt{t}_1$ und `f` vom Typ $\texttt{t}_2$ ist, liefert die Funktion einen Typen zurück, der zu beiden Typen $\texttt{t}_1$ und $\texttt{t}_2$ kompatibel ist.

Auch diese Routine muss um die Unterstützung von Produkttypen erweitert werden.

Für Listentypen und Basissorten gilt, dass die beiden Typen entweder nicht kompatibel sind oder entweder $\texttt{t}_1$ oder $\texttt{t}_2$ zu beiden Typen kompatibel ist: Die beiden Sorten `real` und `string` haben zum Beispiel keinen gemeinsamen Typ. Für die beiden Listentypen `list` und `int list` hingegen würde `int list` als gemeinsamer Typ bestimmt werden, weil dies der spezifischere der beiden Listentypen ist.

Um in der erweiterten Fassung der Routine für die beiden Produkttypen $\texttt{e} = \texttt{e}_1 * ... * \texttt{e}_\texttt{n}$ und $\texttt{f} = \texttt{f}_1 * ... * \texttt{f}_n$ den kompatiblen Typen $\texttt{k} = \texttt{k}_1 * ... * \texttt{k}_\texttt{n}$ zu finden, muss für alle Komponenten mit $1 \leq \texttt{i} \leq \texttt{n}$ paarweise der kompatible Typ $\texttt{k}_\texttt{i} = \texttt{getCompatibleType}(\texttt{e}_\texttt{i}, \texttt{f}_\texttt{i})$ gefunden werden.

Bei Produkttypen kommt es vor, dass $\texttt{k} \neq \texttt{e}$ und $\texttt{k} \neq \texttt{f}$ gilt, dass also nur ein komplett neuer Typ kompatibel zu beiden bestehenden Typen ist. Das passiert zum Beispiel bei den beiden Typen $\texttt{e} = (\texttt{int list}) * \texttt{list}$ und $\texttt{f} = \texttt{list} * (\texttt{int list})$. Diese beiden Typen sind kompatibel zueinander, aber der gemeinsame Produkttyp ist weder `e` noch `f`, sondern $\texttt{k} = (\texttt{int list}) * (\texttt{int list})$. In diesem Fall wird der Typ neu im System erzeugt und in der Symboltabelle eingetragen.

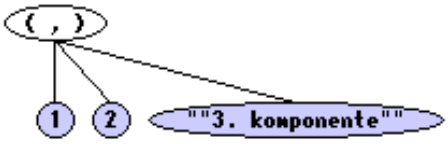


Abbildung 6.6: Darstellung eine Tupels

6.2.1.7 Evaluator

Damit ein Tupel-Knoten ausgewertet werden kann, mussten keine besonderen Änderungen erfolgen. Um einen Knoten auszuwerten, an den weitere Knoten angehängt sind, wertet der Evaluator zunächst diese darunterliegenden Knoten aus.

Da die Projektion (eines Tupels auf eine Komponente) als neue Operation hinzugefügt wurde, musste allerdings die Routine für die Auswertung der Basisoperation um die Berücksichtigung der entsprechenden Knotenart `e_project` erweitert werden.

6.2.1.8 Darstellung

Für die Darstellung von Tupeln wurde die in Abb. 6.6 gezeigte Anordnung von Knoten ausgewählt: Oben soll das Symbol für die Tupelbildung angezeigt werden, darunter nebeneinander die verschiedenen Komponenten des Tupels.

Dies entspricht der Darstellung von Listen. Die Anzeigeroutinen des KIEL-Systems sind so ausgelegt, dass verkettete Knoten der Knotenart `e_list` in der gewünschten Art platziert werden.

Weil Tupel intern ebenfalls mit Hilfe von `e_list` Knoten umgesetzt wurden, können sie ohne größere Änderungen wie gewünscht dargestellt werden. Lediglich das Symbol für den übergeordneten Knoten musste angepasst werden.

6.2.2 Lokale Wertedeklarationen

Um das KIEL-System um lokale Wertedeklarationen zu erweitern, mussten ebenfalls in allen fünf Hauptbereichen Syntaxtree, Parser, Typechecker, Evaluator und Darstellung Anpassungen vorgenommen werden.

Die erforderlichen Anpassungen werden nachfolgend beschrieben.

