# ULTRA

# Ein interaktives Transformationssystem für funktionale Programme

Diplomarbeit an der Universität Ulm

Fakultät für Informatik

vorgelegt von:

Mark Dettinger

1. Gutachter: Prof. Dr. Helmuth Partsch
2. Gutachter: Prof. Dr. Uwe Schöning

1997

# Zusammenfassung

Die Transformationelle Programmentwicklung ist ein Verfahren, ein Programm aus einer gegebenen Problemspezifikation durch sukzessive Anwendung semantikerhaltender Transformationen herzuleiten. Eine solche Herleitung unterteilt sich üblicherweise in drei Phasen.

In der ersten Phase werden deskriptive Konstrukte in der Spezifikation eliminiert, so daß man ein ausführbares funktionales Programm erhält. In der zweiten wird mit weiteren Transformationen versucht, das Programm in ein effizienteres - aber immer noch funktionales - Programm umzuwandeln. Schließlich wird in der dritten Phase das funktionale Programm in ein imperatives Programm transformiert, was in der Regel eine weitere Effizienzsteigerung bringt.

Diese Arbeit präsentiert das Transformationssystem ULTRA (Ulm's Transformation System), welches die zweite Phase der transformationellen Programmentwicklung unterstützt.

Mit ULTRA können funktionale Programme der Programmiersprache Gofer mittels des von Burstall und Darlington entwickelten Unfold/Fold-Verfahrens [BD77] in Interaktion mit dem System in äquivalente Programme transformiert werden.

ULTRA ist selbst in Gofer geschrieben und zeichnet sich durch leichte Bedienbarkeit und Erweiterbarkeit aus. Der Benutzer kann seine eigenen Transformationsregeln definieren und diese auf Typkorrektheit überprüfen lassen. Außerdem steht ihm ein leistungsfähiger *Simplifier* zur Verfügung, der die Programmtransformation teilweise automatisiert. Schließlich hat der Benutzer noch die Möglichkeit, Strategien zu definieren. Eine Strategie ist eine algorithmische Beschreibung, nach der die Auswahl von Transformationsregeln und Anwendungsstellen gesteuert wird.

# Inhaltsverzeichnis

# Kapitel 1 Einleitung

Die Entwicklung korrekter und effizienter Software ist ein Problem, das seit Jahrzehnten Forschungsgegenstand der Informatik ist.

Für die meisten Anwendungen (Textverarbeitungssysteme, Internet-Browser, Computerspiele) reichen die üblichen informellen Methoden des Software Engineerings (Black-Box-Tests, Code Reviews, usw.) aus, um die Korrektheit eines Programms „nachzuweisen“.

Bei manchen Programmen ist Korrektheit jedoch eine lebenswichtige Forderung. Man denke nur an den Autopiloten eines Flugzeugs, die Steuerung eines bemannten Raumfahrzeugs oder an das Programm im Auto, das über das Auslösen des Airbags entscheidet.

Bei der Entwicklung von solchen sicherheitskritischen Anwendungen sind deshalb formale Methoden zum Nachweis der Korrektheit erforderlich. Eine dieser Methoden ist die *transformationelle Programmentwicklung.*

Die transformationelle Programmentwicklung ist ein Verfahren, aus einer gegebenen formalen Spezifikation durch sukzessive Anwendung semantikerhaltender Transformationen ein effizientes Programm herzuleiten. Üblicherweise unterteilt man eine solche Herleitung in drei Phasen.

In der ersten Phase werden deskriptive Konstrukte in der Spezifikation eliminiert, so daß man ein ausführbares funktionales Programm erhält. In der zweiten wird mit weiteren Transformationen versucht, das Programm in ein effizienteres, aber immer noch funktionales, Programm umzuwandeln - z. B. durch Vereinfachung der Rekursionsstruktur. Schließlich wird in der dritten Phase das funktionale Programm in ein imperatives Programm transformiert, was in der Regel eine weitere Effizienzsteigerung bringt. Da die Transformation funktionaler Programme aufgrund ihrer referentiellen Transparenz bedeutend einfacher handzuhaben ist als die Transformation imperativer Programme, vollzieht man den Übergang zum imperativen Programm erst am Ende der Entwicklung (Abb. 1.1).

Ein Programm, das auf diese Art entwickelt wurde, ist korrekt per Konstruktion, da jeder einzelne Transformationsschritt semantiktreu ist. Man kann sich am Ende der Entwicklung sicher sein, daß das Programm die Spezifikation erfüllt. Die protokollierte Entwicklungsgeschichte des Programms dient als Korrektheitsbeweis, und außerdem auch als hervorragende Programmdokumentation.

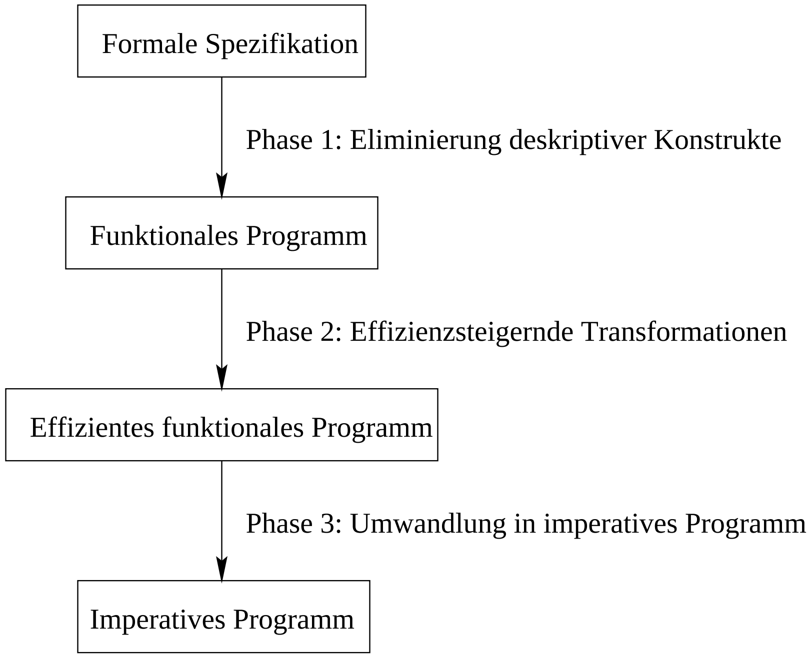


Abbildung 1.1: *Die drei Phasen der transformationellen Programmentwicklung*

## 1.1 Problemstellung

Das Ziel dieser Diplomarbeit war die Entwicklung des Transformationssystems ULTRA (Ulm's Transformation System), einem Werkzeug, das den Programmierer bei der zweiten Phase der transformationellen Programmentwicklung, der Effizienzsteigerung eines funktionalen Programms, unterstützt.

Das System sollte intuitiv über eine grafische Benutzeroberfläche bedient werden können. Dazu sollte es möglich sein, den zu transformierenden Programmteil zu markieren, die gewünschte Transformation auszuwählen und ausführen zu lassen. Durch leichte Bedienbarkeit des Systems sollte insbesondere ein späterer Einsatz von ULTRA in der Lehre ermöglicht werden. Zur Realisierung dieser Anforderung kam bei der Implementierung die an der Universität Ulm entwickelte Sprache *TkGofer* zum Einsatz, die eine Erweiterung der funktionalen Sprache *Gofer* darstellt und speziell für die Entwicklung grafischer Oberflächen geschaffen wurde.

Als Quell- und Zielsprache der von ULTRA durchgeführten Transformationen wurde ebenfalls Gofer gewählt. Da Programmtransformationen nicht auf der konkreten, sondern auf der abstrakten Syntax ausgeführt werden sollten, mußte ein Parser sowie ein Pretty-Printer für diese Sprache entwickelt werden.

Der Benutzer sollte weiterhin die Möglichkeit haben, eigene Transformationsregeln zu definieren. Hierbei stellte sich die Frage nach einer geeigneten Syntax, um solche Regeln aufzuschreiben. Da die Korrektheit der mit ULTRA durchgeführten Programmentwicklungen außerdem von der Korrektheit der benutzerdefinierten Regeln abhängig ist, stellte sich weiterhin die Frage, inwieweit das System dem Benutzer Hilfe anbieten kann, dessen selbst definierte Transformationsregeln auf Korrektheit zu überprüfen. Die Syntax der Regeln wurde deswegen so gewählt, daß sie der Gofer-Grammatik gehorcht. Dies ermöglicht es, die Regeln mit dem Gofer-Interpreter selbst auf syntaktische Korrektheit und Typkorrektheit zu testen.

Um mit angemessenem Aufwand praktische Resultate erzielen zu können, sollte die Programmtransformation außerdem teilweise automatisiert sein. Dazu wurde erstens der *Simplifier* entwickelt, der nach einer fest eingebauten Strategie Ausdrücke zu vereinfachen versucht. Das Problem hierbei war das Herausfinden eines geeigneten Algorithmus, der für eine möglichst große Klasse von Ausdrücken Vereinfachungsmöglichkeiten erkennt. Zweitens wurde die Möglichkeit geschaffen, Transformationsschritte auf verschiedene Arten zu Strategien zu kombinieren. Dazu stehen dem Benutzer verschiedene Kombinatoren zur Verfügung, z. B. Iteration, Rekursion oder sequentielle Komposition von Transformationsschritten. Das Problem war hier, eine Sprache zur Formulierung von Strategien zu entwickeln, die ausdrucksstark genug ist, um einigermaßen interessante Strategien beschreiben zu können.

## 1.2 Überblick über die folgenden Kapitel

In Kapitel 2 wird eine kurze Einführung in die funktionale Programmierung sowie in die transformationelle Programmentwicklung gegeben. In Kapitel 3 wird ULTRA vorgestellt und die Bedienung des Systems anhand einiger Beispiele erklärt. Wie der Benutzer das System durch Hinzufügen von neuen Transformationsregeln, Strategien oder semantischem Wissen über die zu transformierende Sprache erweitern kann, folgt in Kapitel 4. Kapitel 5 geht anschließend ausführlich auf die Implementierung ein. Dabei werden die bei der Implementierung aufgetretenen verschiedenen Probleme erwähnt und deren Lösungen beschrieben. Zum Schluß werden in Kapitel 6 die Ergebnisse dieser Diplomarbeit zusammengefaßt. Im Anhang werden schließlich eine Reihe von Fallbeispielen, d. h. Programmtransformationen, die mit ULTRA durchgeführt wurden, vorgestellt.

## 1.3 Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung dieser Arbeit direkt oder indirekt unterstützt haben. Für die Ermöglichung dieser Diplomarbeit und sein Gutachten danke ich dem Leiter der Abteilung Programmiermethodik und Compilerbau, Prof. Dr. Helmuth Partsch. Für sein Interesse an dieser Arbeit und das Zweitgutachten danke ich Prof. Dr. Uwe Schöning, dem Leiter der Abteilung Theoretische Informatik.

Mein besonderer Dank gilt meinen beiden Betreuern, Drs. Ton Vullinghs und Dr. Wolfram Schulte, die mich in wirklich vielen Stunden mit Anregungen, Verbesserungsvorschlägen, konstruktiver Kritik und Literaturempfehlungen weitergebracht haben.

Desweiteren möchte ich mich bei Dipl. Inf. Thilo Schwinn für die Einführung in die Erstellung grafischer Benutzeroberflächen mit TkGofer sowie bei Dipl. Inf. Stefan Sarstedt und bei Jochen Rütschlin für gelegentliche Tips im Umgang mit dem Textsatzprogramm LaTeX bedanken.

# Kapitel 2 Grundlagen

Dieses Kapitel soll einige im weiteren Verlauf dieser Arbeit benötigten Kenntnisse vermitteln. In Abschnitt 2.1 werden zunächst die Konzepte der funktionalen Programmierung vorgestellt. Abschnitt 2.2 befaßt sich näher mit den Grundlagen der transformationellen Programmentwicklung. In Abschnitt 2.3 wird schließlich auf die maschinelle Unterstützung durch Transformationssysteme eingegangen.

## 2.1 Funktionale Programmierung

Die *funktionale Programmierung* basiert auf der Definition von Funktionen im mathematischen Sinne. Die Auswertung eines Programms erfolgt durch die schrittweise Reduktion von Ausdrücken. Den Wert einer Variablen durch eine Zuweisung zu verändern, ist in funktionalen Sprachen nicht möglich, da Variablen - im Gegensatz zur imperativen Programmierung - keine Speicherplätze, sondern Ausdrücke bezeichnen. Ausdrücke besitzen aber, unabhängig von der Reihenfolge und dem Zeitpunkt ihrer Auswertung, immer ein und denselben Wert.
Diese Eigenschaft, die sogenannte *referentielle Transparenz*, ermöglicht es, funktionale Programme algebraisch zu manipulieren. Der Ausdruck $f(x) + g(x)$ ist beispielsweise garantiert äquivalent zu $g(x) + f(x)$. In einer imperativen Sprache wäre hier Vorsicht angebracht, da die Variable $x$ ja von der Funktion $f$ verändert werden könnte. In funktionalen Sprachen können jedoch keine Seiteneffekte (Veränderung der Inhalte von Speicherzellen durch Zuweisungen) auftreten. Insbesondere wird dadurch auch eine parallele Berechnung von Teilausdrücken möglich.

Weitere Eigenschaften moderner funktionaler Sprachen sind die verzögerte Auswertung (*Laziness*) und die mögliche *Über-* und *Unterversorgung* von Funktionen. Verzögerte Auswertung bedeutet, daß Ausdrücke nur dann ausgewertet werden, wenn sie auch wirklich benötigt werden. Unterversorgung bedeutet, eine $n$-stellige Funktion mit $k < n$ Argumenten aufzurufen. Das Ergebnis eines solchen Aufrufs ist eine spezialisierte Funktion, die noch $n - k$ Argumente erwartet. Wenn eine Funktion als Ergebnis wiederum eine Funktion liefert, darf sie auch überversorgt, d. h. mit zuvielen Argumenten aufgerufen werden.

In diesem Fall wird die Funktion zuerst mit der richtigen Anzahl von Argumenten aufgerufen. Die zurückgelieferte Ergebnisfunktion wird dann anschließend auf die überzähligen Argumente angewandt.

Ein interaktives Programm zu schreiben, das mit dem Benutzer in Form von Ein- und Ausgaben kommuniziert, stellt in einer funktionalen Sprache allerdings ein Problem dar. Theoretisch ist solche Interaktion gar nicht realisierbar, da durch sie das Prinzip der referentiellen Transparenz verletzt wird. Die beliebige Reihenfolge, in der Ausdrücke ausgewertet werden, steht außerdem der manchmal gewünschten sequentiellen Ausführung von Aktionen im Wege.

Um dieses Problem zu lösen, wurde die funktionale Programmierung um verschiedene Konzepte erweitert, wie z. B. die Programmierung mit *Continuations* oder *Monaden*. Bei der Implementierung von ULTRA wurde auf eben diese Konzepte zurückgegriffen. Für eine Einführung in diese Thematik verweise ich den Leser auf [Wad95] und [PJW93].

Eine umfassende Einführung in die funktionale Programmierung findet man in [BW89] und [SV96]. Eine eingehende Beschreibung der in dieser Arbeit verwendeten Sprache *Gofer* ist in [Jon93a] und [Jon93b] enthalten.

## 2.2 Transformationelle Programmentwicklung

Dieses Kapitel soll in die theoretischen Grundlagen der transformationellen Programmentwicklung einführen. Für eine ausführlichere Beschreibung des Prinzips der transformationellen Programmentwicklung verweise ich den interessierten Leser auf [Pep84], [BEH+87] und [Par90].

Als erstes soll im folgenden das Unfold/Fold-System vorgestellt werden, welches als grundlegendes System für Programmtransformationen angesehen werden kann.

### 2.2.1 Das Unfold/Fold-System

Das Unfold/Fold-System wurde 1977 von Burstall und Darlington [BD77] entwickelt. Es beschreibt im Prinzip sechs verschiedene Arten von Transformationen, die auf ein Programm angewandt werden können.

1. Define

   Unter einem *Define*-Schritt verstehen wir die Einführung einer neuen Funktionsdefinition. Während einer Programmtransformation können damit neue Ideen eingebracht werden.

   Zu beachten ist bei einem *Define*-Schritt, daß der dabei neu eingeführte Funktionsname nicht schon existiert.

2. Unfold

   Ein *Unfold*-Schritt besteht in dem Ersetzen eines Funktionsaufrufs durch den Rumpf der Funktionsdefinition, wobei Argumente entsprechend substituiert werden. Angenommen, es existiert die Funktionsdefinition

$$f\ p_1\ ...\ p_n = body$$

mit den Patterns $p_1$ bis $p_n$ und es liegt ein Aufruf $f\ e_1\ ...\ e_n$ vor, wobei die Ausdrücke $e_1, ..., e_n$ auf die Patterns $p_1, ..., p_n$ passen. Dann kann dieser Aufruf entfaltet werden. Damit wird er durch *body* ersetzt, wobei in *body* alle Vorkommen von Variablen der $p_i$ durch die entsprechenden Teilausdrücke der $e_i$ ersetzt werden.

*Beispiel 1:* Angenommen, unser Programm lautet

```
sum xs = if xs==[] then 0 else head xs + sum (tail xs)

main = sum [1,2,3]
```

und wir entfalten den Aufruf `sum [1,2,3]`. Dann erhalten wir

```
sum xs = if xs==[] then 0 else head xs + sum (tail xs)

main = if [1,2,3]==[]
       then 0
       else head [1,2,3] + sum (tail [1,2,3])
```

Eventuell müssen vor einem *Unfold*-Schritt gebundene Variablen im Rumpf der Funktionsdefinition umbenannt werden, sofern es Namenskonflikte zwischen diesen und den freien Variablen der Argumente des Aufrufs gibt.

*Beispiel 2:* Angenommen, wir entfalten den Aufruf `f y 0` im Programm

```
f x1 x2 = let y = g x2
          in h x1 y

e y = f y 0
```

ohne die von der Let-Anweisung gebundene Variable `y` umzubenennen. Dann erhalten wir das falsche Ergebnis

```
f x1 x2 = let y = g x2
          in h x1 y

e y = let y = g 0
      in h y y
```

Benennen wir dagegen `y` vorher in eine noch nicht verwendete Variable um (z. B. `y1`), so erhalten wir die korrekte Lösung

```
f x1 x2 = let y = g x2
          in h x1 y

e = let y1 = g 0
    in h y y1
```

3. Fold

   *Fold* ist die inverse Operation zu *Unfold* und ersetzt einen Ausdruck durch einen passenden Funktionsaufruf.

```
scalarproduct (x1,y1) (x2,y2) = x1*x2 + y1*y2

main = 3*a + 4*b
```

Der Ausdruck `3*a + 4*b` paßt hier genau auf den Rumpf der Funktion `scalar-product` und kann deswegen gefaltet werden. Wir erhalten

```
scalarproduct (x1,y1) (x2,y2) = x1*x2 + y1*y2

main = scalarproduct (3,4) (a,b)
```

Es sollte beachtet werden, daß durch ein *Fold* möglicherweise ein korrektes Programm in ein nichtterminierendes Programm verwandelt wird. Beispielsweise ist es immer möglich, einen kompletten Funktionsrumpf auf seinen eigenen Aufruf zu falten. Aus `f x = x+5` würde dadurch z. B. `f x = f x`.

Um die Korrektheit eines *Fold*-Schrittes zu garantieren, ist es hinreichend, die Terminierung des resultierenden Programms zu beweisen.

4. Apply Property

   *Apply Property* steht für eine ganze Reihe von Umformungen, die auf der Anwendung von Axiomen und Sätzen über die Konstrukte der verwendeten Programmiersprache oder über die im Programm verwendeten Datentypen basieren.

   Beispiele solcher Umformungen sind

```
(x:xs) == []              -->  False
head (x:xs)               -->  x
tail (x:xs)               -->  xs
if False then e1 else e2  -->  e2
```

   Der Rumpf der Funktion `main` in

```
sum xs = if xs==[] then 0 else head xs + sum (tail xs)

main = if [1,2,3]==[]
       then 0
       else head [1,2,3] + sum (tail [1,2,3])
```

   kann durch Anwendung einiger *Apply Property*-Schritte zu `1 + sum [2,3]` vereinfacht werden.

5. Instantiate

   *Instantiate* bedeutet, einen Funktionsaufruf für konkrete Argumente auszuwerten. Instantiieren wir z. B. `sum [1,2,3]` aus dem obigen Beispiel, so erhalten wir `6`.

   Technisch gesehen ist *Instantiate* lediglich eine Abkürzung für die sukzessive Anwendung von *Unfold* und *Apply Property*.

6. Abstract

   Ein *Abstract*-Schritt ist schließlich die Kombination aus der Einführung einer neuen Funktionsdefinition (*Define*) und einem anschließenden *Fold*, in dem die neu eingeführte Funktion verwendet wird.

   Oft wird dabei die neue Funktion lediglich lokal eingeführt, also innerhalb eines Let- oder Where-Konstrukts. Die Funktionsgleichung

```
distance (x1,y1) (x2,y2) = sqrt ((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))
```

   könnte man z. B. durch einen Abstraktionsschritt in

```
distance (x1,y1) (x2,y2) = let pyth a b = sqrt (a*a+b*b)
                           in pyth (x2-x1) (y2-y1)
```

   verwandeln. Hier wurde die Funktion `pyth` zuerst lokal definiert und anschließend zum Falten des Ausdrucks verwendet.

7. Specialize

   In [Thi94] wird noch eine siebte Transformation namens *Specialize* erwähnt. Ein *Specialize*-Schritt erzeugt eine spezialisierte Variante einer Funktionsgleichung. Wenn $f$ durch

$$f\ x_1\ ...\ x_n = body$$

   definiert ist und $p_1, ..., p_n$ Patterns sind, so daß jedes $p_i$ vom gleichen Typ wie das zugehörige $x_i$ ist, so ist

$$f\ p_1\ ...\ p_n = body'$$

   wobei in $body'$ die $x_i$ durch die entsprechenden $p_i$ ersetzt worden sind. Diese Gleichung kann nun der Funktionsdefinition hinzugefügt werden. Da sie auf allen Argumenten, auf die die Patterns $p_1, ..., p_n$ passen, das gleiche Ergebnis wie die ursprüngliche Funktion liefert, entsteht keine Mehrdeutigkeit.

   Wenn durch Spezialisierung aus einer Gleichung mehrere Gleichungen gewonnen worden sind, die zusammen alle Fälle abdecken, so kann die ursprüngliche Gleichung eliminiert werden.

   *Beispiel:* Die Funktionsdefinition

```
fib2 n = (fib n, fib (n+1))
```

   kann durch zwei Spezialisierungsschritte und anschließender Eliminierung der ursprünglichen Gleichung ersetzt werden durch

```
fib2 0     = (fib 0, fib 1)
fib2 (n+1) = (fib (n+1), fib (n+2))
```

   *Specialize* ist in der Literatur auch unter dem Namen *Case Introduction* [Par90] bekannt und kann auch als ein *Apply Property*-Schritt angesehen werden.

### 2.2.2 Beispielanwendung

Mit den Transformationen des Unfold/Fold-Systems kann die Effizienz eines Programms verbessert werden. Ein eindrucksvolles Beispiel dafür bietet die in [BD77] gezeigte Herleitung eines Linearzeit-Algorithmus für die Fibonacci-Funktion:

Wir gehen aus von der naiven Definition

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

Ein *Abstract*-Schritt, angewandt auf die dritte Gleichung, liefert

```
fib (n+2) = let (x,y) = (fib n, fib (n+1))
            in x+y
```

Wir definieren nun eine neue Funktion *fib2*

```
fib2 n = (fib n, fib (n+1))
```

und verwenden sie für einen *Fold*-Schritt

```
fib (n+2) = let (x,y) = fib2 n
            in x+y
```

Spezialisierung von fib2 ergibt

```
fib2 0     = (fib 0, fib 1)
fib2 (n+1) = (fib (n+1), fib (n+2))
```

Es folgen zwei *Instantiate*-Schritte von `fib 0` und tt fib 1.

```
fib2 0 = (0,1)
```

Dann entfalten wir den Aufruf `fib (n+2)` mit der ursprünglichen Definition von `fib`.

```
fib2 (n+1) = (fib (n+1), fib n + fib (n+1))
```

Ein weiterer *Abstract*-Schritt liefert

```
fib2 (n+1) = let (x,y) = (fib n, fib (n+1))
             in (y, x+y)
```

Schließlich kann `(fib n, fib (n+1))` nochmals zu `fib2` gefaltet werden.

```
fib2 (n+1) = let (x,y) = fib2 n
             in (y, x+y)
```

Das resultierende Programm lautet

```
fib 0     = 0
fib 1     = 1
fib (n+2) = let (x,y) = fib2 n
            in x+y

fib2 0     = (0,1)
fib2 (n+1) = let (x,y) = fib2 n
             in (y, x+y)
```

und berechnet die $n$-te Fibonacci-Zahl in linearer Zeit.

Die Operationen *Define*, *Unfold*, *Fold*, *Specialize*, *Instantiate* und *Abstract* kann man als grundlegende elementare Operationen der transformationellen Programmentwicklung sehen. Für die meisten Transformationen werden allerdings auch Transformationsschritte vom Typ *Apply Property* benötigt, von denen es potentiell unendlich viele Varianten gibt. Diese sollen nun genauer betrachtet werden. Dazu benötigen wir als erstes den Begriff des *Programmschemas.*

### 2.2.3 Programmschemata

Unter einem *Programm* verstehen wir jede Beschreibung einer bestimmten Berechnung in der Sprache Gofer. Dabei spielt es im folgenden keine Rolle, ob es sich bei unserem *Programm* um ein komplettes Programm im üblichen Sinne, nur um eine Funktionsdefinition oder lediglich um einen Ausdruck wie `3+4` handelt.

