

Verifikation und Fehlersuche mit Hilfe von Assertions für die nicht-strikte funktionale Programmiersprache Haskell

Thomas Harfeld

Diplomarbeit

15. Juni 2000

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie



Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20)

Danksagung

Ich möchte allen Personen danken, die mich bei der Anfertigung der vorliegenden Diplomarbeit unterstützt haben. Besonderer Dank für wertvolle Anregungen und eine ausgezeichnete Betreuung der Diplomarbeit gebührt Marko Schütz, Matthias Mann und Prof. Dr. Manfred Schmidt-Schauß.

Frankfurt am Main, den 15. Juni 2000

Thomas Harfeld

Versicherung gemäß Diplomprüfungsordnung

Ich versichere hiermit, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Frankfurt am Main, den 15. Juni 2000

Thomas Harfeld

Wichtiger Hinweis

Die nachfolgenden Ausführungen entsprechen nicht in allen Teilen dem Text der Diplomarbeit, der beim Prüfungsamt des Fachbereichs Informatik an der Johann Wolfgang Goethe-Universität in Frankfurt am Main eingereicht wurde.

Frankfurt am Main, den 3. Oktober 2000

Inhaltsverzeichnis

1	Einleitung	1
1.1	Problemstellung	1
1.2	Gang der Untersuchung	1
2	Grundlagen	3
2.1	Funktionale Programmiersprachen	3
2.2	Auswertungsmechanismen	4
2.3	Die Programmiersprache Haskell	8
2.3.1	Überblick	8
2.3.2	Polymorphismus	9
2.3.3	Pattern Matching	11
2.4	Implementierung nicht-strikter funktionaler Programmiersprachen	15
2.4.1	Auswertung auf einem Graphen zur schwachen Kopfnormalform	15
2.4.2	G-Machine	22
3	Verifikation und Fehlersuche mit Hilfe von Zusicherungen (Assertions)	28
3.1	Testen und Verifizieren von Programmen	28
3.2	Programmspezifikation mit Hilfe von Prädikaten	34
3.3	Die Überprüfung von Zusicherungen (Assertions) in der Programmiersprache Haskell	39
3.3.1	Grundkonzept	39
3.3.2	Beurteilung des Grundkonzepts	48
3.3.3	Prinzipien einer semantikerhaltenden Überprüfung von Zusicherungen (Assertions)	53
3.3.4	Aussagekraft einer semantikerhaltenden Überprüfung von Zusicherungen (Assertions)	63
4	Modifizierung des Haskell-Interpreters Hugs für die Überprüfung von Zusicherungen (Assertions)	67
4.1	Überblick	67
4.2	Zentrale Aspekte und Komponenten der Implementierung des Haskell-Interpreters Hugs	68
4.2.1	Implementierungssprache	68
4.2.2	Programmstruktur	70
4.2.3	Speicherverwaltung	71
4.2.4	Auswertungsmechanismus	73
4.3	Der modifizierte Haskell-Interpreter Hugs - Optionen und Restriktionen bei der Überprüfung von Zusicherungen (Assertions)	76
4.4	Änderungen an der Implementierung des Haskell-Interpreters Hugs im Detail	83
4.4.1	Unterscheidung zwischen Zusicherungsknoten und Programmknoten	83
4.4.2	Die Funktion <code>primAssert</code>	89
4.4.3	Wiederherstellung einer Zusicherung	92
4.4.4	Vorbereitung einer Auswertung	95

4.4.5	Modifikation der Fehlerbehandlung	96
4.4.6	Modifikation der Garbage Collection	97
5	Zusammenfassung	99
A	Die Änderungen an der Implementierung des Haskell-Interpreters	
	Hugs im Überblick	101
A.1	Änderungen in der Datei ~/Hugs98/lib/Prelude.hs	101
A.2	Änderungen in der Datei ~/Hugs98/src/options.h.in	102
A.3	Änderungen in der Datei ~/Hugs98/src/connect.h	103
A.4	Änderungen in der Datei ~/Hugs98/src/hugs.c	104
A.5	Änderungen in der Datei ~/Hugs98/src/builtin.c	108
A.6	Änderungen in der Datei ~/Hugs98/src/machine.c	112
A.7	Änderungen in der Datei ~/Hugs98/src/storage.h	119
A.8	Änderungen in der Datei ~/Hugs98/src/storage.c	122
B	Beispielauswertungen	137
	Literaturverzeichnis	172

1 Einleitung

1.1 Problemstellung

In der Software-Entwicklung spielen qualitätssichernde Maßnahmen, die darauf ausgerichtet sind, Fehler in einem Programm vor dem praktischen Einsatz zu entdecken, eine entscheidende Rolle. Hierfür sind unter anderem folgende Gründe verantwortlich:

1. Die elektronische Datenverarbeitung hat längst in Bereiche wie die Medizin oder die Luftfahrttechnik Einzug gehalten, die hohe Anforderungen an die Korrektheit und die Zuverlässigkeit der Systeme stellen. Eine Fehlfunktion der Software - und ebenso der Hardware - zieht in diesen Bereichen unter Umständen schwerwiegende Konsequenzen nach sich.
2. Je früher erkannt wird, daß ein Programm den gestellten Anforderungen nicht gerecht wird, desto geringer sind die Kosten für die Fehlerbehebung. Auf diesen Aspekt hat [Futschek 1989] schon vor geraumer Zeit hingewiesen: "Fehler, die noch während der Entwicklung entdeckt werden, verursachen nur etwa 5% der Kosten, die entstehen würden, wenn diese Fehler erst beim praktischen Einsatz des Programms gefunden werden. Aus diesen Gründen amortisiert sich ein größerer Aufwand während der Entwicklung, da die Wartungskosten der fertigen Programme geringer werden."