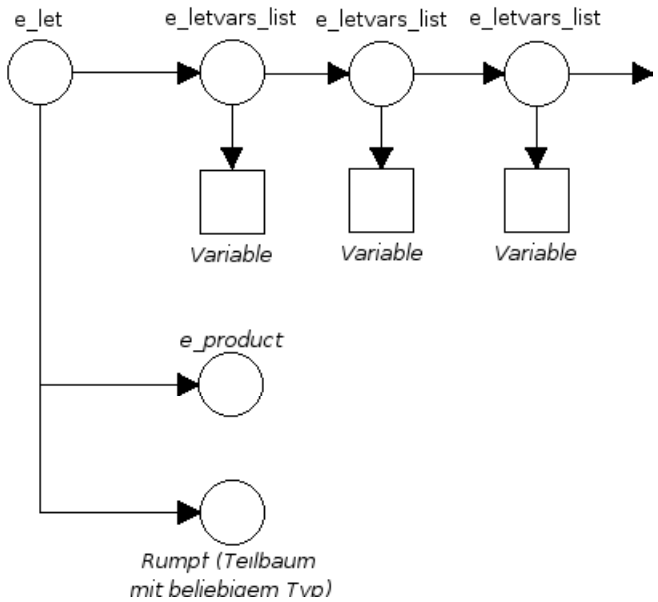


Abbildung 6.7: interne Struktur einer lokalen Wertedeklaration

6.2.2.1 Syntaxtree / Interne Darstellung

Intern wird eine lokale Wertedeklaration von einem speziellen Knoten der Knotenart `e_let` repräsentiert, an den drei weitere Knoten beziehungsweise Teilbäume angehängt sind. Diese drei Nachfolger sind:

- Liste der deklarierten Variablen: eine verkettete Liste von Knoten der Knotenart `e_letvar_list`, an die jeweils ein Knoten für die jeweilige Variable angehängt ist.
- zugewiesenen Werte: ein Teilbaum, dessen einzelne Komponenten bei der Auswertung jeweils den deklarierten Variablen zugewiesen werden sollen. Falls nur eine Variable deklariert wurde, kann dieser Teilbaum jeden beliebigen Typ haben, falls mehrere Variablen deklariert wurden, muss der Teilbaum ein Tupel sein, also als Knotenart `e_product` (siehe 6.2.1) haben.
- Rumpf: ein Teilbaum mit einem beliebigen Typ

Diese interne Struktur einer lokalen Wertedeklaration wird in Abb. 6.7 dargestellt.

6.2.2.2 Parser

Um diese interne Repräsentation zu erzeugen, musste der Parser um die Erkennung entsprechender Terme erweitert werden.

```
expr : LET BRA_L letvars_listBRA_R EQUAL expr IN expr END  {
           $$ = create_expr_let($3, $5, $7);
       }
     | infix_expr { ... }
     | expr ANDALSO expr { ... }
     | ...

letvars_listBRA_R : BRA_R { ... }
                  | ID COLON type BRA_R { ... }
                  | ID COLON type COMMA letvars_listBRA_R { ... }
                  ;
```

Abbildung 6.8: neue Produktion für Nonterminal `expr`

Eine lokale Wertedeklaration kann an jeder Stelle in einem Term verwendet werden, an der ein Ausdruck zugelassen ist. Daher wurde die in Abb. 6.8 dargestellte zusätzliche Produktion für das Nonterminal `expr` hinzugefügt. Die Liste der deklarierten Variablen wird dabei von einem neuen Nonterminal `letvars_listBRA_R` erkannt.

Wenn beim Parsen der Variablenliste ein neuer Bezeichner durch das Nonterminal `letvars_listBRA_R` erkannt wird, wird ein entsprechendes Symbol in der Symboltabelle angelegt, so dass ein Auftreten der neu deklarierten Variablen im Rumpf der lokalen Wertedeklaration keinen Fehler verursacht.

6.2.2.3 Typechecker

Im Rahmen der Analyse nach dem Parsen eines Ausdrucks prüft das KIEL-System die Typen aller Knoten und inferiert die Typen derjenigen Knoten, die noch keinen festen Typ haben. Dabei müssen auch die `e_let`-Knoten bearbeitet werden.

Wenn ein `e_let`-Knoten beim Prüfen in der Routine `checkBody` erreicht wird, dann geht das KIEL-System zunächst die deklarierten Variablen der Reihe nach durch und übernimmt die in der Wertedeklaration festgelegten Typen für alle entsprechenden Variablen-Knoten im Rumpf der Wertedeklaration.

Dies geschieht durch die in Abb. 6.9 dargestellte Schleife über die deklarierten Variablenbezeichner. Für jeden Bezeichner wird die Routine `setMLTypeAllOccurences` aufgerufen, wobei der jeweilige Variablenbezeichner, der zu setzende Typ und der Rumpf der lokalen Wertedeklaration als Argumente übergeben werden.

Die rekursive Routine `setMLTypeAllOccurences` setzt den Typ für alle Vorkommnisse eines Variablensymbols in einem Teilbaum. Dabei beachtet sie, dass in diesem Teilbaum eventuell lokale Wertedeklarationen auftreten können, in denen das Variablensym-

```
Expr_node_ptr vars = LET_VARS_EXPR_NODE(node);
while ( vars != NULL ) {
   setMLTypeAllOccurences(
        ARG_EXPR_NODE(vars)->strrep,
        ARG_EXPR_NODE(vars)->valuetype_expected,
        LET_EXPR_EXPR_NODE(node)
  );
  vars = NEXT_EXPR_NODE(vars);
}
```