Durch Parametrisierung gelangen wir von *Programmen* zu *Programmschemata*. Ein einfaches Programmschema für linear rekursive Funktionen mit einem Argument ist z.B.

```
f n = if C n then T else K2 n (f (K1 n))
```

wobei $C$ (Abbruchbedingung), $T$ (Ergebnis im Terminierungsfall), $K1$ (Berechnung des neuen Arguments für den rekursiven Aufruf) und $K2$ (Modifikation des vom rekursiven Aufruf zuruückgelieferten Ergbnisses) sogenannte *Schemaparameter* sind und ihrerseits für beliebige Programmschemata stehen.

Durch eine *Instantiierung* werden die Schemaparameter eines Programmschemas belegt. Das Ergebnis, die *Instanz*, ist im allgemeinen Fall noch kein Programm, sondern lediglich ein spezielleres Programmschema. Eine Instanz, die keine Schemaparameter mehr enthält, nennt man *Grundinstanz*.

Die Instantiierung $\{C \rightarrow (<2), T \rightarrow 1, K1 \rightarrow (\lambda \rightarrow x-1), K2 \rightarrow (*)\}$ verwandelt obiges Programmschema in die Grundinstanz

```
f n = if n<2 then 1 else n * f (n - 1)
```

Zwei Programmschemata heißen äquivalent, wenn für alle Instantiierungen die resultierenden Instanzen äquivalent sind.

### 2.2.4 Transformationsregeln

Eine *Transformationsregel* ist eine partielle Abbildung von einem Programmschema zu einem anderen, so daß jede Anwendung der Regel auf eine Instanz des Eingabeschemas eine korrekte Transformation darstellt. Eine Transformation ist korrekt, wenn sie ein Programmschema auf ein äquivalentes Programmschema abbildet.

Repräsentiert werden kann eine Transformationsregel auf zwei Arten: als Funktion, die ein Programm als Eingabe nimmt und ein neues Programm produziert (prozedurale Regel) oder als ein Tripel $(C, I, O)$, in dem I und O zwei Programmschemata sind, die unter der Anwendbarkeitsbedingung C äquivalent sind (schematische Regel). Schematische Transformationsregeln sind also Inferenzen der Form $C \vdash I \equiv O$.

Üblicherweise versucht man, Regeln in der schematischen Form zu repräsentieren, sofern dies möglich ist. Für algebraische Identitäten zwischen Sprachkonstrukten (wie z. B. $\vdash$

$map\ (f\ .\ g) \equiv map\ f\ .\ map\ g$) oder Regeln, die Eigenschaften von Datentypen beschreiben (wie z. B. $\vdash a \wedge True \equiv a$ oder $\vdash pop(push\ x\ s) \equiv s$) eignet sich die schematische Form gut.

Transformationen, bei denen Berechnungen durchgeführt werden müssen, die über Pattern Matching und Variablensubstitution hinausgehen, sind dagegen nur in algorithmischer Form repräsentierbar. Ein gutes Beispiel hierfür ist ein Transformationsschritt, der die einzelnen Programmzweige auf Erreichbarkeit überprüft und unerreichbare Zweige löscht.

### 2.2.5 Anwendbarkeitsbedingungen

Anwendbarkeitsbedingungen von Transformationsregeln lassen sich in zwei Klassen unterteilen: syntaktische und semantische Anwendbarkeitsbedingungen.

#### 2.2.5.1 Syntaktische Anwendbarkeitsbedingungen

Syntaktische Anwendbarkeitsbedingungen haben generell mit Namenskonflikten zwischen freien und gebundenen Variablen zu tun. Ein Beispiel dafür haben wir bereits bei der *Unfold*-Operation gesehen, wo die gebundenen Variablen im Rumpf der Funktionsdefinition nicht mit den freien Variablen im Funktionsaufruf in Konflikt geraten durften.

Als weiteres Beispiel betrachten wir das Programmschema

```
F (case C of
        []     -> E1
        (x:xs) -> E2
  )
```

mit den Schemavariablen $F$, $C$, $E1$ und $E2$. Da eine Funktionsanwendung über ein Konditional distributiert, können wir $F$ ins Innere der Case-Anweisung ziehen. Ein äquivalentes Programmschema ist folglich

```
case C of
     []     -> F E1
     (x:xs) -> F E2
```

Doch hier ist Vorsicht angebracht! Die Äquivalenz gilt nämlich nicht, wenn eine der Variablen $x$ oder $xs$ frei in $F$ vorkommt. Die Instanz

```
x + case ys of
         []     -> 0
         (x:xs) -> 1
```

ist z.B. nicht äquivalent zu

```
case ys of
     []     -> x + 0
     (x:xs) -> x + 1
```

Eine Funktionsanwendung $F$ darf also nur dann in eine Case-Anweisung hineingezogen werden, wenn es keine Namenskonflikte zwischen den freien Variablen von $F$ und den Pattern-Variablen der einzelnen Alternativen gibt.

Die Überprüfung von syntaktischen Anwendbarkeitsbedingungen ist glücklicherweise leicht mechanisierbar. Durch entsprechende Variablenumbenennung (Alpha-Konversion) kann die Erfüllung syntaktischer Bedingungen außerdem erzwungen werden. Bei der Arbeit mit dem ULTRA-System hat sich der Benutzer deswegen nie um syntaktische Bedingungen zu kümmern. Sollten Namenskonflikte die Ausführung eines Transformationsschrittes verhindern, so benennt ULTRA die von Case-, Let- oder Lambda-Ausdrücken definierten Patternvariablen eigenständig um.

#### 2.2.5.2 Semantische Anwendbarkeitsbedingungen

Semantische Anwendbarkeitsbedingungen unterscheiden sich von den syntaktischen dadurch, daß zu ihrer Überprüfung Wissen über die Bedeutung der Symbole erforderlich ist.

Die Programmschemata $(A \oplus B) \oplus C$ und $A \oplus (B \oplus C)$ sind z. B. äquivalent, sofern der Operator $\oplus$ assoziativ ist. Doch dies ist leider nicht mehr rein syntaktisch feststellbar. Falls es sich bei $\oplus$ um einen der üblichen arithmetischen Operatoren (+) oder (*) handelt, ist der Fall zwar klar - falls es sich aber um einen benutzerdefinierten Operator handelt, muß dessen Assoziativitätseigenschaft bewiesen werden.

Im allgemeinen Fall sind semantische Anwendbarkeitsbedingungen Hornklauseln, die aus semantischen Prädikaten über Programmschemata aufgebaut sind.

Typische semantische Prädikate sind

- ASSOC($f$): $f$ ist assoziativ
- COMM($f$): $f$ ist kommutativ
- LDISTR($op1$,$op2$): $op1$ ist linksdistributiv über $op2$
- NEUTRAL($op$,$e$): $e$ ist ein neutrales Element bezüglich $op$

Ein Beispiel für eine Regel mit einer semantischen Anwendbarkeitsbedingung ist die oben erwähnte Assoziativitätsregel

$$ASSOC(\oplus) \vdash (A \oplus B) \oplus C \equiv A \oplus (B \oplus C)$$

mit den Schemavariablen $\oplus$, $A$, $B$ und $C$.

## 2.3 Transformationssysteme

Die transformationelle Programmentwicklung bietet eine Reihe von Vorteilen gegenüber der herkömmlichen Programmentwicklung. Leider ist sie aber auch um einiges aufwendiger. Die riesige Anzahl von Schritten, die bisweilen erforderlich ist, macht deswegen die schrittweise Programmentwicklung durch Transformationen ohne Systemunterstützung kaum praktikabel für reale Probleme.

Viele mechanische Aufgaben können dem Programmierer jedoch vom Rechner abgenommen werden. Dazu zählen die Bereitstellung einer Regeldatenbank, die Prüfung von syntaktischen und semantischen Anwendbarkeitsbedingungen und die korrekte Ausführung der Transformationsschritte. Auch sollte der Programmierer einzelne Schritte zurücknehmen können, um aus Sackgassen zurückzusetzen. Und schließlich sollte ein gutes Transformationssystem einige Simplifikationsstrategien beherrschen, um triviale Vereinfachungen von alleine vornehmen zu können. Der Programmierer kann sich somit auf die Tätigkeit konzentrieren, bei der Intuition und Kreativität gefragt ist, nämlich die Auswahl von erfolgversprechenden Regeln und Anwendungsstellen.

Transformationssysteme unterscheiden sich voneinander vor allem hinsichtlich zweier Kriterien: dem Spektrum der vom System transformierten Sprache und dem Ausmaß der Unterstützung, die das System dem Benutzer bietet.

- Spektrum der transformierten Sprache

  Das Spektrum der Programmiersprachen reicht von High-Level-Spezifikationssprachen mit deskriptiven Konstrukten wie Quantoren, Mengenkomprehensionen und nichtdeterministischen Operatoren bis zu maschinennahen imperativen Low-Level-Sprachen. Nur sehr wenige Programmiersprachen wie z.B. CIP-L (siehe Abschnitt 2.3.1.1) decken dabei das gesamte Spektrum ab.

  Die von ULTRA transformierte Sprache Gofer ist eine rein funktionale Sprache ohne deskriptive Konstrukte. ULTRA ist damit ein System, daß für die zweite Phase der transformationellen Programmentwicklung (Abb. 1.1), die Optimierung eines funktionalen Programms, gedacht ist.

- Ausmaß der Systemunterstützung

  Das Ausmaß an Interaktion, das vom Benutzer während einer Transformation verlangt wird, entscheidet oft darüber, ob das System für größere Probleme überhaupt eingesetzt werden kann. In der einfachsten Implementierung ist der Benutzer für jeden einzelnen Transformationsschritt verantwortlich. In vollautomatischen Systemen wird dagegen die Regelauswahl nach eingebauten heuristischen Regeln vom Rechner selbst vollzogen.

  ULTRA ist ein semi-automatisches System. Prinzipiell entscheidet der Benutzer über die anzuwendenden Regeln. Um die Arbeit zu erleichtern, gibt es allerdings den Simplifier, der die am häufigsten gebrauchten Vereinfachungsregeln wie partielle Evaluation, Eliminierung überflüssiger Fallunterscheidungen oder Löschen von unerreichbaren Programmzweigen automatisch durchführt, sowie die Möglichkeit, mehrere Transformationsregeln zu Strategien zu kombinieren.

### 2.3.1 Existierende Transformationssysteme

Transformationssysteme gibt es etwa seit Mitte der Siebziger Jahre. Eine ausführliche Bewertung verschiedener Systeme wird in [PS83] und [PV95] vorgenommen.

Auf drei Transformationssysteme, die vom Prinzip her ähnlich zu ULTRA sind, soll nun zum Abschluß dieses Kapitels kurz eingegangen werden.

#### 2.3.1.1 CIP-S

Das CIP-System wurde an der Technischen Universität München im Rahmen des CIP-Projekts enwickelt [BBB+85, BEH+87]. Die Implementierungssprache von CIP-S ist C. Die vom System transformierte Sprache CIP-L (Computer-aided Intuition-guided Programming Language) wurde speziell für die transformationelle Programmierung entworfen und deckt das gesamte Spektrum von der Spezifikation bis zur imperativen Sprache ab. CIP-S ist damit ein System für alle drei Phasen der transformationellen Programmentwicklung (Abb. 1.1). CIP-S ist darüber hinaus sogar sprachunabhängig. Die Sprache CIP-L könnte also leicht durch eine andere Sprache ersetzt werden. Diese Allgemeinheit hat jedoch auch ihren Preis. Die Forderung nach Sprachunabhängigkeit macht es unmöglich, einen guten automatischen Simplifier zu programmieren, da dieser nur auf benutzerdefinierte schematische Regeln und nicht auf im System „fest verdrahtete“ Algorithmen zurückgreifen kann.

#### 2.3.1.2 HOPS

Das *Higher Object Programming System (HOPS)* wurde 1984-1988 ebenfalls an der Technischen Universität München entwickelt. Ab 1988 wurde das Projekt an der Universität der Bundeswehr München fortgeführt [BK94]. Die Implementierungssprache von HOPS ist Smalltalk-80, die transformierte Sprache ist eine sehr kleine, speziell für das System entwickelte, funktionale Sprache. Programme werden in HOPS grafisch in Form von gerichteten Graphen dargestellt. Eine textuelle Darstellung von Programmen existiert nicht. Die Programmtransformation geschieht durch direkte Manipulation der Graphen mittels einer grafischen Benutzeroberfläche. Ein Transformationsschritt ist in HOPS also die Ersetzung eines Teilgraphen des Programms durch einen anderen Graphen.

#### 2.3.1.3 KIDS

Das *Kestrel Interactive Development System (KIDS)* wurde am Kestrel Institute entwickelt [Smi90]. KIDS ist ein in LISP geschriebenes sehr leistungsfähiges System und wurde in [PS96] erfolgreich zur Herleitung eines Transport-Scheduling-Algorithmus verwendet. KIDS bietet sehr mächtige Regeln an, wie z. B. *Finite Differencing*, *Partial Evaluation* oder *Data Type Refinement* [Par90]. Außerdem beherrscht KIDS eine Reihe von Design-Taktiken für Algorithmen wie *Backtracking* oder *Divide & Conquer*. Das System ist leider aber auch nicht leicht zu bedienen und erfordert daher ein erhebliches Maß an Schulung.

# Kapitel 3 Arbeiten mit dem ULTRA-System

Abbildung 3.1: *ULTRA in Aktion*

Dieses Kapitel führt in die Bedienung des ULTRA-Systems ein. Dabei wird erklärt, wie die einzelnen Transformationen des Unfold/Fold-Systems mit ULTRA realisiert werden können.

ULTRA wird über eine grafische Benutzeroberfläche gesteuert. Der Benutzer markiert einen Teil des zu transformierenden Programms und wählt die gewünschte Transformation. Das System führt dann die Transformation aus und stellt das modifizierte Programm wieder dar.

## 3.1 Start

Gestartet wird ULTRA vom Gofer Interpreter aus durch Aufruf der Funktion `main`. Es öffnet sich das in Abb. 3.2 gezeigte Hauptfenster.

Abbildung 3.2: *ULTRA direkt nach Programmstart*

Links oben befindet sich das Editierfenster, links unten das Transformationsfenster. Am rechten Rand befindet sich die Regelauswahlbox.

Während einer Sitzung mit dem ULTRA-System befindet man sich immer in einem von zwei möglichen Modi: dem Editiermodus oder dem Transformationsmodus. Nach dem Programmstart befindet man sich zunächst im Editiermodus. Welcher Modus gerade aktiv ist, erkennt man an den aktuellen Hintergrundfarben.

| Fenster | Hintergrundfarbe im Editiermodus | Hintergrundfarbe im Transformationsmodus |
|---|---|---|
| Editierfenster | weiß | gelb |
| Transformationsfenster | gelb | weiß |
| Regelauswahlbox | dunkelgrün | hellgrün |

Durch Drücken des *Quit Buttons* kann das Programm jederzeit wieder verlassen werden.

## 3.2 Das Editierfenster

Solange man sich im Editiermodus befindet, können im Editierfenster Gofer-Programme geschrieben werden. Grundlegende Funktionen wie Cut/Copy/Paste sowie Laden und Speichern stehen dabei zur Verfügung.

## 3.3 Starten einer Transformation

Will man eine Programmtransformation beginnen, so drückt man den *Start Transformation* Button. Das Programm im Editierfenster wird nun geparst und anschließend im Transformationsfenster mit dem Pretty-Printer ausgegeben. Außerdem geht das System in den Transformationsmodus über.

Will man nicht das komplette Programm, sondern nur einen Teil davon transformieren, so markiert man die gewünschten Programmzeilen im Editierfenster und drückt dann den *Start Transformation* Button.

## 3.4 Das Transformationsfenster

Befindet man sich im Transformationsmodus, so steht dem Benutzer in diesem Fenster eine kontextsensitive Textmarkierung zur Verfügung. Das heißt, es können nicht beliebige Teile des Programmtextes markiert werden, sondern nur wohlgeformte funktionale Ausdrücke.

## 3.5 Simplify

Wenn der *Simplify* Button gedrückt wird, so versucht ULTRA, den gerade markierten Ausdruck selbstständig zu vereinfachen. Das System benutzt hierzu die eingebaute Simplifikationsstrategie, die in Kapitel 5.5 erklärt wird.

Angenommen, wir haben im Editierfenster das Programm

```
test x = if x+3==9 then 1+2==3 else 4+5==6
```

eingegeben und dann eine Transformation gestartet.

Wir markieren nun den Term `1+2` und drücken den *Simplify* Button. Das System vereinfacht nun `1+2` zu `3` und gibt das Ergebnis wieder im Transformationsfenster aus. Wir erhalten also

```
test x = if x+3==9 then 3==3 else 4+5==6
```

Wir vereinfachen nun der Reihe nach `3==3` zu `True`, `4+5` zu `9`, `9==6` zu `False` und `x+3==9` zu `x==6`. Unser Programm sieht nun so aus:

```
test x = if x==6 then True else False
```

Im letzten Schritt markieren wir den gesamten Term `if x==6 then True else False` und drücken wieder den *Simplify* Button. Das Endergebnis lautet:

```
test x = x==6
```

Dieses Ergebnis hätten wir übrigens auch in einem Schritt erhalten können, wenn wir gleich zu Beginn den kompletten Term markiert und anschließend den *Simplify* Button gedrückt hätten.

## 3.6 Unfold

Einen Funktionsaufruf zu entfalten heißt, ihn durch den Rumpf der Funktionsdefinition zu ersetzen, wobei Argumente entsprechend substituiert werden.

Betrachten wir folgendes Programm:

```
f a b = a+b

test1 = f x 1

test2 = f 3 5

test3 = if f 4 7 < 9 then 1 else 2

test4 = f 10
```

Wir markieren den Ausdruck `f x 1` und drücken den *Unfold* Button. Das System führt nun ein Pattern Matching des markierten Ausdrucks mit der linken Seite der Definition von `f` durch. Das Pattern Matching endet erfolgreich und die gefundene Substitution lautet $\{a \to x, b \to 1\}$. Diese Substitution wird nun auf die rechte Seite angewandt und wir erhalten als Ergebnis

```
test1 = x+1
```

In vielen Fällen ist es möglich, nach einem *Unfold* das Ergebnis weiter zu vereinfachen. Deswegen wird nach jedem *Unfold* der Simplifier gestartet. Markieren wir den Term `f 3 5` und drücken wiederum *Unfold*, so evaluiert der Term nicht nur zu `3+5`, sondern gleich zu

```
test2 = 8
```

Ein *Unfold* eines Aufrufs mit konstanten Argumenten entspricht somit einem *Instantiate*-Schritt (siehe Kapitel 2.2.1).

Die möglichen Vereinfachungen nach einem *Unfold* wirken sich eventuell nicht nur auf den entfalteten Ausdruck, sondern auf dessen gesamten Kontext aus. Entfalten wir den Ausdruck `f 4 7`, so evaluiert die gesamte rechte Seite von `test3` zu

```
test3 = 2
```

Eine typische Eigenschaft funktionaler Sprachen ist die mögliche Unterversorgung von Funktionen. Ein Beispiel dafür bietet die Funktion `test4` mit der rechten Seite `f 10`. `f 10` paßt natürlich nicht auf die linke Seite `f a b`, weshalb man auf den ersten Blick meinen könnte, daß ein *Unfold* hier nicht möglich ist. Man kann aber die Gleichung `f a b = a+b` auch als Lambda-Abstraktion `f a = /b -> a+b` interpretieren. Genau dies macht ULTRA in diesem Fall, weswegen `f 10` durch ein *Unfold* zu

```
test4 = \b -> 10+b
```

evaluiert.

## 3.7 Fold

Durch Drücken des *Fold Buttons* wird der markierte Ausdruck gefaltet, d. h. er wird durch einen äquivalenten Funktionsaufruf ersetzt.

```
f a b = a + b

test1 x y = (3*x + 11) - y
```

Wird in diesem Programm der Ausdruck `3*x + 11` markiert und der *Fold* Button gedrückt, so erhält man:

```
f a b = a + b

test1 x y = f (3*x) 11 - y
```

Zur Bestimmung des passenden Funktionsaurufs, der einen markierten Ausdruck ersetzen könnte, werden nicht nur die Funktionsdefinitionen des aktuellen Programms herangezogen, sondern auch alle ehemaligen Versionen, die während der Entwicklung auftraten. Dies ermöglicht die Herleitung rekursiver Funktionsaufrufe, wie das folgende Beispiel zeigt.

```
ge a b = if b==0 then true
         else if a==0 then false
         else ge (a-1) (b-1)

sub a b = if b==0 then a
          else if a==0 then dummy
          else sub (a-1) (b-1)

gesub a b = (ge a b, sub a b)
```

Durch Entfalten der Funktionsaufrufe *ge* und *sub* innerhalb von *gesub* und die anschließenden Vereinfachungen gelangt man zu

```
gesub a b = if b==0 then (true, a)
            else if a==0 then (false, dummy)
            else (ge (a-1) (b-1), sub (a-1) (b-1))
```

Der Ausdruck `(ge (a-1) (b-1), sub (a-1) (b-1))` paßt nun zwar nicht mehr auf die rechte Seite der aktuellen Version von `gesub`; er paßt jedoch auf die rechte Seite der Version, bei der wir unsere Transformation begonnen haben - bei `gesub a b = (ge a b, sub a b)`. Versuchen wir den Ausdruck `(ge (a-1) (b-1), sub (a-1) (b-1))` zu falten, so wird ein Pattern Matching mit der aktuellen sowie allen ehemaligen Versionen von `gesub` durchgeführt. Das Pattern Matching mit `gesub a b = (ge a b, sub a b)` gelingt und die gefundene Substitution lautet $\{a \rightarrow a-1, b \rightarrow b-1\}$. Als Ergebnis erhalten wir

```
gesub a b = if b==0 then (true, a)
            else if a==0 then (false, dummy)
            else gesub (a-1) (b-1)
```

Wenn mehrere Funktionen als Kandidaten für ein *Fold* in Frage kommen, so wird der Benutzer gefragt, auf welchen Funktionsaufruf gefaltet werden soll. (Abb. 3.3)

Abbildung 3.3: *Auswahl der Funktion für den Fold-Schritt*

Wie in Kapitel 3 beschrieben, kann durch ein *Fold* ein korrektes Programm in ein nichtterminierendes Programm verwandelt werden. Um die Korrektheit eines *Fold*-Schrittes zu garantieren, ist es hinreichend, die Terminierung des resultierenden Programms zu beweisen. Das System liefert hierzu noch keine automatische Unterstützung. Der *Fold*-Schritt wird jedoch ins Protokoll der Transformation aufgenommen. Um die Korrektheit der gesamten Programmentwicklung zu beweisen, müßte der Benutzer später den entsprechenden Terminierungsbeweis ins Protokoll einfügen.

Da das Falten eines kompletten Funktionsrumpfes auf den eigenen Funktionsaufruf immer zur Nichtterminierung führt, wird dies von ULTRA gar nicht erst zugelassen.

## 3.8 Die Regelauswahlbox

Die Regelauswahlbox enthält vier verschiedene Typen von Einträgen: Taktiken (im Programmcode fest verankerte Transformationsregeln), Regeln (benutzerdefinierte Transformationsregeln), Regelkataloge und Strategien.

Regelkataloge sind Sammlungen von Regeln und Strategien. Durch einen Mausklick kann man einen Katalog öffnen bzw. schließen. Mit dem *View Rule Definition* Button kann man sich eine Transformationsregel oder Strategie anzeigen lassen.

### 3.8.1 Anwendung von Regeln

Um eine Regel anzuwenden, markiert man im Transformationsfenster den zu transformierenden Ausdruck und wählt in der Regelauswahlbox die gewünschte Regel an. Durch Drücken des *Apply* Buttons wird die Regelanwendung initiiert. Ist die Regelanwendung möglich, so erscheint kurz darauf das Ergebnis im Transformationsfenster, wenn nicht, so wird dem Benutzer der Grund dafür mitgeteilt. Meistens verläuft eine Regelanwendung automatisch, in einigen Fällen ist jedoch Interaktion seitens des Benutzers erforderlich.

Betrachten wir dazu die in Abbildung 3.4 gezeigte Regel *add neutral element left*.

Abbildung 3.4: *Definition der Regel add neutral element left*

Wie am Eingabeschema der Regel erkennbar ist, ist diese Regel auf jeden beliebigen Ausdruck *expr* anwendbar. Die Schemavariablen *op* und *e* kommen im Input-Schema nicht vor, werden also nicht durch Pattern Matching bestimmt, sondern können beliebig gewählt werden, sofern die Bedingung „*e* ist neutrales Element bezüglich der Operation *op*“ erfüllt ist. Nun gibt es natürlich mehrere Möglichkeiten, eine solche Wahl zu treffen. $\{op = (+), e = 0\}$ wäre eine davon, $\{op = (*), e = 1\}$ eine weitere. ULTRA überläßt deswegen dem Benutzer die Entscheidung. Die möglichen Instanzen des Output-Schemas, die das System aus seiner Wissensbasis ableiten kann, werden dem User in einer Auswahlbox präsentiert (Abb. 3.5). Nachdem der Benutzer seine Wahl getroffen hat, wird der Transformationsschritt zu Ende geführt.

Es werden nur Instanzen zur Auswahl angeboten, die aus der Wissensbasis abgeleitet werden können. Da zur Zeit noch keine Möglichkeit besteht, die Wissensdatenbank während einer Transformation um die fehlenden Gesetze zu erweitern, sollte man dies erledigen, bevor man die Programmtransformation beginnt.