In der Literatur werden eine ganze Reihe von Ideen und Techniken diskutiert, die darauf abzielen, die Fehlerbehebung methodisch zu unterstützen. Das Spektrum reicht von Studien über die Entwicklung von Testdatengeneratoren bis hin zu dem Konzept, "die Architektur der zugrundeliegenden Maschine so zu entwerfen, daß die Test- und Debuggingprozesse unterstützt werden" [Myers 1995].

Beachtung verdient unter anderem der Vorschlag, in Programmiersprachen eine zusätzliche Debugginganweisung einzubauen, die während der Programmausführung sogenannte *Assertions* (*Zusicherungen*) überprüft. Zusicherungen sind Bedingungen, die ein Programm erfüllen muß, um korrekt zu sein.

Dieser Ansatz ist insofern interessant, weil sich in Zusicherungen Elemente einer formalen Spezifikation zum Ausdruck bringen lassen.

Im Rahmen der vorliegenden Arbeit wird untersucht, in welcher Form es gelingt, eine Anweisung für die Überprüfung von Zusicherungen in die nicht-strikte funktionale Programmiersprache Haskell einzubauen.

1.2 Gang der Untersuchung

Der Abschnitt 2 beschreibt wesentliche Eigenschaften funktionaler Programmiersprachen, gibt einen kurzen Überblick über die Programmiersprache Haskell und erörtert die Grundlagen einer nicht-strikten Auswertung.

Der Abschnitt 3 befaßt sich mit der eigentlichen Problemstellung:

- Der Abschnitt 3.1 stellt zunächst die beiden wichtigsten Methoden einander gegenüber, mit denen versucht wird, die Fehlerfreiheit eines Programms festzustellen - das Testen und das Verifizieren. Und zwar mit dem Ziel, die Frage zu be-

antworten, welchen Stellenwert die Überprüfung von Zusicherungen gegenüber der formal exakten Methode der Verifikation einnimmt.

- Der Abschnitt 3.2 legt dar, wie die von einem Programm zu bewältigende Aufgabe mit Hilfe von Prädikaten spezifiziert werden kann, so daß sie sich in Zusicherungen zum Ausdruck bringen läßt. Hierbei wird insbesondere auch auf Unterschiede zwischen imperativen und funktionalen Programmiersprachen eingegangen.
- Im Rahmen des Abschnitts 3.3 wird schließlich ein Konzept für die Überprüfung von Zusicherungen in der Programmiersprache Haskell erarbeitet.

Der Abschnitt 4 beschreibt die Umsetzung des im Rahmen des Abschnitts 3.3 erarbeiteten Konzepts in der Implementierung des Haskell-Interpreters Hugs.

Der Abschnitt 5 faßt die wichtigen Ergebnisse der vorliegenden Arbeit zusammen.

2 Grundlagen

2.1 Funktionale Programmiersprachen

Algorithmen für die Lösung von Problemen werden bei funktionalen Programmiersprachen mit Hilfe von Funktionen formuliert. Der Begriff der Funktion ist dabei im mathematischen Sinne zu verstehen. Er bezeichnet eine Regel, die aus den Werten von Argumenten einen Ergebniswert berechnet. Das heißt, ein funktionales Programm besteht aus einer Menge von Ausdrücken, die Funktionen definieren, ergänzt um einen Ausdruck, der in der durch sie gegebenen Umgebung ausgewertet wird [Hinze 1992].

Von zentraler Bedeutung ist in der funktionalen Programmierung die Forderung nach Seiteneffektfreiheit. Man spricht in diesem Zusammenhang von *Referential Transparency* (*Referentielle Transparenz*).

Eine Programmiersprache heißt *referentiell transparent*, wenn für Ausdrücke, die in dieser Sprache formuliert werden, folgendes gilt [Davie 1992]:

- Ein Ausdruck bezeichnet einen Wert. Die Art und Weise, wie dieser berechnet wird, ist unwichtig.
- Der Wert eines Ausdrucks ist in einem gegebenen Kontext stets gleich. Aus diesem Grund ändert sich der Wert eines Ausdrucks auch nicht, wenn irgendeiner seiner Teilausdrücke durch einen anderen Ausdruck mit dem gleichen Wert ersetzt wird (*Substitutionsprinzip*).