Abbildung 6.9: Auszug aus `checkBody`: Iteration über Variablen einer lokalen Wertedeklaration

```
if ( inExpr->nodetype == e_let ) {
    // set type for all occurences in the expression-part
    // only if the variable is not defined in the let
    Expr_node_ptr vars = LET_VARS_EXPR_NODE(inExpr);
    int variable_is_defined_here = 0;
    while ( vars != NULL ) {
        if ( 0 == strcmp(ARG_EXPR_NODE(vars)->strrep,ofVar)  ) {
            variable_is_defined_here = 1;
            vars = NULL;
        } else {
           vars = NEXT_EXPR_NODE(vars);
        }
    }
    if ( ! variable_is_defined_here ) {
        setMLTypeAllOccurences(ofVar, mltype,
            LET_EXPR_EXPR_NODE(inExpr));
    }

   // always set type of all occurences in the value-part
   setMLTypeAllOccurences(ofVar, mltype,LET_VAL_EXPR_NODE(inExpr));

} else {

    // set type of current node if it is a matching variable-node,

    // recurse into all subtrees
    ...

} //if == e_let
```

Abbildung 6.10: Auszug aus der Routine `setMLTypeAllOccurences`

bol ebenfalls deklariert und damit verschattet wird. Ein Auszug aus dem Quelltext der Routine wird in Abb. 6.10 dargestellt.

Die Routine prüft am Anfang, ob der aktuelle Knoten eine neue lokale Wertedeklaration repräsentiert.

Falls es sich um eine neue lokale Wertedeklaration handelt, prüft die Routine zunächst, ob dem selben Variablenbezeichner auch hier ein Wert zugewiesen werden soll. Wenn dies der Fall ist, ruft sich die Routine rekursiv selbst auf, um die Variablenersetzung im rechten Teil dieser inneren Wertedeklaration durchzuführen. Ansonsten werden durch rekursive Aufrufe die Variablensymbole in allen drei Nachfolgebäumen ersetzt.

Falls es sich nicht um eine neue lokale Wertedeklaration handelt, wird zunächst geprüft, ob der aktuelle Knoten ein Variablen-Knoten ist, der das gerade bearbeitete Variablensymbol repräsentiert. Nur wenn dies der Fall ist, wird der Typ dieses Knotens entsprechend gesetzt. Andernfalls ruft sich die Routine selbst rekursiv auf, um die Typen in allen dem aktuellen Knoten nachfolgenden Teilbäumen zu setzen.

Nach Beendigung dieser Rekursion ist der korrekte Typ für alle Vorkommnisse des Variablensymbols im Gültigkeitsbereich der lokalen Wertedeklaration gesetzt.

Jetzt bestimmt das KIEL-System durch einen Aufruf der Routine `checkBody` die Typen des Rumpfes und des Werte-Teilbaumes (der rechten Seite der Wertedeklaration).

Danach wird geprüft, ob der soeben bestimmte Typ des Werte-Teilbaums zu den Typen der neu deklarierten Variablen $x_1 : t_1, \ldots, x_n : t_n$ passt. Dabei müssen zwei Fälle unterschieden werden:

Zunächst wird der Sonderfall abgehandelt, dass nur eine Variable deklariert wurde und der Werte-Teilbaum kein Tupel ist. Dies ist genau dann zulässig, wenn der Teilbaum den Typ der neu deklarierten Variable hat.

Falls der Werte-Teilbaum ein Tupel $(e_1, e_2, \ldots, e_m)$ ist, dann wird zunächst geprüft, ob die Anzahl der deklarierten Variablen n und die Anzahl der Komponenten des Tupels m gleich sind. Danach wird komponentenweise für alle $1 \leq i \leq n$ geprüft, ob der deklarierte Typ t_i der Variablen x_i und der Typ der entsprechenden Tupelkomponente e_i kompatibel sind.

Nur wenn dies für alle Variablen und Komponenten der Fall ist, wird als Ergebnis der für den Rumpf gefundene Typ als Typ des gesamten Ausdrucks zurückgegeben.

6.2.2.4 Evaluator

Bei der Auswertung von Ausdrücken musste in der Routine `evalThisNode` die Expansion von Knoten der Knotenart `e_let` hinzugefügt werden. Diese Knoten haben, wie

in 6.2.2.1 geschildert, jeweils drei Nachfolger: Variablenliste, rechte Seite der Deklaration (Wert) und Rumpf.

Im ersten Schritt der Expansion einer lokalen Wertedeklaration muss bestimmt werden, an welche Ausdrücke die deklarierten Variablen gebunden werden sollen.

Falls nur eine Variable deklariert wird, ist es einfach: Die Variable wird an die gesamte rechte Seite der Deklaration gebunden.

Falls mehrere Variablen deklariert werden, sind zwei Fälle zu unterscheiden: Entweder wurde die rechte Seite der Deklaration bereits ausgewertet und ist ein Tupel der Knotenart `e_product`; dies ist bei einer Leftmost-Innermost-Auswertung immer der Fall. Oder die rechte Seite wurde noch nicht ausgewertet und besteht zum Beispiel aus dem Aufruf einer Rechenvorschrift oder einer weiteren lokalen Wertedeklaration. Dieser Fall kann bei einer Auswertung nach der Leftmost-Outermost-Strategie auftreten.