Eine andere Variante von Interaktivität hat man bei der Regel *caseIntro*1 zur Einführung

Abbildung 3.5: *Auswahl des gewünschten Ergebnisses bei Anwendung der Regel „add neutral element left" auf den Term $x$*

einer Fallunterscheidung.

Abbildung 3.6: *Definition der Regel* $caseIntro1$

Hier gibt es für die Schemavariable $c$ unendlich viele Instantiierungen. Eine Auswahlbox der ableitbaren Instanzen zusammenzustellen, ist folglich unmöglich. Stattdessen wird der Benutzer hier aufgefordert, die gewünschte Instantiierung über die Tastatur einzugeben. Wendet man beispielsweise auf den Term 42 die Regel $caseIntro1$ an, so öffnet sich das in Abb. 3.7 gezeigte Fenster.

Für jede Schemavariable, die noch unbelegt ist, gibt es ein Eingabefeld - in diesem Fall also genau eines, nämlich für $c$. Der Benutzer kann nun in den Eingabefeldern die gewünschten Ausdrücke eingeben, mit denen die Schemavariablen belegt werden sollen. Wenn der Benutzer den *OK* Button drückt, werden die eingegebenen Ausdrücke geparst. Sind alle Ausdrücke syntaktisch korrekt, so werden sie für die Schemavariablen ins Ausgabeschema eingesetzt. Dann schließt sich das Fenster und der Transformationsschritt wird zu Ende geführt.

Mit Transformationsregeln wie *caseIntro1* können *Specialize*-Schritte (siehe Kapitel 2.2.1) realisiert werden.

Abbildung 3.7: *Aufforderung zur Instantiierung bei Anwendung der Regel caseIntro1 auf den Term 42*

### 3.8.2 Regeln rückwärts anwenden

Da alle in ULTRA verwendeten Transformationsregeln Äquivalenzen zwischen Programmschemata beschreiben, können sie auch rückwärts angewandt werden. Der *Apply Backwards* Button hat die gleiche Funktion wie der *Apply* Button, nur werden vor der Regelanwendung Eingabe- und Ausgabeschema der Regel miteinander vertauscht. Eine *case introduction* wird damit zur *case elimination*, ein *assocL* zu *assocR*.

## 3.9 Define

Mit *Define* kann man während einer Transformation eine neue Funktionsdeklaration einführen. Durch die Kombination von *Define* und *Fold* kann ein *Abstract*-Schritt (siehe Kapitel 2.2.1) realisiert werden.

*Beispiel:* Angenommen, unser aktuelles Programm lautet

```
div a b = if a < b
          then 0
          else 1 + div (a - b) b

mod a b = if a < b
          then a
          else mod (a - b) b

divmod a b = (div a b, mod a b)
```

Unser Ziel ist die Herleitung einer rekursiven, von `div` und `mod` unabhängigen Version von `divmod`. Das Entfalten von `div` und `mod` in `divmod` und die danach erfolgenden automatischen Vereinfachungen liefern

```
divmod a b = if a < b
             then (0,a)
```

```
                else (1 + div (a-b) b, mod (a-b) b)
```

Um die Struktur der beiden Komponenten im Tupel `(1 + div (a-b) b, mod (a-b) b)` anzugleichen, fügen wir mit der Regel *add neutral element left* ein `0+` hinzu und erhalten

```
divmod a b = if a < b
             then (0,a)
             else (1 + div (a-b) b, 0 + mod (a-b) b)
```

Um nun den Rumpf der Funktion `divmod` falten zu können, benötigen wir die Additionsfunktion auf Tupeln. Wir drücken den *Define* Button und geben als neue Funktion ein:

```
tupleplus (a1,a2) (b1,b2) = (a1+b1, a2+b2)
```

Abbildung 3.8: *Define*

Nach Drücken von *Ok* wird die Funktion unserem Programm hinzugefügt. Nun können wir den Ausdruck `(1 + div (a-b) b, 0 + mod (a-b) b)` selektieren und mit *Fold* zu `tupleplus (1,0) (div (a-b) b, mod (a-b) b)` falten.

Im letzten Schritt falten wir `(div (a-b) b, mod (a-b) b)` zu `divmod (a-b) b` und haben damit unser Ziel erreicht.

```
divmod a b = if a < b
             then (0,a)
             else tupleplus (1,0) (divmod (a-b) b)

tupleplus (a1,a2) (b1,b2) = (a1+b1, a2+b2)
```

## 3.10 Replace

Mit *Replace* kann ein Ausdruck durch einen beliebigen anderen Ausdruck ersetzt werden. Drückt man den *Replace* Button, so öffnet sich ein Fenster und das System fragt, durch welchen äquivalenten Ausdruck der aktuell markierte Ausdruck ersetzt werden soll. (Abb. 3.9)

Dies ermöglicht es, zusätzliches Wissen in eine Transformation einzubringen, ohne daß man abbrechen und die entsprechende Transformationsregel definieren muß. Man sollte sich aber vor Augen halten, daß jede Anwendung von *Replace* eine potentielle Fehlerquelle in einer Transformation darstellt. Technisch gesehen ist ein *Replace*-Schritt eine Anwendung der Regel

$$a \equiv b \vdash a \equiv b$$

Abbildung 3.9: *Replace*

Um am Ende die Korrektheit des abgeleiteten Programms nachzuweisen, muß also für jeden *Replace*- Schritt die Äquivalenz des ersetzten Ausdrucks mit dem neuen Ausdruck bewiesen werden. Das System liefert hierzu noch keine richtige Unterstützung. Es werden allerdings einige einfache Überprüfungen durchgeführt (z. B. ob eine leere Liste durch eine nicht-leere ersetzt werden soll). Wenn ULTRA die Ausdrücke als offensichtlich nicht äquivalent erkennt, so erhält der Benutzer die Meldung „*This is not an equivalent expression!*". Ansonsten wird der Transformationsschritt zugelassen und ins Protokoll aufgenommen.

## 3.11 Navigation mit Back/Forward

Mit den beiden Buttons *Back* und *Forward* kann man in der Geschichte der aktuellen Entwicklung vor- und zurücknavigieren. Damit hat man die Möglichkeit, während einer Transformation aus Sackgassen zurückzusetzen und andere Wege einzuschlagen.

## 3.12 Hint

Wird der *Hint* Button gedrückt, so werden alle Regeln, die auf den gerade markierten Ausdruck anwendbar sind, in einem Fenster angezeigt. (Abb. 3.10) Dies ist dann nützlich, wenn man eine bestimmte Transformationsregel im Kopf hat, die man auf einen Ausdruck anwenden will, deren genauen Namen man aber vergessen hat oder von der man nicht weiß, in welchem Regelkatalog sie steht.

Abbildung 3.10: *Anzeige der anwendbaren Regeln nach Drücken des Hint Buttons*

## 3.13 Beenden einer Transformation

Der Transformationsmodus kann auf zwei Arten verlassen werden. Mit *Cancel* wird die Transformation abgebrochen. Das Programm im Editierfenster bleibt dabei unverändert. Mit *Done* wird die Transformation erfolgreich beendet. Das Programm im Editierfenster bzw. der markierte Programmteil wird dann gelöscht und durch das Programm im Transformationsfenster ersetzt.

## 3.14 Protokoll

Alle Transformationsschritte, die der Benutzer während einer Sitzung mit ULTRA durchführt, werden protokolliert und in die Datei *protocol* ausgegeben.

Das Protokoll kann als Dokumentation der Programmentwicklung dienen. Um es in einen Korrektheitsbeweis für das hergeleitete Programm zu verwandeln, muß der Benutzer noch für alle *Fold*- und *Replace*-Schritte die nötigen Beweise nachliefern.

Am Anfang des Protokolls wird das Programm ausgegeben, wie es zu Beginn der Transformation aussah. Dann folgt für jeden Transformationsschritt ein Header, der den Schritt beschreibt, und das veränderte Programm nach Ausführung des Schritts.

```
----------------------------------------------------------------------
-- Focus: <Funktionsname>
-- Action: <Aktion>
----------------------------------------------------------------------

<Neues Programm>
```

*Focus* nennt den Namen der Funktion, deren Rumpf sich im nächsten Schritt verändert. *Action* beschreibt die Art des Schritts (Unfold, Fold, Simplify, Regelanwendung, usw.).

Einige kommentierte Beispielprotokolle befinden sich im Anhang A.

# Kapitel 4 Benutzerdefinierte Erweiterungen

Der Benutzer hat die Möglichkeit, ULTRA durch Hinzufügen von eigenen Transformationsregeln oder neuen algebraischen Gesetzen zu erweitern, ohne den Quellcode des Systems zu ändern.

Im Verzeichnis THEORIES befindet sich eine Datei *thy.p*, die die Namen der einzelnen Regelkataloge enthält. Nach dem Programmstart werden alle hier aufgeführten Kataloge eingelesen. Für jeden Regelkatalog existiert eine Datei, die einen oder mehrere der folgenden Abschnitte enthält.

- Programs
- Rules
- Strategies
- Laws

Die Syntax dieser Dateien gehorcht der Gofer-Grammatik. Dadurch ist es möglich, sie mit dem Gofer-Interpreter einer Typ-Überprüfung zu unterziehen. (siehe Abschnitt 4.5) Insgesamt haben die Dateien folgende Gestalt:

```
-- Programs ------------------------------------------------

{siehe Abschnitt Funktionsdefinitionen}

-- Rules ---------------------------------------------------

{siehe Abschnitt Regeldefinitionen}

-- Strategies ----------------------------------------------

{siehe Abschnitt Strategien}

-- Laws ----------------------------------------------------

{siehe Abschnitt Algebraische Gesetze}
```

## 4.1 Funktionsdefinitionen

Der Abschnitt *Programs* ist im Prinzip einfach ein Gofer-Programm, das nach Programmstart eingelesen wird. Alle hier definierten Funktionen sind dem System später bekannt und werden als Kandidaten für *Fold* und *Unfold* in Betracht gezogen.

Die Typ-Signaturen der Funktionen werden beim Einlesen ignoriert und sind deswegen optional.

## 4.2 Regeldefinitionen

Im Abschnitt *Rules* befinden sich die eigentlichen Transformationsregeln, die in folgender Syntax geschrieben werden:

```
ruleName args = InputScheme =:= OutputScheme
```

oder wenn eine Anwendbarkeitsbedingung vorliegt:

```
ruleName args = Clause =>> InputScheme =:= OutputScheme
```

Die Syntax für Anwendbarkeitsbedingungen ist dabei

```
Clause ::= Predicate
         | Predicate /\ Clause
```

Die Argumente der Regel (`args`) sind die Schemavariablen, die im Eingabeschema, Ausgabeschema oder in der Bedingung der Regel vorkommen.

- *Beispiel 1:* Eine Regel, die den Then- und Else-Zweig einer If-Anweisung vertauscht und die Bedingung negiert

  ```
  ifSwap a b c = if (not c) then a else b =:= if c then b else a
  ```

  `a`, `b` und `c` müssen hier als Schemavariablen deklariert sein, da sie für beliebige Ausdrücke stehen.

- *Beispiel 2:* Eine Regel, die ausdrückt, daß die Summe einer Liste von Zahlen unverändert bleibt, wenn man die Liste umdreht

  ```
  sumReverse = sum . reverse =:= sum
  ```

  Hier wäre es falsch, `sum` und `reverse` als Schemavariablen anzugeben. Die Regel

  ```
  sumReverse sum reverse = sum . reverse =:= sum
  ```

  würde ausdrücken, daß es bei einer Funktionskomposition `f . g` nicht auf `g` ankommt, da `f . g = f` für alle `f` und `g` gilt - was natürlich nicht stimmt.

- *Beispiel 3:* Eine Konversionsregel zwischen den Fold-Operatoren *foldl* und *foldr*

```
foldlr op e = assoc op /\ neutral op e =>>
              foldl op e =:= foldr op e
```

`op` und `e` müssen als Schemavariablen deklariert sein. Zusätzlich benötigen wir hier aber eine semantische Bedingung, um `op` und `e` einzuschränken. `foldl op e` und `foldr op e` sind nämlich nur dann äquivalent, wenn `op` assoziativ ist und `e` das neutrale Element von `op` ist.

Eine kürzere Schreibweise wäre gewesen:

```
foldlr op e = monoid op e =>> foldl op e =:= foldr op e
```

- *Beispiel 4:* Eta-Abstraktion

  ```
  etaAbstraction f = f =:= \x -> f x
  ```

  Hier wird `x` nicht als Schemavariable angegeben, da die Variable von der Lambda-Abstraktion gebunden wird. (Generell kann nur eine freie Variable eine Schemavariable sein.)

  Der Name der gebundenen Variable wird bei einer Regelanwendung als Default-Name verwendet. Wenn die obige Regel also auf `g` angewandt wird, so ist das Ergebnis $\lambda$`x -> g x`.

  Tritt allerdings `x` frei in `f` auf, so ergänzt ULTRA den Default-Namen um eine Zahl, so daß der Name einer noch nicht verwendeten Variablen entsteht. Auf den Ausdruck `x x1` angewandt, liefert die Regel das Ergebnis $\lambda$`x2 -> x x1 x2`.

- *Beispiel 5:* Eta-Reduktion

  ```
  etaReduction f = \x -> f x =:= f
  ```

  Gebundene Variablen im Eingabeschema müssen ebenso nicht als Schemavariablen angegeben werden, da diese automatisch als solche angesehen werden. Die hier definierte Regel `etaReduction` ist also auch dann anwendbar, wenn für die Variable `x` ein anderer Name verwendet wurde.

Zu beachten ist noch, daß die Definitionen der Funktionen, die in den Transformationsregeln frei vorkommen, im Abschnitt *Programs* aufgeführt werden müssen. Wenn also die in Beispiel 2 verwendete Regel `sumReverse` im Abschnitt *Rules* steht, müssen die Definitionen der Funktionen `sum` und `reverse` im Abschnitt *Programs* stehen.

Will man Transformationen mit abstrakten Funktionen durchführen, so sollte man diese z. B. als

```
abstractFunction = error "not executable"
```

definieren. Dies macht dann Sinn, wenn man ein Programm transformieren will, in dem man manche Funktionen noch nicht implementiert hat, aber schon Transformationsregeln für diese Funktionen entworfen hat. Möglicherweise fallen ja diese sogenannten *abstrakten Funktionen* bei der Transformation durch die Vereinfachungen raus, und man spart sich ihre Implementierung.

## 4.3 Algebraische Gesetze

Der Abschnitt *Theories* enthält die Definitionen der semantischen Prädikate, die in den Anwendbarkeitsbedingungen der Transformationsregeln verwendet werden können. Definiert werden diese Prädikate über *Prolog-Klauseln*, die lediglich syntaktisch an Gofer angepaßt sind.

Prädikate, die grundlegende algebraische Eigenschaften von Funktionen darstellen, werden als Fakten aufgeschrieben:

```
assoc (.)  = Yes
assoc (++) = Yes
assoc (+)  = Yes
assoc (*)  = Yes
assoc min  = Yes
assoc max  = Yes

comm  (+)  = Yes
comm  (*)  = Yes
comm  (==) = Yes
comm  (&&) = Yes
comm  (||) = Yes

neutral (*)  1  = Yes
neutral (+)  0  = Yes
neutral (++) [] = Yes

transitive (<)  = Yes
transitive (>)  = Yes
transitive (==) = Yes
```

Komplexere Prädikate können über Hornklauseln definiert werden, d. h. durch UND-Verknüpfung aus einfachen Prädikaten zusammengesetzt werden.

```
monoid f e     = neutral f e /\ assoc f
abelgroup f e  = monoid f e /\ comm f
equivalence a  = transitive a /\ symmetric a /\ reflexive a
```

## 4.4 Strategien

Situationen wie „Wende auf diesen Ausdruck erst Regel A und dann Regel B an“, „Wende Regel X auf alle Teilterme dieses Ausdrucks an“ oder „Versuche alle Regeln dieses Katalogs in dieser Reihenfolge und wende die erste Regel an, die paßt“ treten relativ häufig auf.

Um sich hier unnötige Arbeit zu ersparen, können Transformationsregeln auf verschiedene Arten zu Strategien kombiniert werden. Dazu stehen folgende Kombinatoren zur Verfügung.

| | |
|---|---|
| REVERSE *name* | Regel *name* rückwärts anwenden, d. h. Ein- und Ausgabeschema vertauschen |
| $s1$ ANDTHEN $s2$ | zuerst Strategie $s1$ anwenden, bei Erfolg anschließend $s2$ auf das Ergebnis anwenden |
| $s1$ ORELSE $s2$ | zuerst Strategie $s1$ versuchen, bei Mißerfolg $s2$ versuchen |
| SUBTREES $s$ | Strategie $s$ (nicht rekursiv) auf die (direkten) Teilterme eines Ausdrucks anwenden |
| DFS $s$ | Strategie $s$ zuerst rekursiv auf die Teilterme eines Ausdrucks anwenden und dann auf den gesamten Ausdruck |
| REPEAT $s$ | Strategie $s$ solange iteriert auf einen Ausdruck anwenden, bis er sich nicht mehr ändert |
| CALL *name* | Aufruf der Substrategie *name*. Dies ermöglicht zum einen die Strukturierung von komplexen Strategien, zum anderen schafft es die Möglichkeit der Rekursion |

Mit Hilfe dieser Kombinatoren können beliebig komplexe Strategien geformt werden. Bei fehlender Klammerung sind ANDTHEN and ORELSE linksassoziativ und haben niedrigere Priorität als die unären Kombinatoren SUBTREES, DFS, CALL, REPEAT und REVERSE.

*Beispiel 1:* Eine Menge von Regeln $\{r1, r2, r3, ...\}$ soll in einer vorgegebenen Reihenfolge getestet werden. Die erste anwendbare Regel soll ausgeführt werden. Dann soll wieder von vorne angefangen werden. Wenn keine anwendbare Regel mehr gefunden wird, soll die Transformation stoppen. Diesen Algorithmus könnte man einfach aufschreiben als

```
repeatedTryAll = "REPEAT (r1 ORELSE r2 ORELSE r3 ORELSE ...)"
```

*Beispiel 2:* Der Ausdruck `((((x+1)+1)+1)+1)+1` soll zu `x+5` vereinfacht werden. Dazu muß die Klammerung geändert werden, so daß das `x` ganz nach außen gezogen wird. Um nicht fünfmal die Regel *assocR* anwenden zu müssen, definieren wir eine Strategie

```
depthFirstAssocR = "DFS assocR"
```

und wenden diese einmal an. Das Ergebnis lautet `x+((((((((1+1)+1)+1)+1)`. Nun enthält die rechte Seite dieses Ausdrucks keine freien Variablen mehr und kann mit dem automatischen Simplifier zu `5` vereinfacht werden. (Es sei an dieser Stelle angemerkt, daß der Simplifier in diesem speziellen Beispiel durchaus in der Lage gewesen wäre, gleich den gesamten Ausdruck zu `x+5` zu vereinfachen.)

Eine Transformation, in der eine komplexere Strategie zur Anwendung kommt, wird in Anhang A.3 gezeigt.

In der aktuellen Implementierung von ULTRA müssen Strategien, wie in den Beispielen gezeigt, als Strings angegeben werden. Für eine zukünftige Version ist geplant, dafür die normale funktionale Schreibweise zu verwenden. Die Syntax der beiden obigen Beispiele wird dann vermutlich lauten:

```
repeatedTryAll = doRepeat (r1 `orElse` r2 `orElse` r3 `orElse` ...)

depthFirstAssocR = doDFS assocR
```

## 4.5 Korrektheitsüberprüfung der Regelkataloge

Da alle mit ULTRA durchgeführten Transformationen nur korrekt modulo der Korrektheit der vorgegebenen Regeln sind, sollte der Benutzer keine leichtfertigen Änderungen in den Regelkatalogen vornehmen.

Schön wäre natürlich ein automatischer Test, der die benutzerdefinierten Transformationsregeln auf Korrektheit überprüft. Im vollen Umfang ist so ein Test jedoch unmöglich, da das Äquivalenzproblem für Programme (und damit natürlich auch für Programmschemata) unentscheidbar ist.

Zu Verdeutlichung des Problems betrachte man nur einmal folgende Transformationsregel:

```
myNewRule = let collatz n = if n<2 then True
                            else if even n then collatz (n/2)
                                           else collatz (3*n+1)
            in collatz
            =:=
            const True
```

Ein Korrektheitsbeweis dieser Regel erfordert einen Terminierungsbeweis der in der Regel verwendeten Funktion `collatz`. Dies ist bis heute noch niemandem gelungen. Andererseits ist auch keine Zahl $n$ bekannt, für die *collatz* $n$ nicht terminiert. (Näheres hierzu findet man in [Lag96].)

Wir müssen uns folglich auf einen Test beschränken, dessen Bestehen für die Korrektheit einer Transformationsregel notwendig, aber nicht hinreichend, ist. Ein erster Schritt in diese Richtung ist ein Test auf *syntaktische Korrektheit* und auf *Typkorrektheit*. Eine Regel wie

```
wrongRule f g = map (f . g) =:= (map f) . g
```

ist z. B. offensichtlich nicht korrekt, da Ein- und Ausgabeschema nicht vom selben Typ sind.

Ein Test auf Typkorrektheit läßt sich aber einfach durchführen, indem man die Regelkataloge probeweise in den Gofer-Interpreter lädt. Dazu muß der Interpreter lediglich die Typisierungen der Operatoren `/\`, `=:=` und `=>>` kennen. Diese sind im ersten Regelkatalog (*basic.thy*) angegeben und lauten:

```
infix 1 =:=
infix 0 =>>
infixr 3 /\

data Clause = Yes | No

(/\) :: Clause -> Clause -> Clause
```

```
(/\) = error "not executable"

(=:=) :: a -> a -> a
(=:=) = error "not executable"

(=>>) :: Clause -> a -> a
(=>>) = error "not executable"
```

Um die Kataloge zu überprüfen, startet man den Gofer-Interpreter, wechselt ins Verzeichnis THEORIES und gibt „:p thy.p“ ein. Werden die Dateien dann fehlerfrei eingeladen, so sind alle Regeln syntaktisch korrekt und typkorrekt.

Bei der Regel `wrongRule` aus dem obigen Beispiel würde der Gofer-Interpreter dagegen folgende Fehlermeldung ausgeben:

```
Reading script file "example.thy":
Type checking
ERROR "example.thy" (line 23): Type error in application
*** expression     : f . g
*** term           : f
*** type           : a -> b
*** does not match : c a -> b
*** because        : unification would give infinite type
```

# Kapitel 5 Implementierung

Dieses Kapitel beschreibt, wie das ULTRA-System implementiert wurde. Dabei wird vor allem auf die dabei aufgetretenen verschiedenen Probleme und deren Lösung eingegangen. In Abschnitt 5.1 wird die Definition der abstrakten Syntax, auf der die Transformationen durchgeführt werden, vorgestellt. Abschnitt 5.2 beschreibt den Parser, der ein Gofer-Quellprogramm in die abstrakte Syntax umwandelt. Auf den Pretty-Printer, der ein in abstrakter Syntax gegebenes Programm wieder zurück in die konkrete Syntax umwandelt und formatiert ausgibt, wird gleich darauf in Abschnitt 5.3 eingegangen. Abschnitt 5.4 beschreibt den eigentlichen Kern des Systems, der für die Ausführung der Transformationen zuständig ist. Die Strategie, nach der der Simplifier vorgeht, um Ausdrücke zu vereinfachen, wird in Abschnitt 5.5 erklärt. Schließlich wird in Abschnitt 5.6 kurz auf die grafische Benutzeroberfläche des Systems eingegangen.

## 5.1 Definition der abstrakten Syntax

Die Definition der abstrakten Syntax, d. h. der Entwurf geeigneter Datentypen zur Repräsentation von Ausdrücken, Funktionsdefinitionen und Programmen war die erste Entwurfsentscheidung bei der Implementierung von ULTRA.

Die abstrakte Syntax sollte möglichst einfach sein, da später die Programmtransformationen auf ihr ausgeführt werden. Würde man also z. B. zwischen If- und Case-Konstrukten oder zwischen Let- und Where-Anweisungen unterscheiden, so müßte man später auch jeweils für beide Varianten die entsprechenden Transformationen definieren.

Andererseits sollte die abstrakte Syntax ausdrucksstark genug sein, um Programme beim Parsen nicht bis zur Unkenntlichkeit „heruntercompilieren“ zu müssen. Theoretisch kommt man ganz ohne Let- und Case-Konstrukte aus, da diese in Lambda-Ausdrücke übersetzt werden können - allerdings würde dabei die Originalstruktur des Programms verlorengehen. Nach dem Unparsen würde man den Quellcode dann kaum wiedererkennen.

### 5.1.1 Ausdrücke

Der Datentyp für Ausdrücke ist angelehnt an den Vorschlag, der im Kapitel „*Data types for the Core Language*“ in [PJL92] gemacht wird.

```
data Expression
   = EVar    Name                          -- variable
   | ECon    Name [Expression]             -- constructor
   | EConst  Constant                      -- constant
   | EAp     Expression   Expression       -- application
   | ELam    [Expression] Expression       -- lambda abstraction
   | ECase   Expression   [Alter]          -- case expression
   | ELet    [Eqn]        Expression       -- let expression
   | EWild                                 -- wildcard
   | EUndef                                -- undefined expression
   | EPara Name                            -- program scheme parameter

type Alter = (Expression, Expression)   -- pattern & right-hand-side
type Eqn   = (Expression, Expression)   -- left & right-hand-side
```

Gelesen werden kann diese Definition in etwa als „Ein Ausdruck ist entweder eine Variable, die aus einem Namen besteht, oder ein Konstruktor, der aus einem Namen und einer Liste von Ausdrücken besteht,..., oder eine Case-Anweisung, die aus einer Bedingung und einer Liste von Alternativen besteht, oder...“.