Imperative Programmiersprachen verletzen das Prinzip der referentiellen Transparenz. Bei ihnen bestehen Programme im wesentlichen aus einer Folge von Anweisungen, die sequentiell abgearbeitet werden. Die elementarste Anweisung ist die Wertzuweisung, kurz: Zuweisung. Durch eine Zuweisung kann der Wert einer Variablen, die stets einen Speicherplatz bezeichnet, geändert werden. So erhöht etwa die in der Programmiersprache C formulierte Anweisung

```
x = x + 1;
```

den Wert der Variablen x um 1. Aufgrund dieser Änderung unterscheidet sich die Verwendung der Variablen x vor der Zuweisung von der Verwendung nach der Zuweisung. Durch die Abarbeitung der Anweisungsfolge

```
x = 1;
x = x + 1;
y = x + 1;
```

hat y zum Beispiel den Wert 3. Vertauscht man die beiden letzten Zeilen, hat y dagegen den Wert 2, denn die Teilausdrücke $x + 1$ repräsentieren jeweils unterschiedliche Werte. Das steht im Widerspruch zu dem Prinzip der referentiellen Transparenz und ist der Grund dafür, daß die Auswertungsreihenfolge bei der Abarbeitung eines imperativen Programms nicht ohne weiteres variiert werden kann.

Bei funktionalen Programmiersprachen beeinflussen sich die Auswertungen von Teilausdrücken demgegenüber nicht gegenseitig. Variablen bezeichnen in diesem Fall Werte, keine Speicherplätze, deren Inhalt beliebig geändert werden kann. Das erlaubt die parallele Ausführung von Rechenvorgängen, ohne etwa Synchronisie-

rungsfehler befürchten zu müssen, die im Falle des Vorhandenseins von Zuweisungen aus einer falschen Reihenfolge beim Aktualisieren der Variablen resultieren können [Abelson 1991]. Darüber hinaus bietet der Erhalt der referentiellen Transparenz den Vorteil, daß Programme einer mathematischen Betrachtungsweise einfacher zugänglich sind. Wenn ein Ausdruck nur von den Werten seiner Teilausdrücke abhängt, ist es beispielsweise möglich, modulare Beweistechniken einzusetzen [Davie 1992].

Ein weiteres charakteristisches Merkmal funktionaler Programmiersprachen sind Funktionen höherer Ordnung. Dem Anwender wird die Handhabung von Funktionen ermöglicht, die funktionale Objekte als Argument haben und/oder als Ergebnis zurückgeben. Ein typischer Anwendungsfall ist eine Funktion map , die eine Funktion f auf sämtliche Elemente einer Liste anwendet.

Beispiele für funktionale Programmiersprachen sind Haskell, ML, Miranda und LISP.

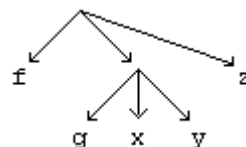
2.2 Auswertungsmechanismen

Die Ausführung eines funktionalen Programms besteht in der Auswertung eines Ausdrucks. Das wirft zunächst die Frage auf, wie dieser dargestellt werden soll.

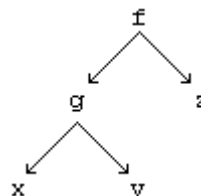
In Betracht kommt unter anderem die Termrepräsentation: Mit dem Begriff der Termrepräsentation wird die Darstellung eines Ausdrucks durch einen markierten geordneten Baum bezeichnet. Sie hat für den Ausdruck

$$f (g x y) z$$

zum Beispiel folgendes Erscheinungsbild:



Es ist auch möglich, eine kompaktere Anordnung zu wählen. Und zwar durch die Verwendung eines Baums, bei dem jeder Knoten eine Operation darstellt und die Nachfolger eines Knotens die dazugehörigen Argumente repräsentieren, so daß nicht ausschließlich die Blätter Informationen tragen:



Auf der Grundlage der Termrepräsentation kann man die Auswertung eines Ausdrucks durch eine Folge von Transformationen / Ersetzungen auf einem Baum beschreiben. Die einzelnen Schritte - im allgemeinen als *Reduktionen* bezeichnet - wer-

den dabei nach genau festgelegten Regeln ausgeführt. Die wichtigste Regel ist die Ersetzungsregel für Funktionsanwendungen (*Definitionsexpansion*):

Wenn eine Funktion f definiert ist als

$$f \ v_1 \ v_2 \ \dots \ v_n = r,$$

wird eine Anwendung dieser Funktion auf n Argumente durch die rechte Seite der Gleichung - den Rumpf der Funktion - ersetzt, wobei für die formalen Parameter die aktuellen Parameter eingesetzt werden.

$$f \ a_1 \ a_2 \ \dots \ a_n \rightarrow r[v_1/a_1, v_2/a_2, \dots, v_n/a_n]$$

Einen Ausdruck, der durch die Anwendung einer Transformationsregel direkt ersetzt werden kann, bezeichnet man als *Reducible Expression*, kurz: *Redex*.

Beispiel:

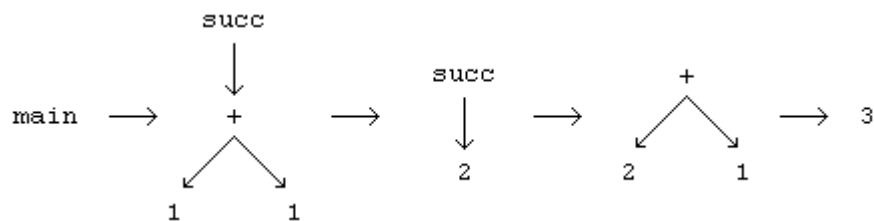
Gegeben sei die folgende Funktionsdefinition:

$$\text{succ } x = x + 1$$

Auszuwerten sei der folgende Ausdruck:

$$\text{main} = \text{succ } (1 + 1)$$

Auswertung:



Dieses Beispiel veranschaulicht, daß die Angabe von Transformationsregeln in so allgemeiner Form wie bei der Definitionsexpansion nicht ausreicht, um einen Auswertungsmechanismus zu spezifizieren. Auch über die Reihenfolge, in der Reduktionen ausgeführt werden, muß Klarheit herrschen, denn die Entscheidung, das Argument der Funktion `succ` vor der Funktionsanwendung zu reduzieren, ist willkürlich. Sie kann auch umgekehrt getroffen werden.