Falls die rechte Seite der Deklaration bereits ausgewertet ist, wird jede neu deklarierte Variable einfach an die entsprechende Komponente des Wertetupels gebunden. Im anderen Fall wird es komplizierter: Für jede Variable muss zunächst ein Ausdruck erzeugt werden, in dem die passende Projektionsoperation auf die gesamte rechte Seite der Deklaration angewendet wird und so aus dem erst später auszuwertenden Ausdruck die richtige Komponente extrahiert.

Nachdem für jede Variable der passende Ausdruck zur Bindung bestimmt wurde, werden mit Hilfe der Funktion `replaceAllOccurencesAndSetExpansiondepth` alle Vorkommnisse dieser Variable im Rumpf der lokalen Wertedeklaration durch diesen Ausdruck ersetzt. Die Funktion berücksichtigt dabei – ähnlich wie die im letzten Abschnitt ausführlich vorgestellte Routine `setMLTypeAllOccurences` – mögliche Verschattungen von Variablen, falls im Rumpf lokale Wertedeklarationen vorkommen.

Danach wird die gesamte lokale Wertedeklaration gegen ihren wie oben geschildert veränderten Rumpf ausgetauscht.

6.2.2.5 Darstellung

Für die Darstellung einer lokalen Wertedeklaration wurde die in Abb. 6.11 auf der nächsten Seite gezeigte Anordnung von Knoten ausgewählt: Unter dem Knoten für die lokale Wertedeklaration soll zunächst die linke Seite der Deklaration mit den Variablensymbolen, dann der rechte Teil der Deklaration mit dem Wertetupel und schließlich der Rumpf angezeigt werden. Dabei sollen die Variablendeklarationen eckig statt oval gezeichnet werden, um sie vom Auftreten der Variablen im Rumpf unterscheiden zu können.

Eine entsprechende Positionierung der Knoten nehmen die bestehenden Routinen des KIEL-Systems bereits vor. Um die Deklaration der Variablen hervorzuheben, wurde in

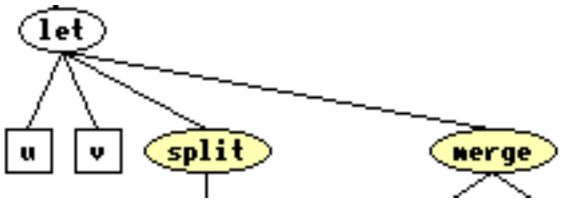


Abbildung 6.11: Darstellung einer lokalen Wertedeklaration

der Datei `drawable_expression.c`, in der die Zeichenroutinen für alle Baumelemente zusammengefasst sind, ein Sonderfall eingefügt, der diese Knoten mit eckigen Kästchen statt mit einem Oval darstellt.

6.3 Benutzeroberfläche

Um die Verwendung des KIEL-Systems zu erleichtern, wurden auch an der Benutzeroberfläche einige Erweiterungen vorgenommen.

In diesem Abschnitt werden die durchgeführten Änderungen für die Implementierung des Speicherns der Fensterkonfiguration und des Verarbeitens von Kommandozeilenparametern skizziert.

6.3.1 Speichern der Fensterkonfiguration

Die Position und die Größe aller Fenster des KIEL-Systems sollen beim Beenden der Anwendung gespeichert und beim nächsten Start wiederhergestellt werden.

Dazu wird beim Beenden des Programms die Routine `options_save()` aus der neu erstellten Datei `Misc/window_options.c` aufgerufen. Diese Routine speichert die Position und Größe der einzelnen Fenster in einer Konfigurationsdatei namens `.kielrc` im Homeverzeichnis des aktuellen Benutzers.

Dabei wird der Einfachheit halber kein Textformat verwendet, sondern ein binäres Format, bei dem die einzelnen Werte für Position und Größe an festen Positionen in der Datei liegen.

Beim Programmstart wird die Routine `options_load()` aufgerufen. Diese sucht im Homeverzeichnis des aktuellen Benutzers nach der Datei `.kielrc`. Falls die Datei gefunden wird, liest das KIEL-System sie ein und positioniert alle Fenster entsprechend.

6.3.2 Kommandozeilenparameter

In der neuen Version des KIEL-Systems sollen bereits beim Programmstart einige Befehle an das Programm gegeben werden können.

Als Teil des ohnehin bereits vom KIEL-System benutzten DBUG-Pakets werden auch Makros zur Verarbeitung von Kommandozeilenparametern geliefert.

Mit Hilfe dieser Makros konnte `kiel_main.c` so erweitert werden, dass die Parameter `--full-eval`, `--exp` und `--prg` erkannt werden. Die erkannten Vorgaben werden in globalen Variablen abgelegt.

Nach dem Start der GTK-Benutzeroberfläche werden in Abhängigkeit von den globalen Variablen entsprechende Aktionen gestartet.