Im Gegensatz zu dem in [PJL92] vorgeschlagenen Datentyp sind hier beliebige Ausdrücke auf der linken Seite von Gleichungen in Let-Konstrukten erlaubt. Ebenso sind für formale Parameter in Lambda-Abstraktionen beliebige Patterns und nicht nur Variablen zugelassen. Dadurch können mit der abstrakten Syntax alle Let-Konstrukte und Lambda-Abstraktionen repräsentiert werden, die in Gofer ausgedrückt werden können.

*Beispiel 1:* Die abstrakte Syntax von `let f x = x+3 in f 4` lautet

```
ELet [(EAp (EVar "f") (EVar "x"),
       EAp (EAp (EConst (CFun "+")) (EVar "x")) (EConst (CNum 3)))]
     (EAp (EVar "f") (EConst (CNum 4)))
```

Abbildung 5.1: *Syntaxbaum von let f x = x+3 in f 4*

*Beispiel 2:* Die abstrakte Syntax von $\lambda$`(x,y) -> x+y` lautet

```
ELam [ECon "Tuple" [EVar "x", EVar "y"]]
     (EAp (EAp (EConst (CFun "+")) (EVar "x")) (EVar "y"))
```

Abbildung 5.2: *Syntaxbaum von* $\lambda(x+y) \rightarrow x+y$

Beide Ausdrücke wären mit dem Datentyp

```
data Expression
   = ...
   | ELam [Name] Expression               -- lambda abstraction
   | ELet [(Name,Expression)] Expression  -- let expression
   | ...
```

nicht repräsentierbar gewesen.

### 5.1.2 Funktionsdefinitionen

Eine Funktionsdefinition, in [PJL92] *Supercombinator* genannt, ist ein Tripel, bestehend aus dem Funktionsnamen, einer Liste von formalen Parametern und dem Funktionsrumpf:

```
type SCDefn = (Name, [Expression], Expression)
```

Dagegen wäre der Typ

```
type SCDefn = (Name, [Name], Expression)
```

wiederum nicht allgemein genug, da als formale Parameter nicht nur Variablennamen, sondern auch Patterns zugelassen sind.

*Anmerkung:* Der Typ `Expression` ist für formale Parameter eigentlich zu allgemein und könnte theoretisch auch durch einen eingeschränkteren Typ ersetzt werden. Denn If-Konstrukte sind als Parameter z. B. nicht erlaubt. Die Korrektheit wird hier durch den Parser garantiert, der eine Funktionsdefinition wie `f (if ...)  = ...` nicht parst, obwohl sie mit der abstrakten Syntax theoretisch darstellbar wäre.

### 5.1.3 Programme

Ein Programm ist eine Liste von Funktionsdefinitionen.

```
type Program = [SCDefn]
```

### 5.1.4 Konstanten

Es fehlt noch der Datentyp für Konstanten. Von diesen gibt es fünf Arten: ganze Zahlen, Boolesche Werte, eingebaute Funktionen (Primitiven), String- und Character-Literale. Fließkommazahlen sind noch nicht implementiert.

```
data Constant = CNum Int
              | CBool Bool
              | CFun Name
              | CString String
              | CChar Char
```

Strings könnte man auch weglassen und sie stattdessen als Listen von Characters verwalten. Die Ausdrücke

```
EConst (CString "hi")
```

und

```
EAp (EAp (EConst (CFun ":")) (EConst (CChar 'h')))
    (EAp (EAp (EConst (CFun ":")) (EConst (CChar 'i')))
         (EConst (CFun "[]")))
```

sind z. B. äquivalent. Ein eigener Konstruktor für String-Literale wurde aus Effizienzgründen eingeführt.

### 5.1.5 Diskussion alternativer Ansätze

Im nachhinein stellte sich heraus, daß man die gewählten Datenstrukturen an zwei Stellen verbessern könnte.

#### 5.1.5.1 Lambda-Abstraktionen

Die abstrakte Syntax von Lambda-Abstraktionen lautet

```
data Expression = ...
                | ELam [Expression] Expression
                | ...
```

Der Konstruktor `ELam` ist für beliebig viele formale Parameter ausgelegt. Der Ausdruck $(\lambda x\ y \rightarrow f\ x\ y)$ hätte die abstrakte Repräsentation

```
ELam [EVar "x", EVar "y"]
     (EAp (EAp (EVar "f") (EVar "x")) EVar "y")
```

bekommen sollen. Wenn nun aber auf den Ausdruck $(\lambda x\ y \rightarrow f\ x\ y)x$ eine Eta-Reduktion (Abb. 5.3) angewandt werden soll, schlägt das Pattern Matching gegen das Eingabeschema

```
ELam [EPara "x"] (EAp (EPara "e") (EPara "x"))
```

fehl, da hier die ein-elementige Liste `[EPara "x"]` nicht auf die zwei-elementige Liste `[EVar "x", EVar "y"]` paßt.

Das Problem ließ sich beseitigen durch eine Änderung im Parser, so daß Lambda-Abstraktionen mit mehreren formalen Parametern nun in geschachtelte Lambda-Ausdrücke mit jeweils einem Parameter übersetzt werden. Die Repräsentation des Ausdrucks $(\lambda x\ y \rightarrow f\ x\ y)$ in abstrakter Syntax ist nun also

Abbildung 5.3: *Definition der Regel Eta Reduction*

```
ELam [EVar "x"]
     (ELam [EVar "y"]
           (EAp (EAp (EVar "f") (EVar "x")) (EVar "y")))
```

was der konkreten Syntax $(\lambda x \rightarrow (\lambda y \rightarrow f\ x\ y))$ entspricht.

In dieser Form kann der Ausdruck mit zwei Eta-Reduktionen zu $f$ vereinfacht werden.

Da in der abstrakten Syntax von Lambda-Abstraktionen nun aber nur noch ein-elementige Listen verwendet werden, könnte man den Datentyp `Expression` nun nachträglich zu

```
data Expression = ...
                | ELam Expression Expression
                | ...
```

vereinfachen.

#### 5.1.5.2 Funktionsdefinitionen

Die abstrakte Syntax einer Funktionsdefinition lautet

```
type SCDefn = (Name, [Expression], Expression)
```

Die Repräsentation von `f x y = x+y` ist also

```
("f",
 [EVar "x", EVar "y"],
 (EAp (EAp (EConst (CFun "+")) (EVar "x")) (EVar "y"))
)
```

Ein *Unfold* von `f` im Ausdruck `f 3` soll das Ergebnis `λy -> 3+y` liefern. Dazu wird zuerst eine äquivalente Funktionsdefinition `f = λx y -> x+y` konstruiert, die keine Argumente mehr enthalt. Die rechte Seite dieser Definition wird in `f 3` eingesetzt. Anschließend wird das Ergebnis mit Beta-Reduktionen (5.5.5) so weit wie möglich vereinfacht.

Ein alternativer Ansatz wäre demnach gewesen, Funktionsdefinitionen gleich beim Parsen auf die argumentlose Repräsentation zu bringen. Der Datentyp `SCDefn` könnte dann zu

```
type SCDefn = (Name, Expression)
```

vereinfacht werden.

## 5.2 Der Parser

Das Parsen eines Gofer-Quelltextes läßt sich in fünf Phasen unterteilen:

1. Alle Kommentare werden entfernt.
2. Das Programm wird in die einzelnen Top-Level-Definitionen zerlegt. Top-Level-Definitionen sind entweder Funktionssignaturen, Typdeklarationen oder Funktionsgleichungen.
3. Da wir immer davon ausgehen, daß der Benutzer das Programm bereits vom Gofer-Interpreter auf Typkorrektheit und Deklariertheit von Datentypen überprüft hat, sind für uns nur die Funktionsgleichungen von Belang, weswegen im nächsten Schritt alle Signaturen und Typdeklarationen entfernt werden.
4. Jede Funktionsgleichung wird nun einzeln geparst. Dieser Vorgang unterteilt sich seinerseits in drei Schritte.
   - Klammern und Strichpunkte werden abhängig vom Layout in Let- und Case-Anweisungen eingefügt.
   - Es folgt die lexikalische Analyse, in der zusammengehörende Gruppen von Zeichen (Zahlen, Strings, Schlüsselwörter) zu Tokens zusammengefaßt werden.
   - Zum Abschluß erfolgt die syntaktische Analyse, die eine Liste von Tokens konsumiert und eine Funktionsgleichung in abstrakter Syntax zurückliefert.
5. Verlief das Parsen von allen Gleichungen erfolgreich, so wird als letzter Schritt der Pattern Matching Compiler auf das ganze Programm angewandt.

Der entsprechende Quellcode lautet:

```
parsePrg :: String -> Maybe Program
parsePrg prg = let fs = map parseSCDefn $ preprocess prg
               in if null fs then Fail
                  else if Fail `elem` fs
                  then Fail
                  else Ok (patternMatchCompile $ map selectOk fs)

preprocess :: String -> [String]
preprocess = map offside . filter codeLine . decomposePrg . stripComments

parseSCDefn :: String -> Maybe SCDefn
parseSCDefn f = let es = just function $ scanner f
                in if null es then Fail else Ok (snd (head es))

selectOk :: Maybe a -> a
patternMatchCompile :: Program -> Program
```

Im folgenden wird auf die einzelnen Abschnitte des Parsens genauer eingegangen.

### 5.2.1 Entfernen von Kommentaren

Die Funktion `stripComments :: String -> String` entfernt alle Kommentare - sowohl einzeilige, die mit „- -“ eingeleitet werden, als auch Blöcke, die mit „{-“ beginnen und „-}“ enden. Bei Blöcken muß die in Gofer mögliche Schachtelung von Kommentaren berücksichtigt werden. Wenn innerhalb eines Kommentars ein weiterer Kommentar geöffnet wird, müssen also auch zwei Kommentare geschlossen werden. Eine weitere Fehlerquelle bei der Implementierung sind Strings und Characterliterale. Diese können nämlich Kommentarsymbole enthalten, die einen naiv implementierten Algorithmus aus dem Takt bringen würden. Dieses Problem wird noch verschärft durch mit einem Backslash geschützte Quotes (') und Doublequotes (") innerhalb von Strings und Characterliteralen. Die hier vorgestellte Funktion arbeitet jedoch auch in diesen Fällen korrekt.

```
stripComments :: String -> String
stripComments = stripComments' 0
```

Als weiteres Argument wurde hier ein Zähler für die Schachtelungstiefe der Kommentare eingeführt.

```
stripComments' :: Int -> String -> String
stripComments' n ('-':'-':cs)     = stripComments' n (dropWhile (/='\n') cs)
stripComments' n ('{':'-':cs)     = stripComments' (n+1) cs
stripComments' n ('-':'}':cs)     = stripComments' (max 0 (n-1)) cs
stripComments' 0 ('\"':cs)        = '\"' : skipString 0 cs
stripComments' n ('\"':cs)        = skipString n cs
stripComments' 0 ('\'':'\\':c:'\'':cs)='\'':'\\':c:'\'':stripComments' 0 cs
stripComments' n ('\'':'\\':c:'\'':cs)=stripComments' n cs
stripComments' 0 ('\'':c:'\'':cs) = '\'' : c : '\'' : stripComments' 0 cs
stripComments' n ('\'':c:'\'':cs) = stripComments' n cs
stripComments' 0 (c:cs)           = c : stripComments' 0 cs
stripComments' n (c:cs)           = stripComments' n cs
stripComments' n ""               = ""
```

Die Funktion `skipString` überliest einen String komplett. Wie oben erwähnt, muß dabei darauf geachtet werden, daß durch Backslash geschützte Doublequotes nicht fälschlicherweise als Ende des Strings erkannt werden.

```
skipString :: Int -> String -> String
skipString 0 ('\\':c:cs)  = '\\' : c : skipString 0 cs
skipString n ('\\':c:cs)  = skipString n cs
skipString 0 ('\"':cs)    = '\"' : stripComments' 0 cs
skipString n ('\"':cs)    = stripComments' n cs
skipString 0 (c:cs)       = c : skipString 0 cs
skipString n (c:cs)       = skipString n cs
skipString n ""           = ""
```

### 5.2.2 Zerlegen in die einzelnen Top-Level-Definitionen

Die Funktion `decomposePrg :: String -> [String]` nimmt den Programmtext und liefert eine Liste zurück, in der jedes Element entweder eine Typdeklaration, eine Funktionssignatur oder eine Funktionsgleichung ist. Zum Beispiel wird aus der Definition

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

die 4-elementige Liste

```
["fib :: Int -> Int",
 "fib 0 = 0",
 "fib 1 = 1",
 "fib n = fib (n-1) + fib (n-2)"]
```

### 5.2.3 Entfernen der Signaturen und Typ-Deklarationen

Da Typ-Informationen in der gegenwärtigen Version von ULTRA nicht verwendet werden, werden diese mit `filter codeLine :: [String] -> [String]` gelöscht. Aus dem obigen Beispiel wird nun

```
["fib 0 = 0",
 "fib 1 = 1",
 "fib n = fib (n-1) + fib (n-2)"]
```

### 5.2.4 Einfügen von Strichpunkten und Klammern

In dieser Phase werden abhängig vom Layout Strichpunkte und Klammern in Let- und Case-Anweisungen eingefügt, um später das Parsen zu erleichtern. Aus

```
case ys of
        []      -> 1
                   + 2
        (x:xs) -> A x xs
```

wird beispielsweise

```
case ys of { [] -> 1 + 2; (x:xs) -> A x xs }
```

Zuerst werden alle Zeilen vom Typ `Line` nach `IndentLine` umgewandelt. `IndentLine` ist ein Tupel aus einer Zahl und einem String. Die Zahl gibt dabei die Einrückung der Zeile an, der String den eigentlichen Inhalt der Zeile (ohne führende Leerzeichen).

```
type Line       = String
type IndentLine = (Int, String)

indent :: Line -> IndentLine
indent = indent' 0
         where indent' :: Int -> Line -> IndentLine
               indent' n (' ' :xs) = indent' (n+1) xs
               indent' n ('\t':xs) = indent' ((n `div` 8)*8+8) xs
               indent' n (c:cs)    = (n,c:cs)
               indent' n ""        = (n,"")
```

Aus dem oben aufgeführten Beispiel würde dabei

```
[( 0, "case ys of"),
 ( 8, "[]     -> 1"),
 (18, "+ 2"),
 ( 8, "(x:xs) -> A x xs")]
```

Auf den so konvertierten Programmtext wird nun die Funktion `insertSeps :: [IndentLine] -> String` angewandt, die nach folgendem Algorithmus verfährt:

Wird ein `let` oder `of` erkannt, so wird danach eine öffnende Klammer eingefügt, die Einrückung $i$ des nächsten Zeichens wird auf einen Stack gelegt und für die nachfolgenden Zeilen wird wie folgt verfahren:

- Ist die Einrückung der Zeile genau $i$, so beginnt eine neue Alternative (in einer Case-Anweisung) bzw. eine neue Gleichung (in einer Let-Anweisung), also wird *vor* dieser Zeile ein Semikolon eingefügt.
- Ist die Einrückung größer als $i$, so wird die Zeile einfach übernommen und mit der nächsten Zeile fortgefahren.
- Ist die Einrückung kleiner als $i$, so gehört die Zeile bereits zur nächsten Anweisung, also wird *vor* dieser Zeile eine schließende Klammer eingefügt und die Zahl $i$ wird vom Stack „gepopt“.

Zu beachten ist noch, daß der Algorithmus bei geschachtelten Let/Case-Anweisungen rekursiv gestartet werden muß. Aus diesem Grund ist der Stack für die verschiedenen Einrückungen notwendig.

Dieses Verfahren entspricht noch nicht ganz der in Gofer verwendeten Layout-Regel. Ein Problem gibt es noch dann, wenn eine Case-Anweisung mit runden Klammern eingeklammert ist und die schließende Klammer dabei auf der Zeile der letzten Alternative steht. Dann müßte nämlich nach der Case-Anweisung die schließende geschweifte Klammer vor der runden Klammer eingefügt werden.

Z. B. wird die Funktionsdefinition

```
test xs = f (case xs of []     -> 0
                        (x:xs) -> 1)
```

in der aktuellen Implementierung noch nicht richtig behandelt. Das Problem kann der Benutzer vorerst umgehen, indem er die schließende Klammer auf die nächste Zeile setzt, mit einer kleineren Einrückung als die letzte Alternative.

```
test xs = f (case xs of []     -> 0
                        (x:xs) -> 1
            )
```

### 5.2.5 Lexikalische Analyse

In der lexikalischen Analyse wird der vorbearbeitete Quelltext in eine Folge von Tokens umgewandelt. Der Datentyp `Token` ist dabei folgendermaßen definiert:

```
data Token = Ident String       -- Bezeichner
           | Symbol String      -- Symbole, Operatoren, Schluesselwoerter
           | Number Int         -- natuerliche Zahlen
           | String String      -- Stringliterale
           | Char Char          -- Characterliterale
           | Constructor String -- Typkonstruktor
           | Error              -- Fehlertoken
```

Die Umwandlung des Quelltexts in die Folge von Tokens wird von der Funktion `scanner` erledigt.

```
scanner :: String -> [Token]
scanner ('T':'r':'u':'e':xs)          = Symbol "True"   : scanner xs
scanner ('F':'a':'l':'s':'e':xs)      = Symbol "False"  : scanner xs
scanner ('i':'f':' ':xs)              = Symbol "if"     : scanner xs
...                                   = ...             : scanner xs
...                                   = ...             : scanner xs
scanner ('\'':c:'\'':xs)              = Char c          : scanner xs
scanner ('\"':xs)                     = scanString "" xs
scanner (' ':xs)                      = scanner xs
scanner ('\n':xs)                     = scanner xs
scanner ('\t':xs)                     = scanner xs
scanner (c:cs) = if (isDigit c) then scanNumber (ord c-48) cs
                 else if (isLower c) then scanIdent [c] cs
                 else if (isUpper c) then scanConstructor [c] cs
                 else scanOperator [c] cs
scanner ""     = []

scanNumber :: Int -> String -> [Token]
scanIdent :: String -> String -> [Token]
scanOperator :: String -> String -> [Token]
scanString :: String -> String -> [Token]
scanConstructor :: String -> String -> [Token]
```

### 5.2.6 Syntaktische Analyse

In einer funktionalen Sprache wie Gofer ist es möglich, komplexe Parser aus elementaren Parsern mit sogenannten Parserkombinatoren zusammenzubauen. Parserkombinatoren sind höhere Funktionen, die einen oder mehrere Parser als Argumente erwarten und einen neuen Parser zurückliefern. Für eine ausführliche Beschreibung dieser Technik verweise ich den Leser auf [Fok95] und [PJL92].

Mit Hilfe der Parserkombinatoren kann relativ leicht ein Parser für Gofer geschrieben werden. Die Grammatik der Sprache kann nahezu direkt in ein Programm übersetzt werden. Die Hauptfunktion `function`, welche eine Funktionsgleichung parst, hat beispielsweise die Form

```
function :: Parser Token SCDefn
function = name <*> many atomic <*> sym "=" *> expr
                <@ \(name,(args,body)) -> (name,args,body)
```

`name`, `atomic` und `expr` sind hier die entsprechenden Funktionen zum Parsen von Variablen, atomaren Ausdrücken und beliebigen Ausdrücken. `<*>` und `*>` sind Parserkombinatoren zur Hintereinanderschaltung von Parsern, `<@` ist ein Parsermodifikator, der auf das Ergebnis eines Parsers eine Funktion anwendet.

Da Parserkombinatoren in der Literatur bereits ausreichend behandelt worden sind, möchte ich im folgenden lediglich auf zwei Probleme eingehen, die es bei der Implementierung zu beachten galt.

#### 5.2.6.1 Linksrekursionen

Wenn wir die Grammatikregel für Funktionsanwendungen

```
app_expr ::= atomic | app_expr atomic
```

direkt in die Funktion

```
app_expr :: Parser Token Expression
app_expr =  atomic
        <|> app_expr <*> atomic  <@ (\(e1,e2) -> EAp e1 e2)
```

übersetzen würden, so würde der Parser nicht terminieren. Das Problem läßt sich jedoch leicht lösen, indem die Linksrekursion zu einer Wiederholung umgeschrieben wird.

```
app_expr :: Parser Token Expression
app_expr = many1 atomic <@ foldl1 EAp
```

#### 5.2.6.2 Effizienz des Parsers

Die Parserkombinatoren verleiten leicht zu naiven Implementierungen, welche dann exponentielle Laufzeit haben. Angenommen, wir würden folgende Grammatik direkt in ein Programm übersetzen:

```
expr1 ::= expr2 '||' expr1
        | expr2

expr2 ::= expr3 '&&' expr2
        | expr3

expr3 ::= expr4 '<'  expr4
        | expr4 '>'  expr4
        | expr4 '==' expr4
        | expr4 '>=' expr4
        | expr4 '<=' expr4
        | expr4 '/=' expr4
        | expr4

expr4 ::= expr5 '+' expr4
        | expr5 '-' expr5
        | expr5

expr5 ::= number '*' expr5
        | number
```

Der von `expr1` zu parsende Ausdruck sei „1 /= 2".
Zuerst versucht der Parser, eine `expr2` zu erkennen. Anschließend wird aber kein Oder-Symbol (`||`) gefunden, also wird die zweite Alternative probiert und nochmals eine `expr2` geparst. Jeder der beiden Aufrufe von `expr2` ruft zweimal `expr3` auf, da jeweils nach dem ersten Versuch kein nachfolgendes Und-Symbol (&&) gefunden wird. Die vier Aufrufe von `expr3` sind jeweils erst im sechsten Versuch erfolgreich. `expr4` wird also schon 24 mal aufgerufen, `expr5` 72 mal und `number` 144 mal. Jedesmal wird die 1 geparst, bis endlich

das Ungleichheitszeichen gelesen werden kann. Anschließend wird noch sechsmal die 2 geparst, bis der Ausdruck „1 /= 2“ endlich erkannt ist.

Das Problem läßt sich durch *Links-Faktorisierung* [ASU86] lösen: Wenn mehrere Alternativen einer Grammatikregel das gleiche Präfix haben

```
A ::= a b1 | a b2 | a b3 | a b4
```

so wird diese Regel durch das folgendes Paar von Regeln ersetzt:

```
A  ::= a A'
A' ::= b1 | b2 | b3 | b4
```

wobei `A'` ein neu eingeführtes Nichtterminal ist. Im obigen Beispiel erhalten wir durch Links-Faktorisierung die Grammatik

```
expr1  ::= expr2 expr1'
expr1' ::= '||' expr1
         | epsilon

expr2  ::= expr3 expr2'
expr2' ::= '&&' expr2
         | epsilon

expr3  ::= expr4 expr3'
expr3' ::= '<'  expr4
         | '>'  expr4
         | '==' expr4
         | '>=' expr4
         | '<=' expr4
         | '/=' expr4
         | epsilon

expr4  ::= expr5 expr4'
expr4' ::= '+' expr4
         | '-' expr5
         | epsilon

expr5  ::= number expr5'
expr5' ::= '*' expr5
         | epsilon
```

Diese Grammatik ist nun direkt in einen effizienten Parser übersetzbar.

### 5.2.7 Behandlung von Funktionsdefinitionen mit Pattern Matching

Die Programmiersprache Gofer erlaubt Funktionsdefinitionen mit Pattern Matching. Damit können Funktionen oft übersichtlicher aufgeschrieben werden. Die Fakultätsfunktion würde beispielsweise statt

```
fac n = if n==0 then 1 else n * fac (n-1)
```

so lauten:

```
fac 0 = 1
fac n = n * fac (n-1)
```

Generell gibt es zwei Möglichkeiten, wie man mit einer solchen Funktionsdefinition in einem Transformationssystem umgeht.

Eine Möglichkeit ist, jede funktionale Gleichung einzeln zu behandeln. Man hat dann wenig Aufwand beim Parsen, handelt sich die Probleme aber später ein, wenn ein Funktionsaufruf entfaltet werden soll. Wie müßte das Ergebnis lauten, wenn `fac` in `fac (g x)` entfaltet werden soll? `1`? Oder `(g x) * fac ((g x)-1)`? Wenn nicht abgeleitet werden kann, ob `g x` zu 0 evaluiert oder nicht, muß beim *Unfold* eine Fallunterscheidung konstruiert werden.

Die andere Möglichkeit ist, gleich beim Parsen aus den einzelnen Funktionsgleichungen die entsprechende Fallunterscheidung zu konstruieren. Aus der obigen zweiten Definition von `fac` würde dann

```
fac u1 = case u1 of 0 -> 1
                    n -> n * fac (n-1)
```

Ein *Unfold* von `fac` ist damit unproblematisch. Nach dem *Unfold* kann der Simplifier versuchen, die Fallunterscheidung wieder zu eliminieren, wodurch man die gleichen Ergebnisse wie im ersten Ansatz erzielt.

Für das ULTRA-System wurde der zweite Ansatz gewählt. Der Algorithmus, nach dem die Fallunterscheidungen konstruiert werden, ist angelehnt an das im Kapitel 5 von [PJ87] vorgestellte Verfahren. Zu beachten ist beispielsweise, daß nicht in jedem Fall Fallunterscheidungen konstruiert werden dürfen. Sei z. B. die Funktion `map` folgendermaßen definiert:

```
map f [] = []
map g xs = g x : map g xs
```

Eine falsche Übersetzung wäre

```
map u1 u2 = case u1 of
                 f -> case u2 of
                           [] -> []
                 g -> case u2 of
                           xs -> g x : map g xs
```

da hier der zweite Fall nie erreicht werden kann.