Zwei Strategien, die eine Auswertungsreihenfolge festlegen, haben einen eigenen Namen: Die *Auswertung in applikativer Reihenfolge* und die *Auswertung in normaler Reihenfolge*.

Bei der Auswertung in applikativer Reihenfolge (*Anwendungsordnung, strikte Auswertung*) wird in der Termrepräsentation eines Ausdrucks immer der Redex ersetzt, der in dem Baum am weitesten unten und links positioniert ist. Wenn man sich den Ausdruck als lineare Zeichenfolge vorstellt, kann dieser Mechanismus auch wie folgt formuliert werden: Es wird immer der am weitesten links stehende Redex reduziert, der keinen anderen Redex beinhaltet.

Bei der Auswertung in normaler Reihenfolge (*Normalordnung, nicht-strikte Auswertung*) wird in der Termrepräsentation immer der Redex ersetzt, der in dem Baum am weitesten oben und links positioniert ist. Die alternative Formulierung für eine lineare Zeichenfolge lautet in diesem Fall: Es wird immer der am weitesten links stehende Redex reduziert, der in keinem anderen Redex enthalten ist.

Ein Ausdruck, der keinen Redex beinhaltet, heißt *irreduzibel* oder *in Normalform*.

Was ist das letztendliche Ziel einer Auswertung? Das Ziel kann darin bestehen, einen Ausdruck in seine Normalform zu transformieren [Hinze 1992].

Beispiel:

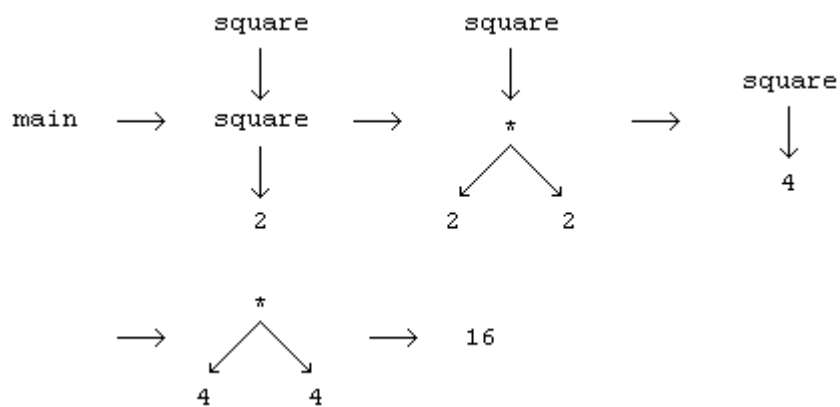
Gegeben sei folgende Funktionsdefinition:

`square x = x * x`

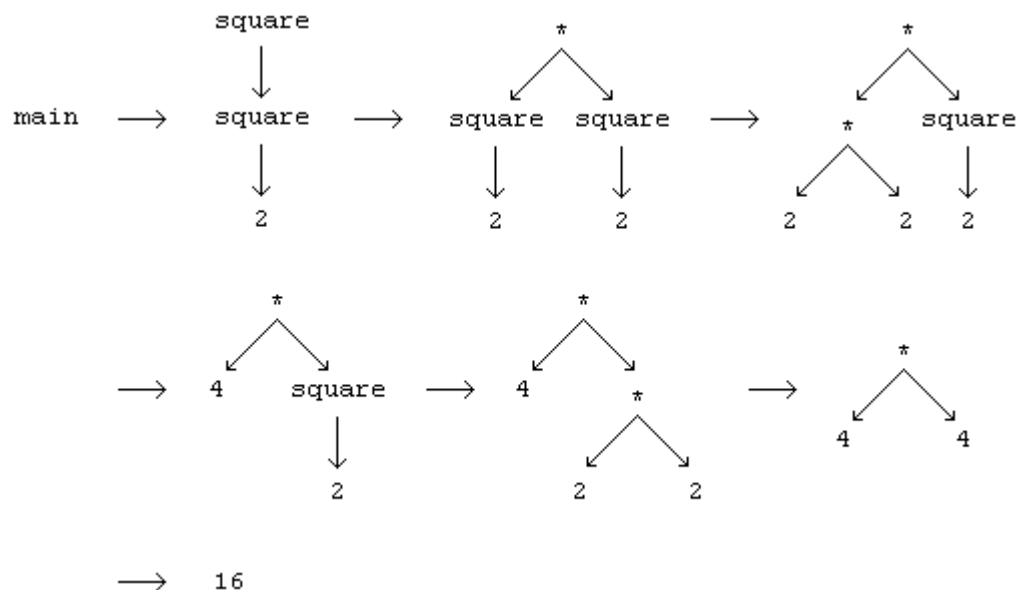
Auszuwerten sei der folgende Ausdruck:

`main = square (square 2)`

Auswertung in applikativer Reihenfolge:

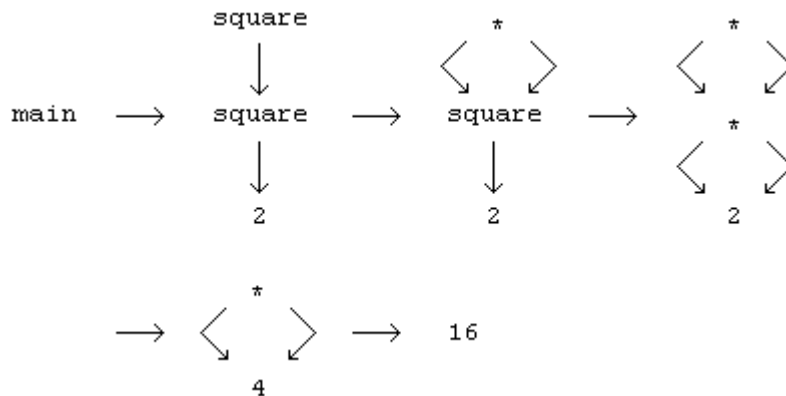


Auswertung in normaler Reihenfolge:



Das Beispiel macht deutlich, daß die Auswertung in normaler Reihenfolge gegenüber der Auswertung in applikativer Reihenfolge im Nachteil ist, wenn ein formaler Parameter im Rumpf einer Funktion mehr als einmal auftritt. Der korrespondierende aktuelle Parameter muß in einem solchen Fall mehrfach berechnet werden, da er unausgewertet in den Rumpf substituiert wird. Hieraus läßt sich allerdings nicht automatisch die Schlußfolgerung ableiten, daß es sich bei der Auswertung in applikativer Reihenfolge um die geeigneteren Reduktionsstrategie handelt. Eine Mehrfachauswertung ist vermeidbar, wenn die Programmausführung nicht auf der Grundlage der Termrepräsentation erfolgt, sondern Graphen für die Darstellung von Ausdrücken verwendet werden. Analog zu dem Begriff der Termrepräsentation spricht man in diesem Zusammenhang von der *Graphrepräsentation* und - bezogen auf den eigentlichen Auswertungsvorgang - von der *Graphreduktion*.

Ein Graph zeichnet sich dadurch aus, daß ein Knoten mehr als eine einlaufende Kante besitzen kann. Das eröffnet die Möglichkeit, mehrere Ausdrücke auf ein und denselben Teilausdruck verweisen zu lassen. Wenn dieser Teilausdruck reduziert wird, erfahren sie alle gemeinsam von dessen Vereinfachung. Für das betrachtete Beispiel ergeben sich hieraus folgende Reduktionsschritte:



Die Auswertung in normaler Reihenfolge angewendet auf eine Graphrepräsentation wird als *Lazy Evaluation* (*Verzögerte Auswertung*) bezeichnet. Deren Ziel ist es letztendlich nicht, einen Ausdruck so lange zu vereinfachen, bis er irreduzibel (in Normalform) ist. Diese Anforderung ist an die strikte Auswertung angepaßt. Man legt vielmehr eine andere Normalform zugrunde, die *Weak Head Normal Form* (*Schwache Kopfnormalform, WHNF*).

Ein Ausdruck ist in schwacher Kopfnormalform, wenn er von der Form

$$f \ x_1 \ x_2 \ \dots \ x_n \ , \ n \geq 0$$

ist und folgendes gilt:

1. f ist eine Variable oder ein Datenobjekt, oder
2. f ist ein Funktionssymbol und der Ausdruck $f \ x_1 \ x_2 \ \dots \ x_m$ stellt für kein $m \leq n$ einen Redex dar.

Die Antwort auf die Frage nach dem Ziel der Auswertung lautet also im Fall der verzögerten Auswertung: Werte einen Ausdruck so lange aus, bis er in schwacher Kopfnormalform ist. Das heißt, ein Ausdruck wird nur reduziert, wenn sein Wert für die

Ermittlung des Gesamtergebnisses unbedingt benötigt wird und dann auch nur einmal.

Dieser Grundsatz umschreibt die Auswertungs-idee aller als nicht-strikt bezeichneten funktionalen Programmiersprachen. Es wäre allerdings unzureichend, solche Programmiersprachen alleine durch die Aussage zu charakterisieren, daß das Argument einer Funktion nur ausgewertet wird, wenn es unbedingt erforderlich ist. Verzögerte Auswertung bedeutet mehr: Falls ein Argument in irgendeiner Form strukturiert ist, handelt es sich also zum Beispiel um ein Tupel oder um eine Liste, werden nur die für das Gesamtergebnis relevanten Teile ausgewertet. Das erlaubt letztendlich sogar den Umgang mit potentiell unendlichen Datenstrukturen. Strikte Programmiersprachen, die in applikativer Reihenfolge auswerten, besitzen diese Eigenschaften nicht.