6.4 Aufrufgraph

Eine weitere neue Funktion ist das Zeichnen eines Aufrufgraphen. In diesem werden alle in einem Programm deklarierten Rechenvorschriften dargestellt. Aufrufe einer Rechenvorschrift aus dem Rumpf einer Rechenvorschrift heraus werden durch Pfeile eingezeichnet. Damit lassen sich Zusammenhänge in einem Programm leicht erkennen.

Das KIEL-System verwendet zur Erstellung dieses Graphen die Bibliothek Graphviz [13]. Diese stellt Funktionen zum Zeichnen gerichteter und ungerichteter Graphen zur Verfügung. Mit dem in der Distribution von Graphviz enthaltenen Programm `dot` kann aus einer textuellen Beschreibung des Graphen eine Postscript-Datei mit einer Abbildung erzeugt werden.

Um einen Aufrufgraphen zu erzeugen, schreibt die Routine `show_callgraph()` in der neuen Datei `Misc/callgraph.c` für alle Knoten und Kanten entsprechende Befehle in eine Textdatei.

Dann wird `dot` aufgerufen und die erstellte Grafik mit Hilfe des Programms `gv` angezeigt.

6.5 Verwendete Werkzeuge

Bei der Weiterentwicklung des KIEL-Systems wurden verschiedene Hilfsmittel eingesetzt, um die Arbeit zu erleichtern. Sie sollen im Zusammenhang mit Ihrem Einsatz im folgenden Abschnitt kurz vorgestellt werden.

6.5.1 Version-Control-System

Ein Version-Control-System (VCS) ist ein Werkzeug, um verschiedene Versionen einer Informationseinheit zu verwalten. Bei der Programmentwicklung kann ein VCS helfen, die Übersicht über Änderungen an den Dateien des Quelltextes zu behalten.

Im Rahmen dieser Arbeit wurde das Version-Control-System Subversion (SVN) [14] eingesetzt, das als Weiterentwicklung aus dem bekannten Concurrent-Versioning-System (CVS) hervorgegangen ist. SVN speichert einen Baum von Verzeichnissen und Dateien auf einem Server in einem sogenannten Repository. Solch ein Repository wurde auch für die Weiterentwicklung des KIEL-Systems angelegt. Danach konnte der Quelltext des bestehenden KIEL-Systems als Revision 1 importiert werden.

Später wurden nach jeden Schritt in der Entwicklung die Änderungen in das Repository überspielt, wobei die erfolgten Änderungen vom Entwickler in einem Kommentar beschrieben wurden. Nach dem Überspielen wurde von SVN für den neuen Stand eine neue Versionsnummer erzeugt.

Mit Hilfe einfacher Befehle kann man im Nachhinein nachvollziehen, wann und warum eine Änderung erfolgt ist. Dies war zum Beispiel sehr hilfreich, um einen während der Entwicklung eingebauten Fehler zu diagnostizieren und zu beheben: Mit Hilfe von SVN war leicht nachvollziehbar, welche Programmzeilen zwischen der letzten fehlerfreien und der ersten fehlerbehafteten Programmversion geändert worden waren.

Vor allem aber ermöglichte SVN die parallele Entwicklung auf mehreren Systemen. So konnte sowohl auf den Sun Systemen im Grundausbildungspool an der Universität als auch auf mehreren privaten PCs am gleichen Quelltext gearbeitet werden. Ohne SVN wäre es nötig gewesen, ständig Quelltexte zwischen diesen Systemen hin und her zu kopieren. Dabei hätte die Gefahr bestanden, beim Wechsel von System zu System Änderungen zu übersehen, zu überschreiben oder zu vergessen.

6.5.2 Entwicklungsumgebung KDevelop 3

Bei der Weiterentwicklung des KIEL-Systems wurde auf die integrierte Entwicklungsumgebung *KDevelop3* zurückgegriffen. Diese stellt unter einer Oberfläche eine leistungsfähige Quelltextverwaltung, einen komfortablen Editor sowie Schnittstellen zum C-Übersetzer und einen Debugger zur Verfügung.

Besonders wichtig für die Entwicklung der Erweiterungen und die Behebung der bestehenden Fehler war die komfortable Anbindung an den Debugger *gdb.* Mit dessen Hilfe lässt sich ein entsprechend übersetztes Programm ausführen und während der Ausführung an einer beliebiger Stelle anhalten. Nun können die Werte aller Variablen inspiziert