Wenn die Liste der Argumente in allen Gleichungen mit einer Variable beginnt, kann in jeder Gleichung die jeweilige Variable durch $u1$ ersetzt werden, und die Fallunterscheidung kann weggelassen werden. Die Übersetzung, die ULTRA liefert, lautet:

```
map u1 u2 = case u2 of
                 [] -> []
                 xs -> u1 x : map u1 xs
```

Generell muß zwischen den folgenden drei Fällen unterschieden werden:

- Alle Gleichungen haben eine Variable im ersten Argument.
- Alle Gleichungen haben Konstruktoren im ersten Argument.
- Manche Gleichungen haben Variablen, manche haben Konstruktoren im ersten Argument.

Es folgt der Algorithmus, wie er im ULTRA-System implementiert wurde.

#### 5.2.7.1 Der Pattern Matching Compiler

Zuerst werden Gleichungen, die zur selben Funktionsdefinition gehören, zu Gruppen zusammengefaßt. Auf jede Gruppe wird dann die Funktion `mkCase` angewandt.

```
patternMatchCompile :: Program -> Program
patternMatchCompile prg = map mkCase (group prg)

group :: [SCDefn] -> [[SCDefn]]
group scs = partition fst3 scs
```

Für Funktionsdefinitionen mit nur einer Gleichung ist nichts zu tun. Bei mehreren Gleichungen wird eine neue Gleichung konstruiert, deren rechte Seite durch die Funktion `pmatch` berechnet wird.

```
mkCase :: [SCDefn] -> SCDefn
mkCase []  = error "mkCase: empty list"
mkCase [f] = f
mkCase fs  = (name, map EVar vars, pmatch vars eqs EUndef)
             where name = fst3 (head fs)
                   n = arity (head fs)
                   vars = take n (difference symbols forbidden)
                   forbidden = foldl union [] (map freevars fs)
                   eqs = map g fs where g (name,args,body) = (args,body)
```

Die Funktion `pmatch` ist der eigentliche Kern des Algorithmus. Sie bekommt drei Argumente: eine Liste von Variablen $[u1, u2, ...]$, eine Liste von Funktionsgleichungen und eine (mit `EUndef` initialisierte) *Default Expression*, die für den Fall, daß die Gleichungen nicht alle Fälle abdecken, eingesetzt wird.

`pmatch` konstruiert dann die rechte Seite der neuen Funktionsdefinition, die die alte Definition über Pattern Matching ersetzen soll. Dazu werden die Gleichungen zunächst in Gruppen aufgeteilt, so daß jede Gruppe entweder nur Variablen oder nur Konstruktoren im ersten Argument hat. Für jede Gruppe wird dann `matchVarCon` aufgerufen, wobei mit der letzten Gruppe angefangen wird. Der von `matchVarCon` zurückgelieferte Ausdruck wird jeweils als *Default Expression* für die Gruppe davor verwendet.

```
pmatch :: [Name] -> [Equation] -> Expression -> Expression
pmatch []     qs def = let es = [ e | ([],e) <- qs ]
                       in if null es then def else head es
pmatch (u:us) qs def = foldr (matchVarCon (u:us)) def (partition isVar qs)
```

`matchVar` wird aufgerufen, wenn der oben beschriebene Fall vorliegt, in dem die Liste der Patterns in jeder Funktionsgleichung mit einer Variable beginnt.

```
matchVar :: [Name] -> [Equation] -> Expression -> Expression
matchVar (u:us) qs def = pmatch us [(ps,subst e v u)
                                   | (EVar v : ps, e) <- qs] def
```

`matchCon` behandelt den Fall, in dem die Liste der Patterns in jeder Funktionsgleichung mit einem Konstruktor beginnt. Da hier die Gleichungen möglicherweise nicht alle Fälle abdecken, muß eventuell für die nicht abgedeckten Fälle die *Default Expression* eingesetzt werden.

```
matchCon :: [Name] -> [Equation] -> Expression -> Expression
matchCon (u:us) qs EUndef
 = ECase (EVar u) (map f (partition (head.fst) qs))
   where f qs        = (head (fst (head qs)), pmatch us (map g qs) EUndef)
         g ((p:ps),e) = (ps,e)
matchCon (u:us) qs def
 = ECase (EVar u) (map f (partition (head.fst) qs) ++ [(EWild, def)])
   where f qs        = (head (fst (head qs)), pmatch us (map g qs) def)
         g ((p:ps),e) = (ps,e)
```

`matchVarCon` wählt zwischen den beiden Fällen `matchVar` und `matchCon` aus.

```
matchVarCon :: [Name] -> [Equation] -> Expression -> Expression
matchVarCon us qs def = matchVar us qs def , if isVar (head qs)
                      = matchCon us qs def , otherwise
```

`arity` liefert die Stelligkeit einer Funktion.

```
arity :: SCDefn -> Int
arity (name,args,body) = length args
```

`isVar` testet, ob das erste Argument einer Funktionsgleichung eine Variable ist.

```
isVar :: Equation -> Bool
isVar (EVar v : _, _) = True
isVar _               = False
```

`subst` ersetzt in einem Ausdruck `e` alle Vorkommen der Variable `u` durch die Variable `v`.

```
subst :: Expression -> Name -> Name -> Expression
subst e u v = mapVar (\x -> if x==u then EVar v else EVar x) e
```

## 5.3 Der Pretty-Printer

Der Pretty-Printer benutzt die *Pretty-Printer Library*, die Teil der *TkGofer Standard Distribution* [VSS96] ist.

In dieser Bibliothek wird eine Klasse `Pretty` und ein neues Widget `PrettyEdit a` zur Verfügung gestellt. `PrettyEdit a` ist ein Fenster zur Darstellung von strukturierten Dokumenten vom Typ `a`, wobei `a` eine Instanz der Klasse `Pretty` sein muß. Das Transformationsfenster von ULTRA ist also vom Typ `PrettyEdit Program` und `Program` ist eine Instanz der Klasse `Pretty`.

Wenn nun mit der TkGofer-Funktion `setValue` dem Fenster ein Wert zugewiesen wird, so wird dieser Wert mit der Funktion `pretty` (einer Methode der Klasse `Pretty`) in das Fenster hineingeschrieben. Alles, was zu tun ist, ist also, die Funktion `pretty` für die Datentypen `Program`, `SCDefn`, `Expression` und `Alter` zu definieren.

### 5.3.1 Definition der Funktion *pretty*

Für den Typ `Program` lautet die Definition

```
instance Pretty Program where
   pretty = foldl1 (\x y -> x $$ pretty "" $$ y) .
            map (\(i,x) -> fork i x) .
            zip [1..]
```

Die einzelnen Funktionsdefinitionen des Programms werden zuerst numeriert (`zip [1..]`). Dann werden ihre Text-Darstellungen von der Funktion `fork` berechnet. `fork i` tut das selbe wie die Funktion `pretty`, nur fügt sie zusätzlich vor Beginn und nach Ende des Textes ein *Begin Tag* bzw. ein *End Tag* mit der Nummer `i` hinzu. Die Tags sind unsichtbare Zeichen und dienen dazu, die Struktur des Programms zu beschreiben. Sie werden später benötigt (siehe Abschnitte 5.3.2 und 5.3.3).

Schließlich werden die Text-Darstellungen der einzelnen Funktionen zum Gesamtprogramm zusammengefügt. Dazu wird der Layout-Kombinator `$$` verwendet, der zwei Texte übereinander anordnet. Jeweils zwischen zwei Funktionsdefinitionen wird dabei eine Leerzeile gelassen.

Für `Expressions` lautet die Definition der Funktion `pretty` auszugsweise

```
instance Tree Expr Name => Pretty Expression where
   pretty (EVar x)    = italic x
   pretty (EAp e1 e2) = fork 1 e1 <> pretty " " <> pfork 2 e2
   pretty (ECase c [(EConst (CBool True),e1),
                    (EConst (CBool False),e2)])
        = (keyword "if "   <> fork 1 c) $$
          (keyword "then " <> fork 2 e1) $$
          (keyword "else " <> fork 3 e2)
   ...
   ...
   pretty (ELam args e)
        =  pretty "(\\" <> pp_spList args
                        <> pretty " -> "
                        <> fork 1 e <> pretty ")"
   ...
```

`<>` ist dabei ein weiterer Layout-Kombinator, der Texte nebeneinander anordnet. Die Funktion `pp_splist` ordnet eine Liste von Texten nebeneinander an und fügt dazwischen jeweils ein Leerzeichen ein. `pfork` ruft `fork` auf und klammert den zurückgelieferten Text in runden Klammern ein. `italic` ist mit `pretty` identisch, verwendet aber kursiven Zeichensatz.

### 5.3.2 Markieren eines Ausdrucks

Wenn mit der linken Maustaste auf das Transformationsfenster geklickt wird, so wird mit Hilfe der Tags der kleinste Ausdruck bestimmt, innerhalb welchem der Mausklick auftrat. Dieser Ausdruck wird dann farblich markiert. Da die Tags eine Klammerstruktur bilden (auf jedes *Begin Tag* folgt ein *End Tag* mit der gleichen Nummer), ist der zu markierende Ausdruck immer eindeutig bestimmt.

Wenn in `if n == 1 then 2 else 3` z. B. auf das `then` geklickt wird, so wird die komplette If-Anweisung markiert. Wird dagegen auf den Zwischenraum zwischen `==` und `1` geklickt, so wird der Ausdruck `n == 1` markiert. Wird genau auf die `1` geklickt, so wird nur diese markiert.

### 5.3.3 Berechnung der Programmposition des markierten Ausdrucks

Wenn ein Transformationsschritt ausgeführt werden soll, stellt sich das Problem, im abstrakten Syntaxbaum des Programms den Knoten zu finden, der dem im Transformationsfenster markierten Ausdruck entspricht. Dazu werden wiederum die Tags verwendet.

Die Funktion `getMarkedPart :: PrettyEdit a -> GUI [Int]` liefert eine Liste von Zahlen, die angibt, wie der markierte Textbereich mit Hilfe der Tags gefunden werden kann.

Die dahintersteckende Idee läßt sich am einfachsten mit einer Analogie zu Textdokumenten beschreiben: Um den Abschnitt, den Sie hier gerade lesen, innerhalb dieser Arbeit zu finden, genügt eine Liste von drei Zahlen: 5.3.3. Die erste Zahl gibt das Kapitel an, die zweite den Abschnitt und die dritte den Unterabschnitt. Die Funktion `getMarkedPart` bestimmt für ein Dokument zu einem markierten Textbereich genau diese Liste.

Beim Ausgeben des Programms mit dem Pretty-Printer (siehe 5.3.1) wurden die Tags getreu der abstrakten Syntax des Programms vergeben. Eine If-Anweisung hat z. B. genau drei Unterabschnitte: Bedingung, Then-Zweig und Else-Zweig. Dadurch ist die von `getMarkedPart` gelieferte Liste mit der Dewey-Notation unseres gesuchten Knotens im Syntaxbaum identisch.

Mit Hilfe der Dewey-Notation ist es nun einfach, den Knoten zu bestimmen und die Transformation an genau dieser Stelle anzuwenden.

## 5.4 Transformationssystem

### 5.4.1 Globaler Zustand

Der Zustand des Transformationssystems wird durch ein Tripel `Globals` beschrieben. Die Komponenten dieses Tripels sind:

- das aktuelle Programm, das transformiert werden soll.
- alle ehemaligen Versionen der Funktionen des aktuellen Programms.
  Diese müssen mitgeführt werden, weil sie als potentielle Kandidaten für *Fold*-Aktionen in Frage kommen.
- die History-Liste, die zum Navigieren (Back/Forward) gebraucht wird.

```
type Globals = Var (Program     -- Current Program
                   ,Program     -- Fold Expressions
                   ,History     -- Past && Future Transformation Steps
                   )
```

### 5.4.2 Unfold

Um einen Funktionsaufruf zu entfalten, kann der Benutzer wahlweise nur den Funktionsnamen oder den kompletten Aufruf inklusive Argumenten markieren. Um `f` in `f 1 2` zu entfalten, ist es also egal, ob er `f`, `f 1` oder `f 1 2` markiert.

Wird der `Unfold` Button gedrückt, so wird als erstes die Funktion `leftmost` auf den markierten Ausdruck angewandt, um die zu entfaltende Funktionsvariable zu bestimmen. Diese steht im abstrakten Syntaxbaum an der linksuntersten Stelle. (Siehe Abb. 5.4)

Abbildung 5.4: *Syntaxbaum des Ausdrucks f 1 2*

```
leftmost :: Expression -> Expression
leftmost (EAp e1 e2) = leftmost e1
leftmost e = e
```

Dann wird mit der Funktion `decl` nach der Funktionsdefinition der zu entfaltenden Variablen gesucht. Endet diese Suche erfolglos, kann das *Unfold* natürlich nicht ausgeführt werden.

```
decl :: Name -> Program -> Maybe SCDefn
decl f []     = Fail
decl f (g:gs) = if f==fst3 g then Ok g else decl f gs
```

Insgesamt wird an drei Stellen nach einer Funktionsdefinition gesucht: im aktuellen Programm, in der Funktionsdatenbank (in der die Definitionen der Standardfunktionen wie `sum` oder `length` stehen) und in der lokalen Umgebung, falls die zu entfaltende Variable innerhalb einer Let-Anweisung steht.

Wird die Definition gefunden, so wird die Funktion `unfold` ausgeführt, welche die Funktionsvariable durch den Rumpf der Definition bzw. durch eine aus den Funktionsparametern und dem Rumpf konstruierte Lambda-Abstraktion ersetzt.

```
unfold :: SCDefn -> Expression -> Expression
unfold (f_name, f_args, f_body) (EVar f)
          = if f==f_name
            then mkLambda f_args f_body
            else EVar f
unfold _ e = e

mkLambda :: [Expression] -> Expression -> Expression
mkLambda [] e     = e
mkLambda (x:xs) e = ELam [x] (mkLambda xs e)
```

*Beispiel:* Sei `f a b = a+b`. `f` soll in `f 1 2` entfaltet werden. Dann ersetzt die Funktion `unfold` das `f` durch die Lambda-Abstraktion `(λa b -> a+b)`. Das vorläufige Resultat ist also `(λa b -> a+b) 1 2`.

Zum Schluß wird versucht, das Ergebnis durch Beta-Reduktion zu vereinfachen (siehe Abschnitt 5.5.5). Das Endergebnis ist hier also `1+2`.

Da sich nach einem *Unfold* meistens weitere Vereinfachungsmöglichkeiten ergeben (hier z. B. kann `1+2` noch zu `3` vereinfacht werden), die sich oft auch auf den umliegenden Kontext auswirken, wird nun noch der Simplifier auf die komplette rechte Seite der Funktion angewandt, innerhalb der sich der entfaltete Ausdruck befindet.

### 5.4.3 Fold

Nach dem Drücken des *Fold Buttons* wird zuerst nach möglichen Funktionen gesucht, auf deren Aufruf der markierte Ausdruck gefaltet werden kann. Potentielle Kandidaten sind wie beim *Unfold* das aktuelle Programm, die Programmdatenbank und die lokale Umgebung (innerhalb von Let-Anweisungen) sowie alle ehemaligen Versionen der Funktionen des aktuellen Programms, welche im globalen Zustand mitgeführt werden. Ist der markierte Ausdruck die komplette rechte Seite einer Funktion, so wird diese Funktion nicht als Kandidat betrachtet, da ein *Fold* ansonsten garantiert zur Nichtterminierung führen würde.

Die Überprüfung, ob ein Ausdruck $e$ auf einen Aufruf der Funktion $f$ gefaltet werden kann, wird von der Funktion `foldPossible` durchgeführt.

```
foldPossible :: Expression -> SCDefn -> Bool
foldPossible e (f_name,f_args,f_body)
      = let pat  = mkPattern f_name f_args
            args = parameters pat
            scheme = mkScheme args f_body
            sub = match scheme e
        in case sub of Fail   -> False
                       (Ok s) -> True
```

Ist ein *Fold* für mehr als eine Funktion möglich, so wird der Benutzer aufgefordert, zu wählen, auf welchen Funktionsaufruf gefaltet werden soll. (Siehe Abb. 3.3)

Schließlich wird die eigentliche *Fold*-Aktion von der Funktion `fold` ausgeführt.

```
fold :: SCDefn -> Expression -> Expression
fold (f_name,f_args,f_body) e
      = let pat  = mkPattern f_name f_args
            args = parameters pat
            scheme = mkScheme args f_body
            sub = match scheme e
        in case sub of Fail   -> e
                       (Ok s) -> applySub' s pat
```

### 5.4.4 Benutzerdefinierte Regeln

Nach Programmstart werden die Regeln und Strategien, die der Benutzer in den Regelkatalogen definiert hat (siehe Kapitel 4), eingelesen, geparst und in der Regelauswahlbox (Abschnitt 3.8) abgelegt.

Die Regeln werden beim Parsen in die *schematische Form* gebracht. Dies ist ein 4-Tupel mit den folgenden Komponenten: Name der Regel, Liste von Anwendbarkeitsbedingungen, Eingabeschema, Ausgabeschema

```
type TransRule = (Name,[Term], Expression, Expression)
```

Die schematische Form, in die die Regel

```
mapPromotion f =
  map f . concat =:= concat . map (map f)
```

gebracht wird, ist z. B.

```
( "mapPromotion"
, []
, EAp (EAp (EConst (CFun "."))
      (EAp (EVar "map") (EPara "f")))
      (EVar "concat")
, EAp (EAp (EConst (CFun "."))
      (EVar "concat"))
      (EAp (EVar "map") (EAp (EVar "map") (EPara "f")))
)
```

Wird eine Regel zur Anwendung ausgewählt, so wird sie zuerst durch die Funktion `mkRule` in die *prozedurale Form* gebracht. Die prozedurale Form einer Regel ist prinzipiell vom Typ `Expression -> Expression`, da sie einen Ausdruck auf einen anderen Ausdruck abbildet. Da eine Regelanwendung jedoch potentiell interaktiv ist, also einen Seiteneffekt auf die `GUI` haben kann, wählen wir als Typisierung

```
type InteractiveRule = Expression -> (Expression -> GUI ()) -> GUI ()
```

Zu lesen ist diese Signatur wie folgt: Eine `InteractiveRule` ist eine Funktion, die zwei Argumente erwartet (den zu transformierenden Ausdruck und eine *Continuation*, die angibt, was mit dem Ergebnis zu machen ist) und einen Seiteneffekt auf die `GUI` hat. Der Grund, warum hier mit *Continuations* gearbeitet werden muß, wird im Abschnitt 5.6.1 erklärt.

Es folgt die Funktion `mkRule`, die die oben beschriebenen Konvertierung von der *schematischen Form* in die *prozedurale Form* erledigt. `mkRule` wird mit drei Argumenten aufgerufen (Anwendbarkeitsbedingungen, Eingabe- und Ausgabeschema) und liefert eine Funktion vom Typ `InteractiveRule` zurück.

```
mkRule :: [Term] -> Expression -> Expression -> InteractiveRule
mkRule cond inp out e cont
 = do let e' = mkUniqueVars e   -- remove all possible name clashes
          sub = match inp e'
      case sub of
      Fail -> do info "input scheme does not match"
                 cont e
      Ok s -> do let c  = applySubTerms s cond
                     ps = prologQuery db c
                 case ps of
                 [] -> do info "applicability condition failed"
```

```
                    cont e
          ps -> do let es = [ let o    = applySub (mkSubst p)
                                                  (applySub' s out)
                                  pars = difference (parameters o)
                                         (freevars_e (instantiate
                                                        out))
                                  forb = freevars_e o
                              in mapPara (replaceByNewVar2 pars
                                                           forb) o
                            | p <- ps ]
                    selectExpr es (\expr ->
                          do enterParameters expr cont)
```

Wird die mit `mkRule` erzeugte Regel später auf einen Ausdruck `e` angewandt, so passiert folgendes: Zuerst werden Namenskonflikte durch Variablenumbenennungen in `e` beseitigt. Dann wird ein Pattern Matching des neuen Ausdrucks `e'` mit dem Eingabeschema der Regel ausgeführt. Geht dieses schief, so erhält der Benutzer die Meldung „input scheme does not match". Ansonsten wird die Anwendbarkeitsbedingung `cond` mit der gefundenen Substitution `s` instantiiert. Die instantiierte Anwendbarkeitsbedingung wird dann mit dem integrierten Prolog-System überprüft. Rückgabe ist eine Liste von möglichen Substitutionen `ps`, welche die Bedingung wahr machen. Ist die Liste leer, so erhält der Benutzer die Meldung „applicability condition failed". Ansonsten werden die möglichen Substitutionen auf das Ausgabeschema angewandt. Man erhält eine Liste `es` von möglichen Ergebnissen, aus denen der Benutzer abschließend das gewünschte Ergebnis auswählt (mit `selectExpr`). Eventuell müssen dann noch die gewünschten Belegungen für freie Schemaparameter eingegeben werden (`enterParameters`). Als letztes wird die Continuation `cont` ausgeführt.

Der Aufruf von `mkRule` erfolgt mit einer Continuation, die einen Ausdruck als Argument erwartet und diesen an der Anwendungsstelle der Transformationsregel in den Syntaxbaum des Programms einbaut.

#### 5.4.4.1 Pattern Matching

Der *Pattern Matcher* vergleicht einen Ausdruck mit dem Eingabeschema einer Regel und liefert je nach Ausgang des Vergleichs `Fail` bzw. die Substitution, die auf das Eingabeschema angewandt werden muß. Eine formale Herleitung des hier verwendeten Algorithmus befindet sich im Kapitel 9.6.2 in [BW89].

Substitutionen werden als Funktionen von Variablennamen nach Ausdrücken repräsentiert.

```
type Subst  = Name -> Expression
```

Der Typ `LSubst` ist eine Erweiterung des Typs `Subst` um das Element `Fail`.

```
data Maybe a = Fail | Ok a
type LSubst  = Maybe Subst
```

`iden` ist die leere Substitution.

```
iden :: Subst
iden x  = EPara x
```

Die Funktion `extend` erweitert eine Substitution $s$ um eine Abbildung der Variable $x$ auf den Ausdruck $e$.

```
extend :: Eq a => (a -> b) -> a -> b -> a -> b
extend s x e y = e   , if x==y
               = s y , otherwise
```

Die Funktion `match` erledigt schließlich die eigentliche Aufgabe des Pattern Matchings.

```
match :: Expression -> Expression -> LSubst
match p e = match1 (Ok iden) (p,e)
```

Hier wurde ein zusätzliches (mit `Ok iden` initialisiertes) Argument eingeführt, in dem die Substitution im Laufe der Berechnung akkumuliert wird. Die einzelnen Fälle der Funktion `match1` ergeben sich aus der Definition der abstrakten Syntax.

```
match1 :: LSubst -> (Expression, Expression) -> LSubst
match1 (Ok s) (EPara x, e) = Ok (extend s x e) , if s x == EPara x
                           = Ok s              , if s x == e
                           = Fail              , otherwise
match1 (Ok s) (EConst c, EConst d) = if c==d then (Ok s) else Fail
match1 (Ok s) (EVar x, EVar y)     = if x==y then (Ok s) else Fail
match1 (Ok s) (ECon c cs, ECon d ds)
        = if c==d then foldl match1 (Ok s) (zip cs ds) else Fail
match1 (Ok s) (EAp e1 e2, EAp e3 e4)
        = foldl match1 (Ok s) [(e1,e3),(e2,e4)]
match1 (Ok s) (ELam xs e1, ELam ys e2)
        = foldl match1 (Ok s) ((e1,e2):(zip xs ys))
match1 (Ok s) (ECase e1 as1, ECase e2 as2)
        = if length as1 == length as2
          then foldl match1 (Ok s) ((e1,e2) : concat (zipWith f as1 as2))
          else Fail
          where f (e1,e2) (e3,e4) = [(e1,e3),(e2,e4)]
match1 (Ok s) (ELet ds1 e1, ELet ds2 e2)
        = if length ds1 == length ds2
          then foldl match1 (Ok s) ((e1,e2) : concat (zipWith f ds1 ds2))
          else Fail
          where f (e1,e2) (e3,e4) = [(e1,e3),(e2,e4)]
match1 _ _ = Fail
```

#### 5.4.4.2 Überprüfen von Anwendbarkeitsbedingungen

Die Überprüfung von Anwendbarkeitsbedingungen geschieht mit einem integrierten Prolog-System für Gofer, welches von Mark Jones entwickelt wurde und Teil der *Gofer Standard Distribution* ist. [Jon93b]

Die Verwendung von Prolog hat den Vorteil, daß auch partiell instantiierte Bedingungen überprüft werden können. Die Rückgabe ist in diesem Fall nicht nur ein Wahrheitswert, sondern eine Liste von Substitutionen, welche die Bedingung wahr machen.

*Beispiel:* Transformationsregel zur Umwandlung vom Rekursionsschema für linear rekursive Funktionen ins Schema für tail-rekursive Funktionen

```
lintotailrec trivial solve modify next combine e =
```

```
monoid combine e =>>
linrec trivial solve modify next combine
      =:=
(tailrec (trivial.snd) (\ (a,b) -> combine a (solve b))
        (\ (a,b) -> (combine a (modify b), next b))) . (\x -> (e,x))
```

Das Problem bei dieser Regel ist, daß der Parameter `e` zwar in Bedingung und Ausgabeschema, aber nicht im Eingabeschema vorkommt. Seine Belegung müßte also entweder vom Benutzer eingegeben oder vom System „erraten“ werden, bevor die Bedingung überprüft werden kann. Das Prolog-System ist jedoch in der Lage, diese Belegung herzuleiten:

Angenommen, die obige Regel soll auf den Ausdruck `linrec (<2) (const 1) id (λx -> x-1) (*)` angewandt werden. Der Parameter `combine` wird durch das Pattern Matching mit `(*)` belegt. Damit kann die Bedingung `monoid combine e` zumindest teilweise instantiiert werden, nämlich zu `monoid (*) e`.