Es gibt allerdings Ausdrücke, die keine Normalform haben. Das gilt zum Beispiel für den Ausdruck

```
main = infinit 2,
```

sofern die Funktion `infinit` als

```
infinit x = infinit x
```

definiert wird, denn dann terminiert die Auswertung nicht:

```
main
-> infinit 2
-> infinit 2
-> ...
```

Um solchen Ausdrücken dennoch einen Wert zuordnen zu können, wird der Wertebereich um ein Element \perp (ausgesprochen: "bottom") ergänzt. Dieses Element repräsentiert einen undefinierten Wert, der jedem Ausdruck zugewiesen wird, der keine Normalform hat [Peyton Jones 1987]. Fehler, die im Rahmen einer Auswertung die Transformation in eine Normalform verhindern, zum Beispiel eine Division durch die Zahl 0, sind in der Programmiersprache Haskell semantisch gleichbedeutend zu \perp [Peyton Jones 1999].

2.3 Die Programmiersprache Haskell

2.3.1 Überblick

Haskell ist eine nicht-strikte funktionale Programmiersprache. Mit ihrer Entwicklung wurde 1987 als Versuch begonnen, die grundlegenden Konzepte der nicht-strikten funktionalen Programmierung in einer standardisierten Sprache zusammenzufassen. Dieser Standard sollte sowohl den Belangen aus dem Bereich Forschung und Lehre genügen, als auch für praktische Anwendungen geeignet sein [Peyton Jones 1999].

Die aktuelle Version von Haskell wird durch den Haskell 98 Report - [Peyton Jones 1999] - spezifiziert. Einleitend werden darin wichtige Merkmale und Bestandteile der Programmiersprache aufgezählt:

- Funktionen höherer Ordnung
- Statische polymorphe Typisierung

- Benutzerdefinierte algebraische Datentypen
- Pattern Matching
- List Comprehensions
- Ein Modulsystem
- Ein monadisches I/O-System
- Eine Anzahl primitiver Datentypen, z.B. Listen, Arrays, ganze Zahlen, Fließkommazahlen

Der Haskell 98 Report ist über die Haskell Homepage erhältlich:

<http://www.haskell.org/>

Hier findet der Leser auch Informationen über verschiedene Implementierungen von Haskell. Stellvertretend seien an dieser Stelle drei genannt:

- Der Haskell-Interpreter Hugs
- Der Glasgow Haskell Compiler (GHC)
- Der Chalmers' Haskell-B Compiler (HBC)

Es würde den Rahmen dieser Arbeit übersteigen, auf sämtliche der genannten Merkmale und Bestandteile von Haskell einzugehen. Zwei, die besondere Beachtung verdienen, weil sie für deren Untersuchungsgegenstand von Bedeutung sind und vom Grundsatz her auch andere funktionale Programmiersprachen auszeichnen, sollen in den beiden nachfolgenden Abschnitten jedoch näher betrachtet werden: Der Polymorphismus und das Pattern Matching.

2.3.2 Polymorphismus

Zahlreiche moderne Programmiersprachen - funktionale und andere - sind typisierte Programmiersprachen. Man kann dabei zwischen solchen mit einem monomorphen Typsystem und solchen mit einem polymorphen Typsystem unterscheiden. Letztere erlauben es, einem Objekt der Sprache mehr als nur einen Typ zuzuordnen. Das bedeutet für den Anwender ein gewisses Maß an Arbeitserleichterung und hilft ihm, Programme übersichtlicher zu gestalten. Prominentes Beispiel sind listenverarbeitende Funktionen. So muß etwa eine Funktion `length`, die die Länge einer Liste berechnet, nicht für jeden Elementtyp neu programmiert werden. Es genügt eine Definition, die auf eine Liste mit ganzzahligen Elementen ebenso angewendet werden kann wie auf eine Liste, die boolesche Werte beinhaltet, usw.:

```
length :: [Int] -> Int
length :: [Bool] -> Int
...
```

Diese Vielgestaltigkeit wird bei Haskell in der Typdeklaration durch die Verwendung einer *Typvariablen* zum Ausdruck gebracht. Es handelt sich dabei um einen Bezeichner, der mit einem Kleinbuchstaben beginnt - üblicherweise ein Buchstabe vom Anfang des Alphabets. Die Definition der Funktion `length` hat in der durch den Haskell 98 Report spezifizierten Programmbibliothek - [Prelude] - zum Beispiel folgende Gestalt:

```
length :: [a] -> Int
length [] = 0
length (_:l) = 1 + length l
```

Eine Funktion, deren Anwendung nicht auf einen Typ beschränkt ist, wird als *polymorphe Funktion* bezeichnet.

Monomorphe Typsysteme sind weniger flexibel. Jedes Objekt einer Sprache besitzt darin genau einen Typ. Das hat für eine Funktion wie `length` zur Folge, eine Definition für jeden Elementtyp angeben zu müssen, auf den sie anwendbar sein soll. Der Funktionsrumpf ist jeweils identisch; die verschiedenen Definitionen unterscheiden sich lediglich in der Typdeklaration.

Daneben verfügt Haskell mit dem *Overloading (Überladung)* noch über einen zweiten Mechanismus, der es erlaubt, eine Funktion nicht auf einen Typ beschränken zu müssen. Der Unterschied zu dem vorstehend beschriebenen Polymorphismus besteht darin, daß ein und dasselbe Funktionssymbol verschieden definierte Funktionen repräsentiert. Analog zu dem Begriff der polymorphen Funktion spricht man in diesem Zusammenhang von einer *überladenen Funktion*.