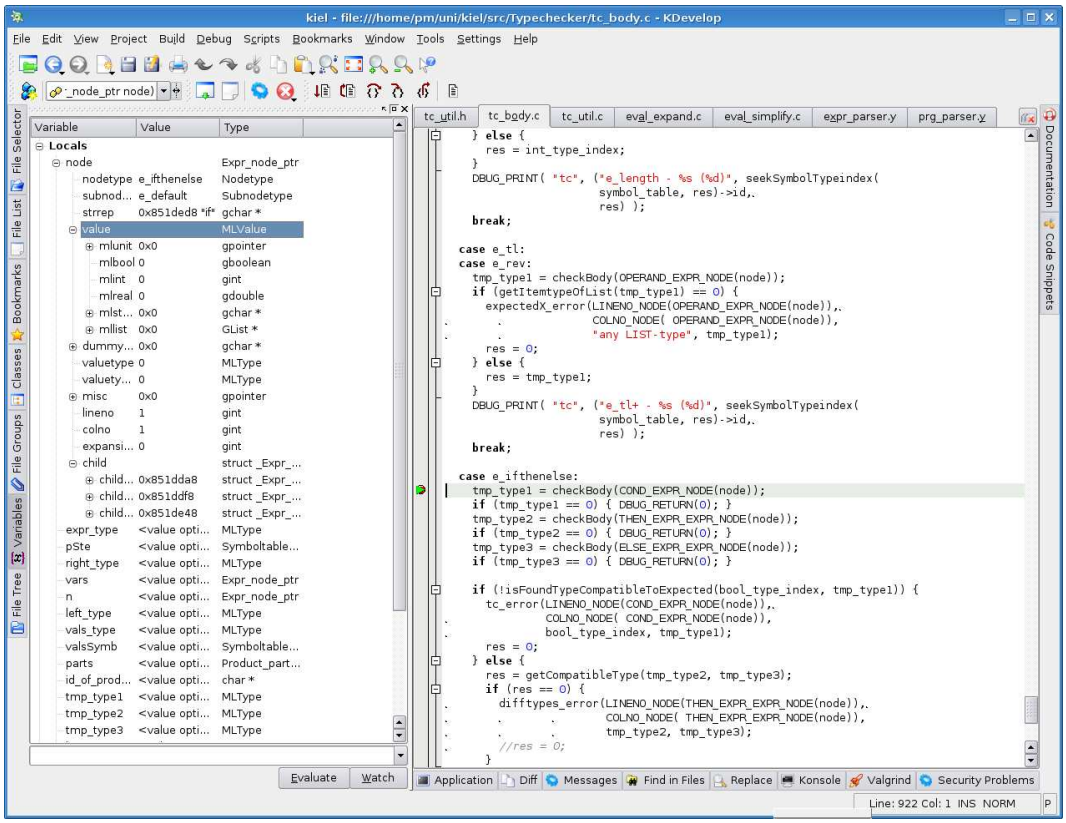


Abbildung 6.12: Entwicklungsumgebung KDevelop3 während der Fehlersuche

werden (siehe Abb. 6.12) und das Programm kann schrittweise weiter ausgeführt werden.

So ist es dem der Entwickler möglich, genau nachvollziehen, welche Befehle im Quelltext des Programms für eine ungewollte Änderung der Daten im Hauptspeicher verantwortlich sind.

Die alte Version des KIEL-Systems hatte ein Problem bei der Auswertung von Termen mit Hilfe der vollständigen Substitution: Nicht alle Aufrufe von Rechenvorschriften wurden expandiert. Um diesen Fehler zu beheben, wurde zunächst ein möglichst einfacher Ausdruck ausgewählt, bei dessen Auswertung das Problem beobachtet werden konnte.

Dann wurde das KIEL-System unter Aufsicht des Debuggers ausgeführt. Vor der Auswertung des Ausdrucks wurde das Programm gestoppt, so dass man nun die Programmschritte bei der Auswertung des Ausdrucks einzeln beobachten konnte. Schnell war klar, dass die Rekursion, die alle Aufrufe von Rechenvorschriften finden und expandieren sollte, durch eine falsche Bedingung vorzeitig abbrach. Mit der Korrektur dieser Abbruchbedingung war der Fehler behoben.

7 Zusammenfassung und Ausblick

Am Ende dieser Arbeit soll zunächst überprüft werden, ob die in Kapitel 3 geschilderten Ziele vollständig erreicht werden konnten.

Abschließend werden mögliche zusätzliche Erweiterungen des KIEL-Systems im Rahmen weiterer Diplomarbeiten avisiert.

7.1 Zusammenfassung

Die Hauptaufgabe im Rahmen dieser Diplomarbeit war die Erweiterung des vom KIEL-System unterstützten Sprachumfangs.

Bislang konnten Nutzer in Ausdrücken und Programmen lediglich die in den Kapiteln 2.4 und 2.5 geschilderten Elemente benutzen.

Die Erweiterung sollte alle in Kapitel 2.6 geschilderten Sprachelemente umfassen. Diese Erweiterung wurde erfolgreich durchgeführt, wie an den entsprechenden Beispielen im Kapitel 5 zu erkennen ist. Mit Hilfe der neu eingeführten Produkttypen, des neuen Projektionsoperators sowie den lokalen Wertedeklarationen lassen sich effizientere und übersichtlichere Programme erstellen.

Daneben waren bei der Benutzung des alten KIEL-Systems einige Fehler bei der Auswertung von Termen aufgefallen. Diese Fehler konnten, wie gewünscht, behoben werden. Die Erweiterung um das automatische Speichern und Wiederherstellen von Fensterposition und -größe hat die Benutzerfreundlichkeit des KIEL-Systems, wie geplant, verbessert. Darüber hinaus bietet das neue KIEL-System eine Funktion für die Darstellung eines Aufrufgraphen. Mit diesem Hilfsmittel kann ein Nutzer auch bei komplizierten Programmen leichter die Übersicht bewahren.