`monoid (*) e` wird nun als Anfrage ans Prolog-System geschickt. Das Prolog-System findet in seiner Wissensdatenbank unter anderem die Hornklauseln

```
monoid f e = assoc f /\ neutral f e

assoc (+) = Yes
assoc (*) = Yes
assoc (++) = Yes

neutral (+)  0  = Yes
neutral (*)  1  = Yes
neutral (++) [] = Yes
```

und ermittelt `e = 1` als einzige Substitution, die die Bedingung wahr macht.

Wenn mit den Klauseln der Datenbank keine erfüllende Substitution hergeleitet werden kann, so wird die Bedingung als nicht erfüllt betrachtet. In diesem Fall erhält der Benutzer die Meldung „applicability condition failed“ und der Transformationsschritt wird nicht durchgeführt. Deswegen sollte man die Datenbank um alle notwendigen Gesetze erweitern, bevor man eine bestimmte Programmtransformation mit ULTRA durchführt.

Für eine zukünftige Programmversion wäre denkbar, Anwendbarkeitsbedingungen, die nicht vom Prolog-System bewiesen werden können, auf Wunsch des Benutzers als offene Beweisverpflichtungen mitzuschleppen und den Transformationsschritt trotzdem durchzuführen.

### 5.4.5 Strategien

Die schematische Form, in der die Strategien nach dem Einlesen der Regelkataloge vorliegen, hat folgende Gestalt:

```
data Strategy = AndThen Strategy Strategy
              | OrElse  Strategy Strategy
              | Repeat  Strategy
              | DepthFirst Strategy
              | Subtrees Strategy
```

```
                    | Tactic  String
                    | Reverse String
                    | Call String
```

Wird eine Strategie angewandt, so wird sie von der Funktion `mkStrategy` in einen Algorithmus übersetzt, der diese Strategie ausführt. Um die bei einigen Strategien nötige sequentielle Ausführung von Transformationen zu erzwingen, wurde der *Continuation Style* verwendet.

Im Continuation Style wird einer Funktion ein zusätzliches Argument, die sogenannte *Continuation* übergeben. Anstelle ein Ergebnis zurückzuliefern, wendet die Funktion dann diese Continuation auf das Ergebnis an.

*Beispiel 1:* Realisierung einer einfachen Regelanwendung

Die elementare Strategie `Tactic` *name* holt die schematische Repräsentation der Transformationsregel *name* aus der Regeldatenbank und bringt diese mit `mkRule` (siehe Abschnitt 5.4.4) in die prozedurale Form.

```
mkStrategy :: Strategy -> InteractiveRule
mkStrategy (Tactic x) e cont = do let (_,cond,inp,out) = getRule x
                                  mkRule cond inp out e cont
```

`e` ist hier der Ausdruck, auf den die Strategie angewandt werden soll. `cont` ist die Continuation, die danach auf das Ergebnis angewandt werden soll.

*Beispiel 2:* Realisierung des ANDTHEN-Operators

Mit „s1 ANDTHEN s2“ soll eine Strategie erzeugt werden, die zuerst Strategie `s1` und bei Erfolg anschließend Strategie `s2` auf das Ergebnis anwendet. Der entsprechende Code lautet:

```
mkStrategy (AndThen s1 s2) e cont
 = mkStrategy s1 e (\e' -> if e==e' then cont e
                                    else mkStrategy s2 e' cont)
```

Zuerst wird `s1` auf `e` angewandt. Das Ergebnis `e'` wird mit `e` verglichen. Wenn sich der Ausdruck nicht geändert hat, wird sofort mit `cont` fortgefahren. Ansonsten wird `s2` auf `e'` angewandt und erst dann `cont` aufgerufen.

### 5.4.6 Navigation mit Back und Forward

#### 5.4.6.1 Anforderung

Durch Drücken des *Back* Buttons soll der letzte Transformationsschritt rückgängig gemacht werden können. Durch mehrmalige Anwendung von *Back* soll der Benutzer bis zum Ausgangspunkt der Programmentwicklung zurückkehren können. *Forward* soll die Umkehrfunktion von *Back* sein und die Rückkehr in die „Gegenwart“, bis zur aktuellsten Programmversion, ermöglichen.

Sobald der Benutzer einen Transformationsschritt ausführt, ist die „Zukunft“ gelöscht. Die Geschichte der Entwicklung wird also nicht als Baum, sondern nur als lineare Liste verwaltet.

#### 5.4.6.2 Grundidee

Wir benötigen zwei Stacks, jeweils einen für die Vergangenheit und die Zukunft. Vor jeder Ausführung eines Transformationsschritts wird das aktuelle Programm auf den Vergangenheits-Stack gelegt und der Zukunfts-Stack wird gelöscht. Bei einer *Back*-Operation wird das aktuelle Programm auf den Zukunfts-Stack gelegt und das oberste Element vom Vergangenheits-Stack wird geholt und als aktuelles Programm eingesetzt. Eine *Forward*-Operation verlauft symmetrisch dazu. Implementieren wir die Stacks mittels Listen, lautet unsere benötigte Datenstruktur also

```
type History = ([Program],[Program])  -- Past && Future
```

#### 5.4.6.3 Optimierung

Diese vorläufige Lösung geht mit dem Speicherplatz allerdings noch sehr verschwenderisch um. Es ist eigentlich nicht nötig, vor jedem Schritt das komplette Programm auf den Stack zu legen. Stattdessen reicht es aus, lediglich die Stelle im Programm, an der sich etwas ändert, sowie den Ausdruck, der an dieser Stelle steht, zu speichern.

Die Stelle im Programm wird mittels Dewey-Notation repräsentiert. Die Dewey-Notation ist eine eindeutige Pfadnotation für Knoten in Bäumen: eine Liste von Zahlen, die angibt, wie das als Syntaxbaum dargestellte Programm von der Wurzel aus zu durchlaufen ist, um an die gesuchte Stelle zu gelangen. Unsere optimierte Datenstruktur lautet damit

```
type History   = ([DeltaPair],[DeltaPair])  -- Past && Future
type DeltaPair = ([Int],Expression)
```

Bei einer *Back*-Operation wird nun zuerst das oberste Element $(d, e)$ vom Vergangenheits-Stack gepopt. Mittels $d$ wird dann das aktuelle Programm durchlaufen. Man gelangt damit an den Ausdruck $e'$, der beim letzten Transformationsschritt an dieser Stelle ins Programm eingesetzt wurde. Der Ausdruck $e'$ wird nun im Programm wieder durch $e$ ersetzt, was den Schritt rückgängig macht. Das Tupel $(d, e')$ wird anschließend auf den Zukunfts-Stack gelegt.

Eine $Forward$-Operation verläuft wiederum symmetrisch dazu.

## 5.5 Der Simplifier

Das Drücken des *Simplify Buttons* aktiviert den Simplifier. Dieser kennt verschiedene Taktiken zur Vereinfachung von Ausdrücken wie symbolische Auswertung, Ausnutzen von Kontextinformationen oder Auflösen von Gleichungen nach einer Variablen und wendet diese Taktiken innerhalb einer bestimmten Strategie an, die in diesem Kapitel beschrieben wird.

Oft wird nach Ausführung einer Vereinfachung der Simplifier rekursiv auf das Ergebnis angewandt, um weitere Vereinfachungen zu versuchen. In solchen Fällen muß darauf geachtet werden, daß der Simplifier auch immer terminiert. Sofern das zu vereinfachende Programm terminiert, ist dies sichergestellt, da bei allen rekursiven Aufrufen die Ausdrücke bezüglich einer rekursiven Pfadordnung [vH97] kleiner werden. Terminiert das zu vereinfachende

Programm nicht, so wird die Terminierung des Simplifiers erzwungen, indem nach einer bestimmten Anzahl von Schritten abgebrochen wird.

### 5.5.1 Arithmetische Ausdrücke

Arithmetische Ausdrücke, die nur die Operatoren (+), (-) und (*) sowie höchstens eine Variable enthalten (die allerdings beliebig oft auftreten kann), werden in Normalform gebracht, d. h. ausmultipliziert und nach Potenzen sortiert. Aus `5-x*x+10+x*2*x` wird beispielsweise `x*x+15`.

Dazu wird der Ausdruck zuerst von der Funktion `encodeAri` in eine Zwischenform gebracht, die zur Darstellung von Polynomen besser geeignet ist: einem Tupel, bestehend aus einem Variablennamen und einer Koeffizientenliste.

```
type AriExpr = (String, [Int])
```

Anschließend wird der Ausdruck von der Funktion `decodeAri` wieder in den Typ `Expression` zurückkonvertiert.

5-x*x+10+x*2*x

↓ `encodeAri :: Expression -> AriExpr`

("x",[15,0,1])

↓ `decodeAri :: AriExpr -> Expression`

x*x+15

### 5.5.2 Gleichungen

Bei einer Gleichung wird zuerst der Simplifier rekursiv auf die linke und rechte Seite der Gleichung angewandt. Werden anschließend beide Seiten als gleich oder ungleich erkannt, so evaluiert der gesamte Ausdruck zu `True` oder `False`.

Wenn beide Seiten zu einer nichtleeren Liste evaluieren, so wird die Transformation `(x:xs) == (y:ys) =:= x==y && xs==ys` angewandt. Anschließend wird versucht, das Ergebnis weiter zu vereinfachen. Mit dieser Technik läßt sich z. B. der Ausdruck `[1,2,x,4,5] == [1,2,3,4,5]` zu `x==3` reduzieren.

Evaluieren beide Seiten der Gleichung zu einem arithmetischen Ausdruck (siehe oben), so wird der Gleichungslöser gestartet, der Gleichungen in einer Variablen bis zum zweiten Grad auflösen kann. Aus `3*x+9==2*x+16` würde z. B. `x==7`.

| vorher | nachher |
|---|---|
| `3*x+9==2*x+16` | `x==7` |
| `x+x==2*x` | `True` |
| `5*x+1==5*x+2` | `False` |
| `x*x-5*x+6==0` | `x==2 \|\| x==3` |

Abbildung 5.5: Beispiele zur Simplifikation von Gleichungen

### 5.5.3 Logische Ausdrücke

Zur Vereinfachung von logischen Ausdrücken werden einfache Transformationsregeln verwendet, die die bekannten Eigenschaften der Booleschen Operatoren ausnutzen, wie z. B.

```
e || True = True
e || False = e
e && True = e
e && False = False
not (not e) = e
```

### 5.5.4 Standard-Goferfunktionen

Die Definitionen vieler Standard-Goferfunktionen können ebenfalls als einfache Transformationsregeln interpretiert werden, mit denen sich Ausdrücke vereinfachen lassen. Beispiele hierfür sind

```
head (x:xs) = x
tail (x:xs) = xs
fst (x,y) = x
snd (x,y) = y
flip f a b = f b a
id e = e
(const c) e = c
(f . g) x = f (g x)
f $ x = f x
```

### 5.5.5 Reduktion von Lambda-Ausdrücken

Eine Anwendung einer Lambda-Abstraktion auf ein Argument wie z. B., `(λx -> x+1) 4` kann durch *Beta-Reduktion* vereinfacht werden. Bei einer Beta-Reduktion wird der gesamte Ausdruck durch eine Instanz des Rumpfes der Lambda-Abstraktion ersetzt, in der alle freien Vorkommen des formalen Parameters durch das Argument ersetzt wurden. Im obigen Beispiel tritt der formale Parameter `x` einmal frei im Rumpf `x+1` auf und wird durch das Argument `4` ersetzt. Das Ergebnis der Beta-Reduktion lautet also `4+1`.

Vorsicht ist geboten, wenn die Namen der formalen Parameter nicht eindeutig sind. Bei einer Beta-Reduktion von `(λx -> (λx -> (+) (x-1)) x 3) 9` darf nicht für das innere `x` substituiert werden, da es in der äußeren Lambda-Abstraktion nicht frei vorkommt. Das korrekte Ergebnis lautet `(λx -> (+) (x-1)) 9 3`.

Ein schwieriger Fall, das sogenannte *name capture problem* [PJ87], tritt dann ein, wenn es einen Namenskonflikt zwischen einer freien Variablen des Arguments mit einem formalen Parameter im Rumpf der Lambda-Abstraktion gibt:

```
(\f -> (\x -> f (f x))) x
```

Hier wäre es falsch, den formalen Parameter `f` durch `x` zu ersetzen und damit den Ausdruck zu `(\x -> x (x x))` zu reduzieren. Bevor die Beta-Reduktion ausgeführt werden darf, muß der formale Parameter `x` in der inneren Lambda-Abstraktion umbenannt werden. Diese Operation ist als *Alpha-Konversion* bekannt [PJ87].

```
(\f -> (\x -> f (f x))) x = (\f -> (\y -> f (f y))) x
                          = (\y -> x (x y))
```

### 5.5.6 Normalisierung von Konditionalen

Wenn die Bedingung eines Konditionals wiederum ein Konditional ist, wird eine Transformation angewandt, die in der Literatur als *Normalisierung eines Konditionals* [Par90] bekannt ist. Aus dem Ausdruck

```
case (case C of
            A1 -> C1
            A2 -> C2
            .. -> ..
            Am -> Cm) of
      B1 -> E1
      B2 -> E2
      .. -> ..
      Bn -> En
```

wird dabei

```
case C of
       A1 -> case C1 of
                  B1 -> E1
                  B2 -> E2
                  .. -> ..
                  Bn -> En
       A2 -> case C2 of
                  B1 -> E1
                  B2 -> E2
                  .. -> ..
                  Bn -> En
       .. -> ..
       Am -> case Cm of
                  B1 -> E1
                  B2 -> E2
                  .. -> ..
                  Bn -> En
```

### 5.5.7 Distributieren von Funktionsanwendungen über Konditionale

Funktionsanwendungen distributieren über Fallunterscheidungen. Die Ausdrücke `f (if c then 1 else 2)` und `if c then f 1 else f 2` sind z. B. identisch. Funktionsanwendungen nach innen zu ziehen bringt Vorteile, denn anschließend können sie eventuell durch

Verwenden von Kontextinformation (5.5.12) vereinfacht werden. Im Beispiel oben kann `f 1` und `f 2` ausgewertet werden. Eventuell wird die Funktion `f` dadurch sogar überflüssig.

```
F (case C of
          A1 -> E1
          A2 -> E2
          .. -> ..
          An -> En
  )
```

wird transformiert zu

```
case C of
       A1 -> F E1
       A2 -> F E2
       .. -> ..
       An -> F En
  )
```

Anschließend wird der Simplifier auf die einzelnen Zweige angewandt.

Die syntaktische Anwendbarkeitsbedingung dieser Transformation ist, daß die freien Variablen von F sich nicht mit den Patternvariablen der $A_i$ überschneiden dürfen. Sollte diese Bedingung nicht erfüllt sein, so wird sie wie üblich durch Umbenennung der Patternvariablen erzwungen.

*Beispiel:*

```
x + case ys of []     -> 0
               (x:xs) -> x
```

Die freie Variable von `(x+)` ist $x$. Die Patternvariablen der zweiten Alternative sind $x$ und $xs$. Die Variable $x$ kommt beidesmal vor. Daher ist die Transformation nicht anwendbar! Das falsche Ergebnis wäre ansonsten

```
case ys of []     -> x + 0
           (x:xs) -> x + x
```

Daher wird die Patternvariable $x$ vor Ausführung der Transformation in eine noch nicht verwendete Variable umbenannt.

```
x + case ys of []      -> 0
               (x1:xs) -> x1
```

Jetzt kann die Transformation ausgeführt werden. Wir erhalten das Ergebnis

```
case ys of []      -> x + 0
           (x1:xs) -> x + x1
```

### 5.5.8 Weitere Distributivitätsregeln

Analog zu Funktionsanwendungen distributiert die Konstruktion von Tupeln über Fallunterscheidungen. Beispiele hierfür sind

```
(e1, if a then b else c) =:= if a then (e1, b) else (e1, c)
(if a then b else c, e2) =:= if a then (b, e2) else (c, e2)
```

Weiterhin distributieren Funktionsanwendungen und Tupelkonstruktionen auch über Let-Anweisungen.

```
f (let a=b in c)    =:= let a=b in f c
(e1, let a=b in c) =:= let a=b in (e1, c)
(let a=b in c, e2) =:= let a=b in (c, e2)
```

Auch hier müssen wieder Namenskonflikte zwischen den von der Let-Anweisung gebundenen Variablen und den freien Variablen des umliegenden Kontexts durch Variablenumbenennungen beseitigt werden, bevor die entsprechende Distributivitätsregel angewandt werden darf.

### 5.5.9 Eliminierung von Konditionalen

Eine Fallunterscheidung wird eliminiert, wenn einer der vier folgenden Fälle vorliegt:

1. Alle Alternativen sind identisch.

   ```
   case C of
        B1 -> E
        B2 -> E
        .. -> E
        Bn -> E
   ```

   Der Ausdruck reduziert sich hier zu `E`.

2. Jede Alternative ist identisch zu ihrem eigenen Pattern.

   ```
   case C of
        B1 -> B1
        B2 -> B2
        .. -> ..
        Bn -> Bn
   ```

   Hier wird der Ausdruck zu `C` reduziert.

3. Nur ein Pattern ist mit der Bedingung unifizierbar.

   ```
   case (1:ys) of
          []     -> 0
          (x:xs) -> x + sum xs
   ```

   Hier kann der erste Fall nie eintreten, da die Liste `(1:ys)` unabhängig vom Wert von `ys` nie leer sein kann. Daher kann der Ausdruck zu `1 + sum ys` reduziert werden.

   Sind mehrere Alternativen möglich, so kann die Fallunterscheidung zwar nicht eliminiert werden, jedoch werden zumindest die unmöglichen Alternativen gelöscht.

   ```
   case (1:zs) of
          []       -> 0
          [x]      -> 1
          (x:y:ys) -> 2
   ```

   würde immerhin zu

```
case (1:zs) of
        [x]      -> 1
        (x:y:ys) -> 2
```

reduziert.

4. Spezialfall: `if c then False else True`
   Dies wird reduziert zu `not c`.

Der entsprechende Code lautet:

```
elimCond :: Rule
elimCond (ECase c [(EConst (CBool True), EConst (CBool False)),
                   (EConst (CBool False), EConst (CBool True))])
        = eval (EAp (EVar "not") c)
elimCond co@(ECase c as)
 = if and (map (uncurry (==)) as) then c
   else if length (nub (map snd as)) == 1 then snd (head as)
   else let as' = map modify (filter possible (map muv as))
            muv (lhs,rhs)      = (lhs, mkUniqueVars rhs)
            possible (lhs,rhs) = matchable lhs c
            modify (lhs,rhs)   = ...
        in tryElimCond (ECase c as')
elimCond e      = e
```

Die Funktion `modify` vereinfacht dabei die rechte Seite eines Zweiges mit Hilfe der partiellen Substitution, die sich aus der Unifikation der Bedingung mit der linken Seite des Zweiges ergibt. Betrachten wir das Beispiel

```
case (1:zs) of
        []       -> 0
        [x]      -> x
        (x:y:ys) -> x*x
```

Die erste Alternative kann hier nie eintreten, wird also gelöscht. Für die zweite Alternative ergibt die Unifikation der Bedingung mit dem Pattern die Substitution { `x=1, zs=[]` }. `modify` vereinfacht deswegen die rechte Seite zu `1`. Die rechte Seite der dritten Bedingung wird analog zu `1*1` vereinfacht. Wir erhalten also

```
case (1:zs) of
        [x]      -> 1
        (x:y:ys) -> 1*1
```

was im weiteren Verlauf der Simplifikation letztendlich zu `1` vereinfacht wird.

### 5.5.10 Automatisches Unfold

Ein Funktionsaufruf mit einem Argument, das keine freien Variablen enthält, kann komplett ausgewertet werden. Dazu wird automatisch ein *Unfold* ausgeführt und anschließend das Ergebnis simplifiziert.

```
sumsquares []     = 0
sumsquares (x:xs) = x*x + sumsquares xs

test = sumsquares [1,2,3]
```

Wird hier der Simplifer auf `sumsquares [1,2,3]` angewandt, so wird `sumsquares` entfaltet, da `[1,2,3]` keine freien Variablen enthält. Anschließend wird der Simplifier auf das Ergebnis angewandt. Nach einigen Schritten gelangt er zum Ausdruck `1 + sumsquares [2,3]` und entfaltet abermals die Funktion `sumsquares`. Letztendlich evaluiert der Ausdruck durch dieses Verfahren, der sogenannten *symbolischen Auswertung*, zu `14`.

So gesehen kann ULTRA auch als (etwas langsamer) Gofer-Interpreter verwendet werden.

### 5.5.11 Vereinfachung von Let-Anweisungen

Bei einer Let-Anweisung wird zuerst einmal der Rumpf durch rekursive Anwendung des Simplifiers vereinfacht. Zusätzlich kann man aber noch die Gleichungen vereinfachen. Dazu werden folgende Maßnahmen durchgeführt:

- Dekomposition von Gleichungen

  ```
  let ST [w,(x,y),z] = ST [1,(2,3),4]
  in f w x y z
  ```

  Steht auf beiden Seiten einer Gleichung der selbe Konstruktor (z. B. eine Liste oder ein Tupel), so wird dieser Konstruktor eliminiert. Die Gleichung wird dabei eventuell in mehrere Gleichungen zerlegt, wobei die genaue Anzahl von der Stelligkeit des Konstruktors abhängt. Im ersten Schritt wird bei obigem Beispiel der redundante Konstruktor ST eliminiert. Anschließend wird die Gleichung `[w,(x,y),z] = [1,(2,3),4]` in drei Gleichungen zerlegt. Wir erhalten dadurch

  ```
  let w     = 1
      (x,y) = (2,3)
      z     = 4
  in f w x y z
  ```

  Im letzten Schritt wird das Tupel `(x,y)` zerlegt.

  ```
  let w = 1
      x = 2
      y = 3
      z = 4
  in f w x y z
  ```

  Danach folgen einige automatische *Unfolds* (siehe oben) und abschließend die Eliminierung der dadurch redundant gewordenden Gleichungen (siehe unten). Das Endergebnis lautet

  ```
  f 1 2 3 4
  ```

- Löschen von redundanten Gleichungen

  ```
  let even n   = if n==0
                 then True
                 else odd (n-1)
      odd n    = if n==0
                 then False
                 else even (n-1)
      number n = even n || odd n
      answer   = 42
  in even x
  ```

Hier ist `even` die einzige lokale Definition, die direkt vom Rumpf der Let-Anweisung benötigt wird. `even` benötigt zudem noch die Funktion `odd`, welche wiederum auf `even` zurückgreift. Die Funktion `number` und die Variable `answer` werden allerdings nie benötigt.

Welche lokalen Definitionen überhaupt erreichbar sind, wird mit einer Abhängigkeitsanalyse ermittelt. Dazu wird ein Graph konstruiert, dessen Knoten die einzelnen lokalen Definitionen sind. Wenn eine Definition eine andere benötigt, so wird eine (gerichtete) Kante zwischen diesen Knoten eingetragen. Dann wird ausgehend von den lokalen Definitionen, welche im Rumpf vorkommen, eine Tiefensuche gestartet, die alle erreichbaren Knoten markiert. Im obigen Beispiel startet die Tiefensuche bei `even` und ermittelt `even` und `odd` als erreichbar. Alle unerreichbaren Definitionen werden anschließend gelöscht. Das Ergebnis lautet in diesem Fall also

```
let even n   = if n==0
               then True
               else odd (n-1)
    odd n    = if n==0
               then False
               else even (n-1)
in even x
```

Falls bei diesem Prozeß keine einzige Gleichung mehr übrigbleibt, so wird die Let-Anweisung gelöscht und durch ihren Rumpf ersetzt. Aus `let x=5 in y` wird also `y`.

### 5.5.12 Ausnutzen von Kontextinformationen

Im Ausdruck

```
if a==0
then if a==b
     then if b==0
          then 1
          else 2
     else 3
else 4
```

ist die innerste If-Anweisung auf den ersten Blick nicht vereinfachbar. Betrachtet man jedoch den gesamten Kontext, so fällt auf, daß immer wenn die Anweisung erreicht wird, sowohl `a==0` als auch `a==b` gelten muß. Aus `a==0` und `a==b` folgt aber `b==0` und daher kann der Else-Zweig von `if b==0 then 1 else 2` nie erreicht werden. Daher kann der Ausdruck zu

```
if a==0
then if b==0
     then 1
     else 3
else 4
```

vereinfacht werden.

#### 5.5.12.1 Grundidee

In den Zweigen von If- und Case-Anweisungen werden Zusicherungen (*Assertions*) generiert, d. h. Aussagen, die innerhalb dieser Zweige gelten.

Diese Assertions werden dann im abstrakten Syntaxbaum des Ausdrucks hinuntergereicht. Unterwegs werden sie dabei verwendet, um Ausdrücke zu vereinfachen. Trifft eine Assertion auf eine weitere If- oder Case-Anweisung, so werden die Assertions, die in den Zweigen dieses Konditionals generiert werden, der von oben kommenden Assertion hinzugefügt.

Durch diese Kombination zweier Assertions kann die entstehende neue Assertion inkonsistent werden. Ist dies der Fall, so ist der aktuelle Programmzweig nicht erreichbar und wird deswegen gelöscht. Bleibt dadurch in einem Konditional nur ein Fall übrig, so wird die Fallunterscheidung eliminiert.