Ein Beispiel für eine überladene Funktion ist die Gleichheitsfunktion (`==`), deren Symbol durch den Verzicht auf die Klammerschreibweise auch als Infix-Operator verwendet werden kann: *expression* `==` *expression*. Im Fall des Haskell-Interpreters Hugs steht dieser Operator unter anderem stellvertretend für die Funktionen `primEqInt` und `primEqChar`, die die Gleichheit zweier Objekte vom Typ `Int` bzw. `Char` definieren. Es handelt sich hierbei um eingebaute Funktionen [Hugs-Prelude]:

```
primitive primEqInt :: Int -> Int -> Bool
primitive primEqChar :: Char -> Char -> Bool
```

Wenn ein Ausdruck der Form

```
expression == expression
```

ausgewertet wird, hängt die Beantwortung der Frage, welche der in Betracht kommenden Funktionen für die Überprüfung der Gleichheit maßgebend ist, vom Typ der Argumente ab. Bei ganzen Zahlen ist die Funktion `primEqInt` anzuwenden, bei Argumenten vom Typ `Char` die Funktion `primEqChar`.

Realisiert wird die Überladung mit Hilfe von *Type Classes (Typklassen)*. Eine Typklasse - oder kürzer: *Class (Klasse)* - stellt eine Menge von Typen dar, über denen zumindest eine überladene Funktion definiert ist. Die Gleichheitsfunktion ist Bestandteil der Klasse `Eq`, die folgende Definition / *Deklaration* hat [Prelude]:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimal complete definition:
  --      (==) or (/=)
  x /= y = not (x == y)
  x == y = not (x /= y)
```


Die Deklaration einer Klasse legt in der ersten Zeile zunächst deren Namen fest. In diesem Fall `Eq`. Es folgt eine Liste mit Namen von Funktionen und deren Typen, die man als *Signature (Signatur)* bezeichnet. Sie bestimmt, welche Funktionen für einen Typ definiert werden müssen, um diesen zu einem Bestandteil / einer *Instanz* der Klasse zu machen. Jede Instanz der Klasse `Eq` zeichnet sich demgemäß durch zwei Funktionen aus: Eine Funktion für die Überprüfung der Gleichheit und eine Funktion für die Überprüfung der Ungleichheit, notiert durch den Operator `/=`. Definiert werden solche Funktionen im Rahmen von *Instanzdeklarationen*, die für das betrachtete Beispiel wie folgt lauten [Hugs-Prelude]:

```
instance Eq Int where (==) = primEqInt
instance Eq Char where (==) = primEqChar
```

Auf die Instanzdeklarationen für den Ungleichheitsoperator kann in diesem Fall ausnahmsweise verzichtet werden. Und zwar deshalb, weil die Deklaration der Klasse `Eq` noch einen dritten - nicht zwingend erforderlichen - Bestandteil umfaßt: die sogenannten *Default Definitions*:

```
-- Minimal complete definition:
--      (==) or (/=)
x /= y = not (x == y)
x == y = not (x /= y)
```

Sie definieren die Gleichheit auf der Grundlage der Ungleichheit und umgekehrt. Sofern solche Definitionen in der Deklaration einer Instanz nicht überschrieben werden, behalten sie ihre Gültigkeit. Das heißt, die Ungleichheit wird für die beiden hier betrachteten Instanzen der Klasse `Eq` mit Hilfe der `not`-Operation "automatisch" mittels der Definition der Funktionen `primEqInt` und `primEqChar` hergeleitet.

Weitergehende Informationen über Typklassen kann der Leser [Peyton Jones 1999] und [Thompson 1999] entnehmen.

2.3.3 Pattern Matching

Die rechte Seite einer Funktionsdefinition ist in Haskell nicht auf einen einfachen Ausdruck beschränkt. Sie kann auch aus mehreren Ausdrücken bestehen, die durch sogenannte *Guards (Wächter)* unterschieden werden. Das allgemeine Schema einer solchen Definition - einer *Conditional Equation (Bedingte Gleichung)* - sieht wie folgt aus:

$$\begin{array}{l} f \ v_1 \ v_2 \ \dots \ v_n \\ | \ g_1 = e_1 \\ | \ g_2 = e_2 \\ \dots \\ | \ g_m = e_m \end{array}$$

Erläuterung der Symbole:

- f ist ein Funktionsname
- v_1, v_2, \dots, v_n sind die formalen Parameter der Funktion f
- g_1, g_2, \dots, g_m sind die Wächter
- e_1, e_2, \dots, e_m sind die durch die Wächter unterschiedenen Ausdrücke

Bei den Wächtern handelt es sich um boolesche Ausdrücke. Sie beeinflussen die Auswertung eines Funktionsaufrufs dahingehend, daß das Ergebnis anhand des ersten Ausdrucks berechnet wird, dessen Wächter den Wert True hat. Dem Anwender bietet sich hierdurch die Möglichkeit, eine Fallunterscheidung zu programmieren - eine Fallunterscheidung mit der Eigenschaft, daß die Wächter von oben nach unten abgearbeitet werden, und das Resultat der Auswertung einen Fehler darstellt, falls kein Wächter den Wert True hat.

Beispiel: Die Fakultätsfunktion