Mittels der in dieser Arbeit dargestellten Weiterentwicklung sollte das KIEL-System zukünftig noch besser als bisher in der Lehre eingesetzt werden können.

7.2 Ausblick

Neben den im Rahmen dieser Diplomarbeit durchgeführten Erweiterungen eröffnen sich weitere Möglichkeiten für die Verbesserung des KIEL-Systems. Einige davon sollen nach-

```
fun filter(f : int -> bool, []: int list ) : int list =
    []
|   filter(f : int -> bool, x::xs : int list ) : int list =
    if f(x) then x::filter(f,xs)
            else filter(f,xs);
```

Abbildung 7.1: Standard ML Rechenvorschrift `filter`

folgend beschrieben werden. Dabei geht es zunächst um die Erweiterung des unterstützten Sprachumfangs und dann um einen Vorschlag zur verbesserten Visualisierung der Auswertung.

7.2.1 Sprachumfang

Als letzte große Lücke beim unterstützten Sprachumfang von Standard ML verbleiben die Funktionstypen. Mit Hilfe des Typkonstruktors -> lassen sich in Standard ML neue Funktionstypen deklarieren. Eine Instanz dieses Typs ist eine Funktion, die zum Beispiel einer Variablen zugewiesen oder im Aufruf einer Rechenvorschrift als Argument übergeben werden kann.

Ein Beispiel für die Nutzung von Funktionstypen ist die universell verwendbare Rechenvorschrift `filter` für Listen von ganzen Zahlen. Diese Rechenvorschrift gibt zu einer Liste `L` von ganzen Zahlen und einer Funktion `f`, die eine ganze Zahl auf einen Wahrheitswert abbildet, genau die Liste zurück, die alle Elemente aus `L` in originärer Reihenfolge enthält, für die `f` wahr ist. Eine Standard ML Rechenvorschrift, die `filter` implementiert, ist in Abb. 7.1 angegeben.

Mit Hilfe von `filter` und den in 2.4.2 definierten Rechenvorschriften `ist_gerade` und `ist_ungerade` kann man zum Beispiel umstandslos alle geraden Zahlen aus einer Liste herausfiltern:

$$\textbf{filter}(\texttt{ist\_gerade}, [2, 5, 9, 1, 4, 88]) \Rightarrow [2, 4, 88]$$

Auch ist es möglich, Hilfsrechenvorschriften in lokalen Wertedeklarationen zu verstecken. Damit können Konflikte bei der Namensvergabe dieser Hilfsroutinen vermieden werden. In 5.1.3 wird eine effiziente Routine zur Berechnung einer beliebigen Fibonacci-Zahl angegebenen. Leider ist es dabei nötig, neben der Rechenvorschrift `fib` auch eine Hilfsrechenvorschrift `G` zu deklarieren.

Diese Rechenvorschrift `G` ist nicht zum allgemeinen Gebrauch bestimmt, sondern soll nur von `fib` aufgerufen werden. Daher liegt es nahe, selbige innerhalb von `fib` im Rahmen einer lokalen Wertedeklaration einzuführen, wie es in Abb. 7.2 dargestellt wird.

Somit kann die Hilfsrechenvorschrift `G` wirklich nur von `fib` genutzt werden, und es

```
fun fib(n : int) : int =
   let val rec G : int -> int * int =
       fn(x:int) =>
              if x = 0 then (1,1)
                       else let val (x : int, y : int) = G(x-1)
                            in  (y,x+y)
                            end
   in let val (x : int, y : int) = G(n)
      in  x
      end
  end;
```

Abbildung 7.2: versteckte Hilfsrechenvorschrift in Standard ML

gibt weniger Möglichkeiten für Namenskonflikte – schließlich könnte auch eine andere Rechenvorschrift eine auf den nichtssagenden Namen `G` getaufte Hilfsrechenvorschrift verwenden.

7.2.2 Visualisierung der Auswertung

Bei der schrittweisen Auswertung von Ausdrücken wird der Ausdrucksbaum nach jedem Schritt neu gezeichnet. Dabei werden aufgrund des verwendeten Algorithmus sämtliche Knoten jedes Mal neu positioniert. Bei größeren Änderungen im Ausdruck – wenn zum Beispiel eine Rechenvorschrift expandiert oder eine Verzweigung vereinfacht wird – kann sich die Darstellung des Baumes deutlich verändern, und es ist schwer zu erkennen, welcher Knoten nun im letzten Schritt expandiert beziehungsweise vereinfacht wurde.

Hier könnte folgendes Verfahren Abhilfe schaffen: Nach dem Klick des Benutzers wird vor der Ausführung des Auswertungsschritts der zu vereinfachende oder zu expandierende Knoten durch einen zusätzlichen grauen Rand markiert. Dann wird der Auswertungsschritt durchgeführt und danach der Baum neu gezeichnet, wobei der ersetzende Knoten ebenfalls durch einen grauen Rand markiert ist. Im Falle der Expansion von lokalen Wertedeklarationen und Rechenvorschriften könnten die durch Ersetzung von Variablensymbolen entstandenen Knoten ebenfalls markiert sein.