Für den Ausdruck aus dem obigen Beispiel werden folgende Assertions abgeleitet:

```
{true} if a==0
       then {a==0} if a==b
                   then {a==b==0} if b==0
                                  then {a==b==0} 1
                                  else {false} 2
                   else {a==0 && a/=b} 3
       else {a/=0} 4
```

#### 5.5.12.2 Realisierung

Zur Repräsentation einer *Assertion* wurde die Datenstruktur gewählt, die in [Ses96] zur Optimierung eines ML Pattern Matchers verwendet wurde: ein Tupel aus einer Liste von bekannten Gleichheiten und einer Liste von bekannten Ungleichheiten.

```
type Assertion = ([(Expression, Expression)],[(Expression, Expression)])
```

Die Assertion $\{a = 0\}$, die im Beispiel oben für den Then-Zweig der äußeren If-Anweisung generiert wird, hat die Repräsentation `([(EVar "a", 0)], [])`. Die Assertion $\{a \neq 0\}$ für den Else-Zweig lautet `([], [(EVar "a", 0)])`.

Bei der Kombination zweier Assertions wird in der dadurch entstehenden neuen Assertion die transitive Hülle über den bekannten Gleichheiten berechnet, wobei die Symmetrie der Gleichheitsrelation berücksichtigt wird. Aus $\{a = 0, a = b\}$ wird also $\{a = 0, a = b, b = 0\}$. Außerdem werden redundante Gleichheiten $\{1 = 1\}$ und Ungleichheiten $\{[] \neq (x : xs)\}$ eliminiert. Komplexe Gleich- und Ungleichheiten werden zu zerlegen versucht. Aus $\{[a, b] = [1, 2]\}$ wird $\{a = 1, b = 2\}$.

Eine Inkonsistenz kann in einer Assertion auf drei Arten erkannt werden.

- Eine der Gleichheiten ist offensichtlich falsch.
  *Beispiel:* $\{a = 4, a = 5, 4 = 5\}$
- Eine der Ungleichheiten ist offensichtlich falsch.
  *Beispiel:* $\{a = 1, b = 1, a \neq b, 1 \neq 1\}$.
- Ein Tupel kommt sowohl in der Gleichheits- als auch in der Ungleichheitsliste vor.
  *Beispiel:* $\{a = b, a \neq b\}$

## 5.6 Benutzeroberfläche

Die Benutzeroberfläche von ULTRA ist in TkGofer geschrieben. TkGofer ist eine Erweiterung von Gofer, die für die funktionale Implementierung grafischer Benutzeroberflächen an der Universität Ulm entwickelt wurde. Eine umfassende Einführung in TkGofer wird in [VSS96] gegeben.

### 5.6.1 TkGofer

Die in TkGofer eingeführten GUI Funktionen kommunizieren mit dem *Tcl/Tk* System [Ous95], das für die eigentliche grafische Ausgabe zuständig ist. Gegenüber der ausschließlichen Verwendung von Tcl/Tk hat man mit TkGofer noch den Vorteil der Typsicherheit. Tcl/Tk kennt nur den Typ *String*, wogegen die Widgets in TkGofer Typinformationen enthalten. Man kann beispielsweise ein Eingabefeld definieren, das nur Werte vom Typ `Int` zuläßt.

Mit TkGofer war es relativ einfach, die grafische Oberfläche für ULTRA zu erstellen. Probleme traten lediglich dann auf, wenn während einer Berechnung (z. B. während der Ausführung eines Transformationsschritts) vom Benutzer eine Eingabe verlangt werden mußte, bevor die Berechnung weiterlaufen konnte.

*Beispiel:* Eingabe eines Ausdrucks

Beim Aufruf der Funktion `enterExpression` sollte ein Fenster mit einer Eingabezeile und einem OK-Button geöffnet werden, Der Benutzer sollte dann einen Ausdruck eingeben und den OK-Button drücken. Dann sollte die Funktion `enterExpression` zurückkehren und die Eingabe als Ergebnis zurückliefern. Intuitiv würde man also denken, die Funktion `enterExpression` hat einen Seiteneffekt auf die GUI und liefert eine `Expression` zurück und kommt vermutlich zu folgendem ersten Ansatz.

```
enterExpression :: GUI Expression
enterExpression
   = do w <- window []
        e <- entry [] w
        b <- button [text "OK", command (returnExpr w e)] w
      pack (e ^-^ b)
      where returnExpr w e = do txt <- getValue e
                                let expr = parseExpr txt
                                closeWindow w
                                result expr
```

Aufgerufen würde diese Funktion dann etwa so:

```
... = do actionBefore
         e <- enterExpression
         actionAfter
```

Leider funktioniert dieser Ansatz nicht. Erstens stimmt die Typisierung nicht: die letzte Aktion, die von `enterExpression` ausgeführt wird, ist `pack (e ^-^ b)`, eine Aktion vom Typ `GUI ()`. Daher muß der Typ von `enterExpression` ebenfalls `GUI ()` sein. Zweitens wird versucht, ein Kommando vom Typ `GUI Expression` auf den OK-Button zu legen, was in TkGofer nicht möglich ist. (Nur `GUI ()`-Aktionen dürfen auf Buttons gelegt werden.)

Das Problem läßt sich aber lösen durch Programmierung im *Continuation Style.*

Wir übergeben $\lambda$`e -> actionAfter` der Funktion `enterExpression` als Continuation. Und anstatt wie bisher das Ergebnis zurückzugeben, wendet `enterExpression` am Ende diese Continuation auf das Ergebnis an. Wir schreiben `enterExpression` also neu als

```
enterExpression :: (Expression -> GUI ()) -> GUI ()
enterExpression cont
   = do w <- window []
        e <- entry [] w
        b <- button [text "OK", command (returnExpr w e cont)] w
        pack (e ^-^ b)
     where returnExpr w e cont = do txt <- getValue e
                                    let expr = parseExpr txt
                                    closeWindow w
                                    cont expr
```

und den Aufruf schreiben wir um zu

```
... = do actionBefore
         enterExpression (\e -> actionAfter)
```

Die Typisierung von `enterExpression` ist nun korrekt. Auf dem OK-Button liegt eine Aktion vom Typ *GUI ().* Und das Programm verhält sich genau so wie geplant.

# Kapitel 6 Ergebnisse

Abschließend sollen nun in diesem Kapitel die Ergebnisse dieser Arbeit zusammengefaßt und diskutiert werden. Dabei wird vor allem auf die Vor- und Nachteile des ULTRA-Systems eingegangen (Abschnitt 6.1). Im Abschnitt 6.2 werden einige Erweiterungsmöglichkeiten vorgestellt, die ULTRA noch leistungsfähiger machen könnten.

## 6.1 Zusammenfassung

Da ULTRA ein System ist, das sich immer noch in der Entwicklung befindet, bezieht sich die folgende Bewertung auf den Stand Mitte August 1997.

### 6.1.1 Verwendbarkeit

Der Parser ist schnell genug für kleine Programme. Die erkannte Sprache ist zur Zeit eine Teilmenge von Gofer. Den Parser auf die komplette Sprache Gofer auszuweiten, wäre allerdings kein Problem.

Der Simplifier ist relativ leistungsfähig. Während einer Transformation bleiben nur selten Routinearbeiten übrig, die von Hand zu erledigen sind. Ein Nachteil ist jedoch, daß der Simplifikationsalgorithmus nur schwer zu erweitern ist.

Die Beschreibungssprache für Strategien ist ausdrucksstark genug, um interessante Strategien formulieren zu können, wie z. B. das Auffinden der richtigen Folge von Regelanwendungen mittels Backtracking. Ein Beispiel hierfür wird im Abschnitt A.3 gegeben.

Die größte Schwäche des Systems liegt darin, daß man oft in einer Transformation steckenbleibt, weil eine bestimmte Regel fehlt oder eine semantische Bedingung nicht vom Prolog-System bewiesen werden kann. In diesem Fall muß man leider die Transformation abbrechen, die Regelkataloge editieren und einen neuen Versuch starten.

Ein weiterer Nachteil ist, daß bisher keine Typinformation berücksichtigt wird. Manche Transformationsregeln verlangen jedoch als Anwendbarkeitsbedingung, daß ein Ausdruck von einem bestimmten Typ sein muß. Beispielsweise darf man die Regel

```
etaAbstraction f = f =:= \x -> f x
```

nur dann anwenden, wenn der Ausdruck `f` vom Typ `a -> b` ist. Dies kann vom System aber noch nicht überprüft werden.

Da ULTRA jedoch im Rahmen zweier anschließender Diplomarbeiten weiterentwickelt wird, werden diese Mankos demnächst behoben sein. Vorerst ist das System allerdings für die Lösung großer Probleme nicht geeignet.

Das System eignet sich jedoch bereits hervorragend für den Einsatz in der Lehre. Bei einer Einführungsvorlesung in die transformationelle Programmentwicklung wäre eine Verwendung von ULTRA in den Übungen denkbar, um das Durchführen von Programmtransformationen und das Formulieren von Regeln zu trainieren. Die Bedienung des Systems über die grafische Oberfläche ist leicht zu erlernen, so daß sich auch bei Anfängern schnell Erfolgserlebnisse einstellen. Transformationsregeln lassen sich leicht formulieren. Die automatische Auflösung von syntaktischen Anwendbarkeitsbedingungen hält die Regeln einfach und verhindert Leichtsinnsfehler.

### 6.1.2 Korrektheit

Da ULTRA nicht selbst mit formalen Methoden, sondern konventionell programmiert worden ist, kann angesichts des Programmumfangs von etwa 4900 Zeilen Gofer-Code[1] letztendlich leider nicht garantiert werden, daß das System absolut fehlerfrei ist.

Nach bestem Wissen und Gewissen halte ich das Programm jedoch für korrekt. An kritischen Stellen wurde ein „defensiver Programmierstil“ verwandt: im Zweifelsfall wird eher zuviel getan, um z. B. die korrekte Ausführung eines Transformationsschritts zu gewährleisten. Beispielsweise werden eventuell Variablen zu oft umbenannt, um Namenskonflikte zu beseitigen.

Getestet wurde das System mit etwa 30 Beispielprogrammen, von denen die eine Hälfte typische Fallbeispiele darstellten (ein Teil dieser Programme wird in Anhang A vorgestellt), während die andere Hälfte speziell konstruierte Programme waren, die jeweils einen bestimmten Programmteil (Simplifier, Pattern Matching Compiler,...) unter die Lupe nahmen - mit dem Ziel, eine Code-Abdeckung zu erreichen.

### 6.1.3 Geschwindigkeit

Die Effizienz der Implementierung war in dieser Arbeit eigentlich nicht von Bedeutung. Dennoch möchte ich hier eine kurze Geschwindigkeitsanalyse der einzelnen Komponenten von ULTRA vorstellen:

Die Geschwindigkeit des Parsers ist unabhängig von der Länge des zu parsenden Programms und beträgt im Mittel etwa 40 Programmzeilen pro Sekunde.

Die eigentliche Ausführung eines Transformationsschritts, also Pattern Matching, Überprüfung der Anwendbarkeitsbedingung und Berechnung des Ausgabeprogramms ist zeitlich nicht merkbar.

[1] Eigentlich ist dies nicht einmal besonders viel. Das in C und Tcl programmierte CIP-System[BEH+87] besteht z. B. aus 67000 Zeilen Code und besitzt eine geringere Leistungsfähigkeit als ULTRA.

Die Geschwindigkeit des Simplifiers variiert zwischen 50 und 2000 Programmtransformationen pro Sekunde, je nach Komplexität der ausgeführten Schritte. Dies ist ausreichend, falls der Simplifier im vorgesehenen Sinne (und nicht als Gofer Interpreter) verwendet wird.

Die Geschwindigkeit des Pretty Printers liegt bei etwa 20 Programmzeilen pro Sekunde.

## 6.2 Erweiterungsmöglichkeiten

Es existieren noch viele Möglichkeiten, die Leistungsfähigkeit von ULTRA zu steigern. Folgende Erweiterungen sind z. B. denkbar:

- Vollständiger Parser für Gofer
  Bisher wird nur eine Teilmenge der Sprache Gofer erkannt. Noch nicht implementiert sind z. B. Listenkomprehensionen oder Konditionale in Guard-Schreibweise.

- Besserer Pretty-Printer
  Der Pretty-Printer gibt Programme noch keineswegs perfekt aus. Z. B. werden manchmal Ausdrücke geklammert, obwohl es nicht notwendig wäre.

- Offene Beweisverpflichtungen
  Wenn eine semantische Anwendbarkeitsbedingung nicht vom System bewiesen werden kann, könnte man den Benutzer fragen, ob er den Transformationsschritt unter der Annahme, daß die Bedingung gilt, ausführen möchte. Die Bedingung müßte dann als offene Beweisverpflichtung mitgeführt werden.

- Higher Order Pattern Matching
  Um einen Ausdruck als potentielle Instanz für eine Regelanwendung zu erkennen, muß das Pattern Matching des Ausdrucks mit dem Eingabeschema der entsprechenden Regel erfolgreich sein. Beim *First Order Pattern Matching*, welches in ULTRA verwendet wird, müssen dabei Ausdrücke syntaktisch passen. Die Instanz

  ```
  fac n = if n==0 then 1 else n * fac (n-1)
  ```

  paßt syntaktisch nicht auf die Form

  ```
  f x = if trivial x then solve x else g x (f (next x))
  ```

  und ein *First Order Pattern Matching* geht schief. Damit die richtige Substitution

  ```
  f       -> fac
  x       -> n
  trivial -> (0==)
  solve   -> const 1
  g       -> (*)
  next    -> flip (-) 1
  ```

gefunden wird, muß der Ausdruck vorher mit drei Vorbereitungsschritten (*commute*, *constIntro*, *flipIntro*) in die syntaktisch passende Form

```
fac n = if 0==n then (const 1) n else n * fac (flip (-) 1 n)
```

gebracht werden. Mit *Higher Order Pattern Matching* [Pau93] könnte man dagegen die richtige Substitution direkt bestimmen. *Higher Order Pattern Matching* hat jedoch den Nachteil, daß es nicht vollständig mechanisierbar ist und manchmal die Unterstützung des Benutzers benötigt.

- Bessere Beschreibungssprache für Strategien
  Die Beschreibungssprache für Strategien ist ebenfalls noch verbesserbar. Denkbar wären z. B. parametrisierte Strategien, d. h. Strategien, die eine andere Strategie als Argument nehmen.

- Noch leistungsfähigerer Simplifier
  Ein leistungsfähiger Simplifier nimmt dem Benutzer sehr viel Routine-Arbeit ab. Je mehr der Simplifier allein erledigen kann, desto besser kann sich der Benutzer auf die entscheidenden Schritte bei einer Programmtransformation konzentrieren.

- Benutzeroberfläche
  Die Benutzeroberfläche könnte noch mit mehr Funktionen versehen werden. So fehlt z. B. bisher die Möglichkeit, den kompletten Zustand des Systems speichern zu können, um eine Transformation ein anderes Mal fortzusetzen.
  Wenn das Editieren der Regeldatenbank zur Laufzeit möglich wäre, würde dies die Arbeit ebenfalls erleichtern.

Einen Eindruck von der aktuellen Leistungsfähigkeit des Systems kann sich der interessierte Leser im Anhang A verschaffen, wo einige mit ULTRA durchgeführte Beispieltransformationen vorgestellt werden.

# Anhang A Fallbeispiele

Dieses Kapitel soll einige Beispielanwendungen des ULTRA-Systems zeigen. Zu verschiedenen Programmentwicklungen sind hier die kommentierten Protokolle wiedergegeben.

## A.1 Sumsquares

Wir gehen vom folgenden Programm aus, welches die Summe der Quadrate einer Liste von Zahlen berechnet.

```
map :: (a -> b) -> [a] -> [b]
map f xs = if xs==[] then [] else f (head xs) : map f (tail xs)

sum :: [Int] -> Int
sum xs = if xs==[] then 0 else head xs + sum (tail xs)

square :: Int -> Int
square x = x*x

sumsquares :: [Int] -> Int
sumsquares xs = sum (map square xs)
```

Unser Ziel ist, eine rekursive, von `sum` und `map` unabhängige Version der Funktion `sumsquares` herzuleiten. Dazu beginnen wir mit einem *Unfold* auf `map` innerhalb der Funktion `sumsquares`. Nach den dieser Aktion folgenden automatisch durchgeführten Vereinfachungen sind wir nur noch zwei kleine Schritte vom Ziel entfernt.

```
----------------------------------------------------------------------
Original Program
----------------------------------------------------------------------

map f xs = if xs == []
           then []
           else (f (head xs):map f (tail xs))
```

```
sum xs = if xs == []
         then 0
         else (head xs) + (sum (tail xs))

square x = x * x

sumsquares xs = sum (map square xs)

----------------------------------------------------------------------
Focus:  sumsquares
Action: Unfold map
----------------------------------------------------------------------

sumsquares xs = if xs == []
                then 0
                else sum (square (head xs):map square (tail xs))

----------------------------------------------------------------------
Focus:  sumsquares
Action: Unfold sum
----------------------------------------------------------------------

sumsquares xs = if xs == []
                then 0
                else (square (head xs)) + (sum (map square (tail xs)))

----------------------------------------------------------------------
Focus:  sumsquares
Action: Fold sumsquares
----------------------------------------------------------------------

sumsquares xs = if xs == []
                then 0
                else (square (head xs)) + (sumsquares (tail xs))

----------------------------------------------------------------------
Focus:  sumsquares
Action: Unfold square
----------------------------------------------------------------------

sumsquares xs = if xs == []
                then 0
                else ((head xs) * (head xs)) + (sumsquares (tail xs))

----------------------------------------------------------------------
Transformation finished.
----------------------------------------------------------------------
```

## A.2 Tree Recovery

Von einem Binärbaum, dessen Knoten eindeutige Bezeichner haben, sind ein *Preorder*-Durchlauf und ein *Inorder*-Durchlauf gegeben. Daraus soll der Baum rekonstruiert werden und sein *Postorder*-Durchlauf ausgegeben werden.

*Beispiel:* Gegeben ist der Preorder-Durchlauf `DBACEGF` und der Inorder-Durchlauf `ABCDEFG`. Daraus kann der folgende Binärbaum rekonstruiert werden:

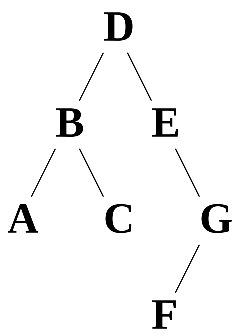


Abbildung A.1: *Rekonstruierter Binärbaum*

Der gesuchte Postorder-Durchlauf lautet also `ACBFGED`.

Das Problem *Tree Recovery* war eine der Aufgaben, die beim lokalen ACM-Programmierwettbewerb 1997 an der Universität Ulm gestellt wurden. Eine mögliche Lösung des Problems in Gofer lautet

```
data Tree = Node Char Tree Tree
          | Nil

recover :: String -> String -> Tree
recover [] [] = Nil
recover (p:ps) is = let (ls,r:rs) = break (p==) is
                    in Node p (recover (take (length ls) ps) ls)
                              (recover (drop (length ls) ps) rs)

postord :: Tree -> String
postord Nil          = ""
postord (Node c l r) = postord l ++ postord r ++ [c]

solve :: String -> String -> String
solve ps is = postord (recover ps is)
```

Die Funktion `recover` rekonstruiert aus Preorder- und Inorder-Durchlauf den Baum. `postord` erzeugt den Postorder-Durchlauf eines Baums. Die Lösung des Problems ergibt sich durch Komposition dieser Funktionen: `solve = postord . recover`

Es stellt sich allerdings die Frage, ob es wirklich nötig ist, den Baum als Datenstruktur aufzubauen, wenn am Ende doch nur dessen Postorder-Durchlauf gefragt ist. Wir versuchen nun deshalb, mit ULTRA das Programm zu optimieren.

```
----------------------------------------------------------------------
Focus:  solve
Action: Unfold recover
----------------------------------------------------------------------

solve ps is
 = case ps of
        [] -> ""
        p : ps1 -> let (ls, r : rs) = break ((==) p) is
                   in postord (Node p (recover (take (length ls) ps1) ls)
                                      (recover (drop (length ls) ps1) rs))
```

Nach dem *Unfold*-Schritt wurde hier der Aufruf von `postord` vom Simplifier zuerst in die Case- und dann in die Let-Anweisung gezogen. Als nächstes entfalten wir `postord`.

```
----------------------------------------------------------------------
Focus:  solve
Action: Unfold postord
----------------------------------------------------------------------

solve ps is
      = case ps of
             [] -> ""
             p : ps1 -> let (ls, r : rs) = break ((==) p) is
                        in (postord (recover (take (length ls) ps1) ls)) ++
                           ((postord (recover (drop (length ls) ps1) rs))
                           ++ [p])
```

Und schon haben wir zwei Ausdrücke der Form `postord (recover ...)`, die wir zu `solve` falten können.

```
----------------------------------------------------------------------
Focus:  solve
Action: Fold solve
----------------------------------------------------------------------

solve ps is
      = case ps of
             [] -> ""
             p : ps1 -> let (ls, r : rs) = break ((==) p) is
                        in (solve (take (length ls) ps1) ls) ++
                           ((postord (recover (drop (length ls) ps1) rs))
                           ++ [p])

----------------------------------------------------------------------
Focus:  solve
Action: Fold solve
----------------------------------------------------------------------

solve ps is
      = case ps of
             [] -> ""
             p : ps1 -> let (ls, r : rs) = break ((==) p) is
```

```
                         in (solve (take (length ls) ps1) ls) ++
                            ((solve (drop (length ls) ps1) rs) ++ [p])

----------------------------------------------------------------------
Transformation finished.
----------------------------------------------------------------------
```

Das optimierte Programm benötigt die Datenstruktur für Bäume nicht mehr. Ebenso sind `recover` und `postord` überflüssig geworden.

## A.3 Maximum Segment Sum

Beim *Maximum Segment Sum Problem* geht es darum, das Maximum der Summen aller Segmente[1] einer Liste von Zahlen zu berechnen, wobei die Zahlen positiv, negativ oder gleich null sein können.

*Beispiel:* Bei der Liste [-3, 14, -1, 5, -9] ist [14, -1, 5] das Segment mit der maximalen Summe. Die gesuchte Antwort wäre hier also 18.

Im Finale des *ACM Programming Contests* 1997 war das *Maximum Segment Sum Problem* eine der gestellten Programmieraufgaben. Die Eingabe wurde dabei von der Jury so groß gewählt, daß die Teilnehmer gezwungen waren, einen Algorithmus zu finden, der das Problem *in linearer Zeit* löst.

Der naive Ansatz, alle $O(n^2)$ Segmente einer gegebenen Liste von $n$ Zahlen zu generieren und jedes dieser Segmente in $O(n)$ Schritten aufzusummieren, ergibt lediglich einen $O(n^3)$-Algorithmus. Mittels Programmtransformation ist es allerdings relativ leicht, einen $O(n)$-Algorithmus aus dem naiven Ansatz herzuleiten. Eine solche Herleitung wird in [Bir89] gezeigt.

### A.3.1 Ein $O(n^3)$-Algorithmus

Wir definieren uns zwei Hilfsfunktionen `inits` und `tails`, die alle Anfangs- bzw. Schlußsegmente einer Liste liefern.

```
inits :: [a] -> [[a]]
inits = takeUntil null . iterate init

tails :: [a] -> [[a]]
tails = takeUntil null . iterate tail
```

`inits [1,2,3]` liefert z. B. `[[1,2,3], [1,2], [1], []]`.
`tails [1,2,3]` liefert `[[1,2,3], [2,3], [3], []]`.
Alle Segmente einer Liste erhalten wir durch

```
segs :: [a] -> [[a]]
segs  = concat . map tails . inits
```

Eine erste Lösung des *Maximum Segment Sum Problems*, die uns als Ausgangspunkt der kommenden Programmentwicklung dienen wird, können wir nun elegant schreiben als

```
mss :: [Int] -> Int
mss  = maximum . map sum . segs
```

### A.3.2 Herleitung eines $O(n)$-Algorithmus

```
----------------------------------------------------------------------
Original Program
----------------------------------------------------------------------
```

[1]Segment = zusammenhängende Teilfolge

Abbildung A.2: *Ausgangssituation der Programmentwicklung*

```
segs = concat . ((map tails) . inits)

mss = maximum . ((map sum) . segs)

----------------------------------------------------------------------
Focus:  mss
Action: Unfold segs
----------------------------------------------------------------------

mss = maximum . ((map sum) . (concat . ((map tails) . inits)))

----------------------------------------------------------------------
Focus:  mss
Action: Apply assocL
----------------------------------------------------------------------

mss = maximum . (((map sum) . concat) . ((map tails) . inits))

----------------------------------------------------------------------
```

Abbildung A.3: *Definition der Regel mapPromotion*

```
Focus:  mss
Action: Apply mapPromotion
----------------------------------------------------------------------

mss = maximum . ((concat . (map (map sum))) . ((map tails) . inits))

----------------------------------------------------------------------
Focus:  mss
Action: Apply assocL
----------------------------------------------------------------------

mss = (maximum . (concat . (map (map sum)))) . ((map tails) . inits)

----------------------------------------------------------------------
Focus:  mss
Action: Apply assocL
----------------------------------------------------------------------

mss = ((maximum . (concat . (map (map sum)))) . (map tails)) . inits

----------------------------------------------------------------------
Focus:  mss
Action: Apply assocL
----------------------------------------------------------------------

mss = (((maximum . concat) . (map (map sum))) . (map tails)) . inits

----------------------------------------------------------------------
Focus:  mss
Action: Unfold maximum
----------------------------------------------------------------------

mss = ((((foldl1 max) . concat) . (map (map sum))) . (map tails)) . inits

----------------------------------------------------------------------
Focus:  mss
```

Abbildung A.4: *Definition der Regel foldl1Promotion*

```
Action: Apply foldl1Promotion
----------------------------------------------------------------------

mss = ((((foldl1 max) . (map (foldl1 max))) . (map (map sum)))
        . (map tails)) . inits

----------------------------------------------------------------------
Focus:  mss
Action: Apply assocR
----------------------------------------------------------------------

mss = (((foldl1 max) . (map (foldl1 max))) . ((map (map sum))
        . (map tails))) . inits

----------------------------------------------------------------------
Focus:  mss
Action: Apply assocR
----------------------------------------------------------------------

mss = ((foldl1 max) . ((map (foldl1 max)) . ((map (map sum)) .
        (map tails)))) . inits

----------------------------------------------------------------------
Focus:  mss
Action: Apply mapDistribution
----------------------------------------------------------------------

mss = ((foldl1 max) . ((map (foldl1 max)) . (map ((map sum) . tails))))
        . inits

----------------------------------------------------------------------
Focus:  mss
Action: Apply mapDistribution
----------------------------------------------------------------------

mss = ((foldl1 max) . (map ((foldl1 max) . ((map sum) . tails))))
```

Abbildung A.5: *Definition der Regel mapDistribution*

```
        . inits

----------------------------------------------------------------------
Focus:  mss
Action: Unfold sum
----------------------------------------------------------------------

mss = ((foldl1 max) . (map ((foldl1 max) . ((map (foldl (+) 0)) . tails))))
        . inits
```

Den Ausdruck `(foldl1 max) . ((map (foldl (+) 0)) . tails)` kann man mittels Horner- Schema effizienter berechnen. Dies ist ein entscheidender Schritt, welcher die Zeitkomplexität des Algorithmus von $O(n^3)$ auf $O(n^2)$ reduziert.

Abbildung A.6: *Auswertung mittels Horner-Schema*

```
----------------------------------------------------------------------
Focus:  mss
Action: Apply hornerScheme
----------------------------------------------------------------------
```

```
mss = ((foldl1 max) . (map (foldl ((\a b -> max (a + b) 0)) 0))) . inits

----------------------------------------------------------------------
Focus:  mss
Action: Apply assocR
----------------------------------------------------------------------

mss = (foldl1 max) . ((map (foldl ((\a b -> max (a + b) 0)) 0)) . inits)
```

Der zweite entscheidende Schritt, welcher die Zeitkomplexität nun auf $O(n)$ reduziert, ist die Anwendung des *Accumulation Lemmas*.

Abbildung A.7: *Accumulation Lemma*

```
----------------------------------------------------------------------
Focus:  mss
Action: Apply accumulationLemma
----------------------------------------------------------------------

mss = (foldl1 max) . (scanl ((\a b -> max (a + b) 0)) 0)

----------------------------------------------------------------------
Transformation finished.
----------------------------------------------------------------------
```

### A.3.3 Analyse der Transformation

Man kann die einzelnen Schritte der eben durchgeführten Transformation in drei Klassen einteilen:

- 3 *Unfolds*: `maximum`, `segs` und `sum`
- 8 Umklammerungen (*assocL* und *assocR*)
- 6 „echte“ Transformationen (*map promotion*, *foldl1 promotion*,...)

Die Strategie, die bei der Transformation verfolgt wurde, war im Prinzip:

1. Entfalte die Definitionen `maximum`, `segs` und `sum`.
2. Versuche durch Umklammern eine der Transformationen *map promotion*, *foldl1 promotion*, *map distribution*, *horner scheme* oder *accumulation lemma* anwendbar zu machen.
3. Falls Schritt 2 erfolglos $\rightarrow$ fertig!
4. Ansonsten wende die Transformation an und gehe zurück zu Schritt 2

### A.3.4 Automatisierung

Wir versuchen nun, die Schritte 2 bis 4 als Strategie zu formulieren - in der Hoffnung, daß diese in Zukunft für ähnliche Probleme, die mit Segmenten zu tun haben, von Nutzen sein kann.

Die Strategie *try* soll alle algebraischen Umformungen, die eine Effizienzsteigerung bringen könnten, ausprobieren. Zusätzlich zu den in unserer Transformation gebrauchten Regeln fügen wir noch zwei weitere (*filter promotion* und *horner2*, eine verallgemeinerte Horner-Regel) hinzu. Schließlich wollen wir keine Strategie entwerfen, die nur ganz speziell für das *Maximum Segment Sum Problem* geeignet ist. Wir tun also so, als wüßten wir nicht, welche Regeln für unser Problem gebraucht werden und welche nicht.

Abbildung A.8: Die Strategie *try*

Die Strategie *assoc* klammert Ausdrücke um, falls es möglich ist. Ausdrücke der Form $A \oplus (B \oplus C)$ werden zu $(A \oplus B) \oplus C$ und umgekehrt, vorausgesetzt, daß der Operator $\oplus$ assoziativ ist.

Die Strategie *search* wendet zuerst *try* auf den aktuellen Ausdruck an. Ändert sich der Ausdruck dadurch nicht, wo wird *search* rekursiv auf die Teilterme des Ausdrucks angewandt. Bringt auch dies keinen Erfolg, so wird mit *assoc* versucht, den Ausdruck umzuklammern. Wenn dies gelingt, wird abermals zuerst die Strategie *try* und dann *search* rekursiv auf die Teilterme probiert. Ist auch dies vergeblich, so wird die Umklammerung durch einen erneuten Aufruf von *assoc* rückgängig gemacht. Die gesamte Strategie steht innerhalb einer REPEAT-Schleife, die solange ausgeführt wird, solange sich der Ausdruck bei einer Iteration verändert.

Abbildung A.9: Die Strategie *assoc*

Durch das Rückgängigmachen der Umklammerung wird gewährleistet, daß sich der Ausdruck in einer Iteration nicht verändert, wenn keine wirkliche Vereinfachung gelingt. Ansonsten würde die REPEAT-Schleife nicht terminieren.

Abbildung A.10: Die Strategie *search*

Um die Terminierung von *search* sicherzustellen, dürfen in *try* nur Transformationsregeln aufgenommen werden, die die Ausdrücke in einem bestimmten Sinne kleiner machen. Dann können zwangsläufig nach endlich vielen Schritten keine Regeln mehr angewandt werden.

### A.3.5 Test

Wir testen nun unsere Strategie am *Maximum Segment Sum Problem* aus. Dazu entfalten wir die Definitionen von `maximum`, `segs` und `sum` und starten anschließend die Strategie *search*.

```
----------------------------------------------------------------------
Original Program
----------------------------------------------------------------------

segs = concat . ((map tails) . inits)

mss = maximum . ((map sum) . segs)
```

```
----------------------------------------------------------------------
Focus:   mss
Actions: Unfold maximum, sum & segs
----------------------------------------------------------------------

mss = (foldl1 max) .
      ((map (foldl (+) 0)) . (concat . ((map tails) . inits)))

----------------------------------------------------------------------
Focus:  mss
Action: Use Strategy search
----------------------------------------------------------------------

mss = (foldl1 max) . (scanl ((\a b -> max (a + b) 0)) 0)

----------------------------------------------------------------------
Transformation finished.
----------------------------------------------------------------------
```

*search* benötigt etwa 7 Sekunden, um das Ergebnis herzuleiten.

## A.4 Ein weiteres Segment-Problem

Als nächstes testen wir nun, ob unsere Strategie auch ein verwandtes Problem bewältigt. Für eine Liste von Elementen wollen wir die Länge des längsten Segments bestimmen, dessen Elemente alle eine vorgegebene Eigenschaft $p$ besitzen.

```
lps :: [a] -> Int
lps p = maximum . map length . filter (all p) . segs
```

### A.4.1 1. Versuch

Wir entfalten `maximum`, `length` und `segs` und wenden *search* an.

```
----------------------------------------------------------------------
Focus:   lps
Actions: Unfold maximum, length & segs
----------------------------------------------------------------------

lps p = (foldl1 max) . ((map (foldl ((\a b -> a + 1)) 0)) .
        ((filter (all p)) . (concat . ((map tails) . inits))))

----------------------------------------------------------------------
Focus:  lps
Action: Use Strategy search
----------------------------------------------------------------------

lps p = ((foldl1 max) . (map ((((foldl1 max) .
        (map (foldl ((\a b -> a + 1)) 0))) . (filter (all p))) . tails)))
        . inits

----------------------------------------------------------------------
Transformation finished.
----------------------------------------------------------------------
```

Die von ULTRA gefundene neue Version von `lps` ist zwar ein wenig effizienter, aber eine echte Reduktion der Zeitkomplexität wurde nicht erreicht. Das Problem ist, daß das Horner-Schema nicht eingesetzt werden kann - zum einen, weil die in `length` verwendete Funktion $\lambda$`a b -> a+1` nicht assoziativ ist, zum anderen, weil durch das Filtern „Lücken“ in der Liste der Segmente entstehen.

Starten wir deshalb einmal von einer anderen Ausgangsversion.

### A.4.2 2. Versuch

Anstatt die gültigen Segmente zu filtern, wenden wir eine Funktion `preprocess` auf die Segmente an, die alle gültigen Elemente in Einsen und alle ungültigen in $-\infty$ verwandelt. Für alle gültigen Segmente $xs$ gilt dann `sum` $xs$ `= length` $xs$, für alle ungültigen gilt `sum` $xs$ `=` $-\infty$.

```
lps :: (a -> Bool) -> [a] -> Int
lps p = maximum . map sum . map (preprocess p) . segs
```

```
preprocess :: (a -> Bool) -> [a] -> [Int]
preprocess p = map (\x -> if p x then 1 else negativeInfinity)
```

Wie wir später `negativeInfinity` realisieren, sei vorerst dahingestellt. Auf jeden Fall liefert uns dieser Algorithmus ebenfalls eine Lösung unseres Problems.

Ein ähnlicher Trick wird übrigens in der in [Bir89] vorgestellten Transformation angewandt, wo ein Dummy-Element $\omega$ für die unendlich kurze Liste eingeführt wird, so daß `length` $\omega$ = $-\infty$ und $xs$ ++ $\omega$ = $\omega$ = $\omega$ ++ $xs$ (für alle Listen $xs$) gilt.

```
----------------------------------------------------------------------
Focus:   lps
Actions: Unfold maximum, sum, preprocess & segs
----------------------------------------------------------------------

lps p
    = (foldl1 max) . ((map (foldl (+) 0)) .
      ((map (map ((x -> if p x then 1
                               else negativeInfinity)))) .
      (concat . ((map tails) . inits))))

----------------------------------------------------------------------
Focus:  lps
Action: Use Strategy search
----------------------------------------------------------------------

lps p = (foldl1 max) .
        (scanl ((\a b -> max (a + (((x -> if p x then 1
                                                  else negativeInfinity))
                              b)) 0)) 0)
```

Diesmal war die Strategie *search* erfolgreich! Die entscheidenden Regeln, die zur Anwendung kamen, waren die verallgemeinerte Horner-Regel (Abb. A.11) und das *Accumulation Lemma* (Abb. A.7).

Abbildung A.11: *Verallgemeinerte Horner-Regel*

Das Ergebnis läßt sich aber noch weiter vereinfachen.

```
-----------------------------------------------------------------------
Focus:  lps
Action: Simplify
-----------------------------------------------------------------------

lps p = (foldl1 max) .
        (scanl ((\a b -> if p b then max (a + 1) 0
                                else max (a + negativeInfinity) 0)) 0)
```

Den Ausdruck `max (a + negativeInfinity) 0` können wir zu `0` vereinfachen. Damit sind wir um das Problem, `negativeInfinity` realisieren zu müssen, herumgekommen.

```
-----------------------------------------------------------------------
Focus:  lps
Action: Replace max (a + negativeInfinity) 0 by 0
-----------------------------------------------------------------------

lps p = (foldl1 max) . (scanl ((\a b -> if p b
                                        then max (a + 1) 0
                                        else 0)) 0)
```

`max (a + 1) 0` kann hier noch zu `a + 1` vereinfacht werden, denn an dieser Stelle gilt immer die Invariante $a \geq 0$.

```
-----------------------------------------------------------------------
Focus:  lps
Action: Replace max (a + 1) 0 by a + 1
-----------------------------------------------------------------------

lps p = (foldl1 max) . (scanl ((\a b -> if p b
                                        then a + 1
                                        else 0)) 0)

-----------------------------------------------------------------------
Transformation finished.
-----------------------------------------------------------------------
```

Wieder haben wir somit einen $O(n^3)$-Algorithmus zu einem $O(n)$-Algorithmus vereinfacht.

## A.5 Eine Anwendung auf den eigenen Quellcode

Zum Abschluß möchte ich noch eine ganz besondere Transformation vorstellen: eine Anwendung von ULTRA auf sich selbst. Mit anderen Worten: hier wurde das fertig implementierte System verwendet, um dessen eigenen Quellcode nachträglich zu optimieren.

In einer älteren Programmversion wurde, um eine Funktion `f` in einem Ausdruck `e` zu entfalten, die Funktion `unfold f e` aufgerufen.

```
unfold :: SCDefn -> Expression -> Expression
unfold (f_name, f_args, f_body) e
       = let pat = mkPattern f_name f_args
             sub = match pat e
         in case sub of
                 Fail  -> e
                 Ok s -> let args = parameters pat
                         in applySub' s (mkScheme args f_body)
```

Die Funktion `unfold` führte ein Pattern Matching der linken Seite von `f` mit `e` durch. Verlief dieses erfolgreich, so wurde die gefundene Substitution auf die rechte Seite von `f` angewandt. Anderenfalls wurde `e` unverändert zurückgeliefert.

Die Hilfsfunktion `mkPattern` erwartete einen Funktionsnamen und eine Liste der Funktionsparameter und konstruierte daraus einen Ausdruck, der anschließend fürs Pattern Matching verwendet werden konnte.
`mkPattern "f" [EVar "x"]` liefert beispielsweise `EAp (EVar "f") (EVar "x")`.

```
mkPattern :: Name -> [Expression] -> Expression
mkPattern f args = mkPattern' (EVar f) args

mkPattern' :: Expression -> [Expression] -> Expression
mkPattern' e [] = e
mkPattern' e (a:as) = mkPattern' (EAp e (parametrize a)) as
```

Unfold-Aktionen in unterversorgten Funktionsaufrufen waren nicht möglich. Um dem abzuhelfen, wurde später die Funktion `lambdaUnlift` eingeführt, die eine Funktionsdefinition in die argumentlose Form bringt.

```
lambdaUnlift :: SCDefn -> SCDefn
lambdaUnlift (name, args, body) = (name, [], mkLambda args body)
```

Nachdem die entsprechenden Änderungen im Code abgeschlossen waren, waren alle Aufrufe der Funktion `unfold` vom Typ `unfold (lambdaUnlift f) e`. Damit war die Funktion `unfold` nun aber zu allgemein geworden, da sie nur noch für argumentlose Funktionsdefinitionen aufgerufen wurde. Alle Aufrufe vom Typ `unfold (lambdaUnlift f) e` wurden deshalb durch `unfold' f e` ersetzt, wobei `unfold'` vorerst folgendermaßen definiert war:

```
unfold' :: SCDefn -> Expression -> Expression
unfold' (f_name, f_args, f_body) (EVar f)
           = unfold (lambdaUnlift (f_name, f_args, f_body)) (EVar f)
unfold' _ e = e
```

Mit der nun folgenden mit ULTRA durchgeführten Transformation wurde diese erste Version von `unfold'` vereinfacht. Das Protokoll ist dabei gekürzt. Nur die an der Transformation beteiligten Funktionen aus dem Quellcode von ULTRA sind aufgeführt. Bei Funktionsdefinitionen, die aus vielen Gleichungen bestehen, sind nur die entscheidenden Gleichungen wiedergegeben.

```
----------------------------------------------------------------------
Original Program
----------------------------------------------------------------------

unfold (f_name, f_args, f_body) e
       = let pat = mkPattern f_name f_args
             sub = match pat e
         in case sub of
                 Fail  -> e
                 Ok s -> let args = parameters pat
                         in applySub' s (mkScheme args f_body)

lambdaUnlift (name, args, body)
             = (name, [], (mkLambda args body))

mkLambda u1 u2
         = case u1 of
                [] -> u2
                x : xs -> ELam [x] (mkLambda xs u2)

mkPattern f args = mkPattern' (EVar f) args

mkPattern' u1 u2
           = case u2 of
                  [] -> u1
                  a : as -> mkPattern' (EAp u1 (parametrize a)) as

match p e = match1 (Ok iden) (p, e)

match1 (Ok s) (EVar x, EVar y)
       = if x == y
         then Ok s
         else Fail

unfold' (f_name, f_args, f_body) (EVar f)
        = unfold (lambdaUnlift ((f_name, f_args, f_body))) (EVar f)
```

Wir beginnen mit dem Entfalten der Definitionen von `lambdaUnlift` und `unfold`.

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold lambdaUnlift
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = unfold ((f_name, [], (mkLambda f_args f_body))) (EVar f)
```

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold unfold
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = let sub = match pat (EVar f)
              pat = mkPattern f_name []
          in case sub of
                  Fail  -> EVar f
                  Ok s -> let args = parameters pat
                          in applySub' s (mkScheme args
                                            (mkLambda f_args f_body))
```

mkPattern wird nun mit einer leeren Liste im zweiten Argument aufgerufen, was auf eine mögliche weitere Vereinfachung hoffen läßt.

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold mkPattern
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = let pat = mkPattern' (EVar f_name) []
              sub = match pat (EVar f)
          in case sub of
                  Fail  -> EVar f
                  Ok s -> let args = parameters pat
                          in applySub' s (mkScheme args
                                            (mkLambda f_args f_body))

----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold mkPattern'
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = let sub = match pat (EVar f)
              pat = EVar f_name
          in case sub of
                  Fail  -> EVar f
                  Ok s -> let args = parameters pat
                          in applySub' s (mkScheme args
                                            (mkLambda f_args f_body))
```

Als nächstes entfalten wir die lokalen Definitionen von pat und sub.

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold pat
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
```

```
        = let sub = match pat (EVar f)
              pat = EVar f_name
          in case sub of
                 Fail  -> EVar f
                 Ok s -> let args = parameters (EVar f_name)
                         in applySub' s (mkScheme args
                                          (mkLambda f_args f_body))

----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold sub
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = let pat = EVar f_name
          in case match pat (EVar f) of
                 Fail  -> EVar f
                 Ok s -> let args = parameters (EVar f_name)
                         in applySub' s (mkScheme args
                                          (mkLambda f_args f_body))

----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold pat
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = case match (EVar f_name) (EVar f) of
               Fail  -> EVar f
               Ok s -> let args = parameters (EVar f_name)
                       in applySub' s (mkScheme args
                                        (mkLambda f_args f_body))
```

Wie man sieht, wurden die nun überflüssigen lokalen Definitionen von `sub` und `pat` vom System automatisch gelöscht. Als nächstes betrachten wir den Ausdruck `parameters (EVar f_name)`. Die Funktion `parameters` liefert die Liste der Schemaparameter eines Programmschemas. Da `EVar f_name` keine Schemaparameter enthält, kann der Ausdruck durch weitere *Unfolds* und anschließende Simplifikationen zur leeren Liste vereinfacht werden. Um die Transformation abzukürzen, verwenden wir jedoch den *Replace* Button und ersetzen `parameters (EVar f_name)` direkt durch `[]`.

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Replace parameters (EVar f_name) by []
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = case match (EVar f_name) (EVar f) of
               Fail  -> EVar f
               Ok s -> let args = []
                       in applySub' s (mkScheme args
                                        (mkLambda f_args f_body))
```

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Simplify
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = case match (EVar f_name) (EVar f) of
               Fail  -> EVar f
               Ok s -> applySub' s (mkScheme [] (mkLambda f_args f_body))
```

`mkScheme vars e` ersetzt alle Variablen `vars`, die in `e` vorkommen, durch Schemaparameter. `mkScheme []` ist somit die identische Abbildung `id`.

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Replace mkScheme [] by id
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = case match (EVar f_name) (EVar f) of
               Fail  -> EVar f
               Ok s -> applySub' s (id (mkLambda f_args f_body))

----------------------------------------------------------------------
Focus:  unfold'
Action: Simplify
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = case match (EVar f_name) (EVar f) of
               Fail  -> EVar f
               Ok s -> applySub' s (mkLambda f_args f_body)
```

Der Aufruf von `match` versucht hier, zwei Variablen gegeneinander zu matchen, was genau dann gelingt, wenn die Variablen identisch sind. Wir entfalten also den Aufruf von `match`.

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold match
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = case match1 (Ok iden) (EVar f_name, EVar f) of
               Fail  -> EVar f
               Ok s -> applySub' s (mkLambda f_args f_body)

----------------------------------------------------------------------
Focus:  unfold'
Action: Unfold match1
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = if f_name == f
          then applySub' iden (mkLambda f_args f_body)
```

```
          else EVar f
```

`applySub' s e` wendet die Substitution `s` auf den Ausdruck `e` an. `applySub' iden` ist somit die identische Abbildung `id`.

```
----------------------------------------------------------------------
Focus:  unfold'
Action: Replace applySub' iden by id
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = if f_name == f
          then id (mkLambda f_args f_body)
          else EVar f

----------------------------------------------------------------------
Focus:  unfold'
Action: Simplify
----------------------------------------------------------------------

unfold' (f_name, f_args, f_body) (EVar f)
        = if f_name == f
          then mkLambda f_args f_body
          else EVar f

----------------------------------------------------------------------
Transformation finished.
----------------------------------------------------------------------
```

Nach Abschluß dieser Transformation wurde im Quellcode von ULTRA die Funktion `unfold'` durch diese optimierte Version ersetzt.

# Literaturverzeichnis

[ASU86] A. Aho, R. Sethi und J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Series in Computer Science, 1986.

[BBB+85] F. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, W. Möller, F. Nickel, H. Partsch, P. Pepper, K. Samelson, M. Wirsing und H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L (Lecture Notes in Computer Science, Volume 183).* Springer Verlag, 1985.

[BD77] R. Burstall und J. Darlington. *A Transformation System for Developing Recursive Programs.* Journal of the ACM 24, Seiten 44–67, 1977.

[BEH+87] F. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner und P. Pepper. *The Munich Project CIP. Volume II: The Transformation System (Lecture Notes in Computer Science, Volume 292).* Springer Verlag, 1987.

[Bir89] R. Bird. *Lectures on Constructive Functional Programming.* In: M. Broy (Herausgeber): *Constructive Methods in Computing Science*, Band F 55. NATO ASI Series, 1989.

[BK94] A. Bayer und W. Kahl. *The Higher Object Programming System HOPS.* Technischer Bericht Universität der Bundeswehr München, 1994.

[BW89] R. Bird und Ph. Wadler. *Introduction to Functional Programming.* Prentice Hall International, 1989.

[Fok95] J. Fokker. *Functional Parsers.* In: J. Jeuring und E. Meijer (Herausgeber): *Advanced Functional Programming. First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 1995*, Band 925 der Reihe *Lecture Notes in Computer Science*, Seiten 1–23. Springer-Verlag, 1995.

[Jon93a] M.P. Jones. *An introduction to Gofer (draft)*, 1993.

[Jon93b] M.P. Jones. *Release notes for Gofer 2.28*, 1993. Included as part of the standard Gofer distribution.

[Lag96] J. Lagarias. *The 3x+1 Problem and its Generalizations.* Technischer Bericht AT&T Bell Laboratories, 1996.

[Ous95] J.K. Ousterhout. *Tcl und Tk.* Addison-Wesley, 1995.

[Par90] H. Partsch. *Specification and Transformation of Programs: A formal approach to software development.* Springer Verlag, 1990.

[Pau93] L.C. Paulson. *Introduction to Isabelle.* Technischer Bericht University of Cambridge, 1993.

[Pep84] P. Pepper. *A simple calculus for program transformations (inclusive of induction).* Science of Computer Programming 9, Seiten 221–262, 1984.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1987.

[PJL92] S.L. Peyton Jones und D. Lester. *Implementing Functional Languages: A Tutorial.* Prentice-Hall International Series in Computer Science, 1992.

[PJW93] S.L. Peyton Jones und Ph. Wadler. *Imperative Functional Programming.* In: *Proc. 20th ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, Januar 1993.

[PS83] H. Partsch und R. Steinbrüggen. *Program Transformation Systems.* Computing Surveys, Vol. 15, No. 3, Seiten 199–236, 1983.

[PS96] Peter Pepper und Douglas R. Smith. *A High-Level Derivation of Global Search Algorithms (with Constraint Propagation).* Science of Computer Programming, 1996. to appear.

[PV95] H. Partsch und T. Vullinghs. *Werkzeuge zur formalen Programmentwicklung.* Unterlagen zum Hauptseminar SS95 an der Universität Ulm, 1995.

[Ses96] P. Sestoft. *ML Pattern Match Compilation and Partial Evaluation.* Lecture Notes in Computer Science 1110, Seiten 449–454, 1996.

[Smi90] D. Smith. *KIDS: A Semiautomatic Program Development System.* IEEE Transactions on Software Engineering, Vol. 16, No. 9, 1990.

[SV96] W. Schulte und T. Vullinghs. *Funktionale Programmierung.* Skript zur Vorlesung an der Universität Ulm, 1996.

[Thi94] P. Thiemann. *Grundlagen der funktionalen Programmierung.* Teubner Verlag, 1994.

[vH97] F. von Henke. *Theorie und Praxis des maschinellen Beweisens.* Vorlesungsskript SS97 an der Universität Ulm, 1997.

[VSS96] T. Vullinghs, T. Schwinn und W. Schulte. *TkGofer User Manual.* Technischer Bericht University of Ulm, 1996.

[Wad95] Ph. Wadler. *Monads for functional programming.* In: *Advanced Functional Programming. First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, Seiten 24–52. Springer-Verlag, 1995.

Name: Mark Dettinger                                                    Matr. Nr.: 0259135

Erklärung

Ich erkläre, daß ich die Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ..............................................