Ein Beispiel für diesen Effekt ist in Abb. 7.3 auf der nächsten Seite dargestellt: Im ersten Fenster wird ein Ausdruck abgebildet, wie er bei der Auswertung der Morris-Funktion auftritt. Nun gibt der Nutzer den Befehl für die Auswertung des nächsten Schrittes: Der innere Aufruf von F soll expandiert werden. Sofort wird dieser innere Aufruf durch einen grauen Rahmen markiert, wie im mittleren Fenster zu sehen ist.

Nach der Expansion wird der neue Ausdruck wie im rechten Fenster abgebildet als Baum

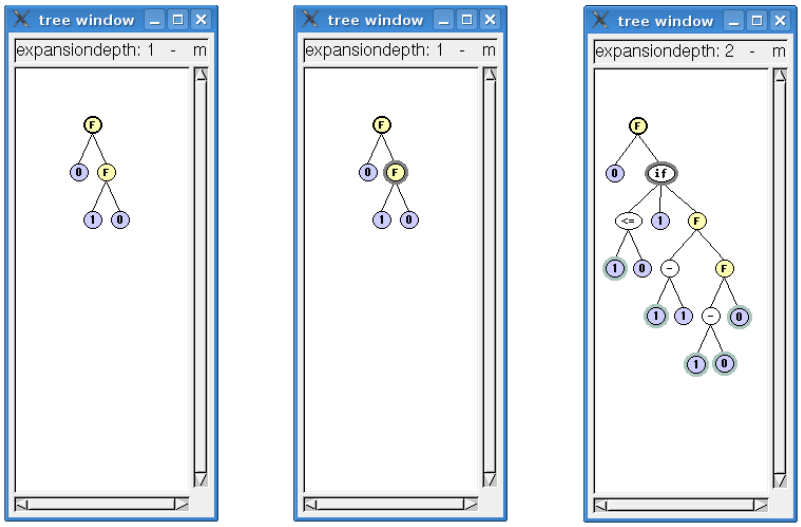


Abbildung 7.3: mögliche Visualisierung eines Auswertungsschritts

angezeigt. Dabei ist die Stelle, an der der Rumpf der Rechenvorschrift eingefügt wurde, durch denselben grauen Rahmen markiert. Im Rumpf selbst sind die Knoten, die durch die Ersetzung von Variablensymbolen entstanden sind, mit einem grünlichen Rahmen markiert.

Mit dieser Erweiterung wäre es für den Nutzer leichter, die Auswertung komplexer Ausdrücke zu verfolgen und das Verständnis komplexer Auswertungen fiele noch leichter.

Literaturverzeichnis

[1] M. Tiedt: *KIEL: Ein Werkzeug zur Visualisierung von Termersetzungssemantik,* Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, 1999

[2] H. Ernst: *Benutzergesteuerte Termersetzung - Erweiterung von KIEL um datatype-Deklarationen und die Deklaration von Rechenvorschriften mit Musterkennung,* Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, 2001

[3] O. Lorenz: *Vom C-Programm zum Applet am Beispiel von KIEL*, Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, 2000

[4] R. Berghammer, *Informatik I,* Skript zur gleichnamigen Vorlesung an der Christian-Albrechts-Universität zu Kiel, 2005

[5] R. Berghammer, *On a Teaching Tool for Functional Programming and Visual Program Executiom*, unveröffentliches Manuskript, 2006

[6] J. D. Ullman, *Elements of ML Programming*, 2nd ed., Prentice Hall, 1997

[7] http://www.cs.bc.edu/~gtan/historyOfFP/historyOfFP.html, 07.06.2006

[8] http://www.faqs.org/faqs/meta-lang-faq/, 07.06.2006

[9] http://en.wikipedia.org/wiki/ML_programming_language, 07.06.2006

[10] M. Becker, *Fibonacci-Zahlen,* http://www.ijon.de/mathe/fibonacci/index.html, 10.07.2006

[11] J. R. Levine, T. Mason, D. Brown: *lex & yacc*, 2nd ed., O'Reilly, 1995

[12] D. Dougherty, A. Robbins: *sed & awk,* 2nd ed., O'Reilly, 1997

[13] E. Gansner, S. North: *An open graph visualization system and its applications to software engineering*, Software - Practice and Experience, vol. 30, no. 11, p. 1203–1233, 2000

[14] C. M. Pilato, B. Collins-Sussman, B. W. Fitzpatrick, *Version Control with Subversion*, O'Reilly, 2004

[15] J. H. Morris, *Lambda-calculus models of programming*, PhD. thesis, Mass. Inst. of Technology, 1968

Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst habe und ausschließlich die angegebenen Hilfsmittel und Quellen verwendet habe.

Kiel, den 25.07.2006

_____

Paul Mallach