```
fak :: Int -> Int
fak x
  | x == 0    = 1
  | otherwise = x * fak (x - 1)    (⇔ | True = x * fak (x - 1))
```

In dieser Definition wird der formale Parameter durch eine Variable repräsentiert. Das ist jedoch nicht zwingend erforderlich. Die Fakultätsfunktion kann man auch so formulieren, daß der Wert, mit dem der aktuelle Parameter bei einer Auswertung innerhalb des ersten Wächters verglichen wird, also die Zahl 0, auf der linken Seite einer Definition als sogenanntes *Pattern (Muster)* Verwendung findet. Hieraus resultiert eine Aufspaltung in zwei Gleichungen:

```
fak :: Int -> Int
fak 0 = 1
fak x = x * fak (x - 1)
```

Bei einem Funktionsaufruf wird zunächst die erste Gleichung bestimmt, deren Muster eine Übereinstimmung mit dem aktuellen Parameter aufweist (zu ihm paßt). Sie liefert das Ergebnis der Applikation. Das heißt, stimmt das Argument mit der Zahl 0 überein, ist das Resultat die Zahl 1. Andernfalls - eine Variable wie *x* ist ein Muster, das zu jedem Ausdruck paßt - wird das Ergebnis mit Hilfe des rekursiven Aufrufs der Fakultätsfunktion ermittelt. Der Auswahlprozeß verläuft dabei analog zu der Vorgehensweise, die für die Wächter maßgebend ist: Man betrachtet die Gleichungen "von oben nach unten". Sofern es sich um eine mehrstellige Funktion handelt, werden zudem auch die Parameter in einer bestimmten Reihenfolge abgearbeitet, nämlich "von links nach rechts".

Die Verwendung von Mustern ist in Haskell sehr gebräuchlich und man bezeichnet die zugrunde liegende Vergleichsoperation als *Pattern Matching (Mustervergleich)*.

Der Mustervergleich gestaltet sich allerdings keineswegs so einfach, wie dies die vorstehende Beschreibung vielleicht vermuten läßt. Wenn der Fakultätsfunktion als Argument zum Beispiel die Zahl 3 übergeben wird, resultiert daraus auch eine Bindung des formalen Parameters an diesen Wert. Außerdem ist es möglich, daß ein Argument reduziert werden muß, um entscheiden zu können, welches Muster paßt. Dann erfolgt die Auswertung jedoch nur soweit wie nötig, denn Haskell ist eine nicht-strikte Programmiersprache.

Will man diesen Mechanismus genauer verstehen, gilt es drei Fragen zu beantworten:

1. Welche Arten von Mustern gibt es?
2. Was genau ist das Ergebnis eines Mustervergleichs?
3. Nach welchen Regeln wird ein Mustervergleich durchgeführt?

Zunächst zu der Beantwortung der ersten Frage. Ein Muster ist im einfachsten Fall eines der folgenden Objekte:

- **Variable**
- **Literal**
Literale sind zum Beispiel Zahlen, Buchstaben und die booleschen Werte True und False.
- **Wildcard: _**
Eine Wildcard dient als Platzhalter für ein Objekt, dessen Eigenschaften in einem bestimmten Kontext nicht von Belang sind.

Beispiel: Die Prelude-Funktion lcm

```
lcm :: (Integral a) => a -> a -> a
lcm _ 0 = 0
lcm 0 _ = 0
lcm x y = abs ((x `quot` gcd x y) * y)
```

Die Funktion lcm berechnet das kleinste gemeinsame Vielfache zweier Zahlen. Wenn eines der Argumente den Wert 0 hat, ist das Ergebnis unabhängig von dem Wert des zweiten Arguments ebenfalls 0. Die Variable, die dieses zweite Argument repräsentiert, kann daher durch eine Wildcard ersetzt werden.

- **Tupel-Pattern**
Hierbei handelt es sich um ein Tupel, dessen Elemente p_1, p_2, \dots, p_n wiederum Muster sind. Die allgemeine Form lautet: (p_1, p_2, \dots, p_n)
- **Listen-Pattern**
Hierbei ist zwischen zwei Mustern zu unterscheiden:
 - $[]$ ist das Muster für eine **leere Liste**.
 - $(x:xs)$ ist das Muster für eine **nicht-leere Liste**.
- **Konstruktor-Pattern**
Hierbei handelt es sich um einen Konstruktor C der Stelligkeit n, angewendet auf n Muster p_1, p_2, \dots, p_n . Die allgemeine Form lautet: $(C p_1 p_2 \dots p_n)$
- **n+k-Pattern**
Hierbei ist n eine Variable und k eine positive ganze Zahl.

In einigen Fällen ist es hilfreich, ein Muster mit einem Namen zu versehen. Eine Funktion, deren Zweck darin besteht, das erste Element einer Liste zu duplizieren, kann beispielsweise definiert werden als:

```
duplicateHead :: [a] -> [a]
duplicateHead (x:xs) = x : x : xs
```

In dieser Definition erscheint der Ausdruck $x : xs$ zweimal: Auf der linken Seite als Muster und auf der rechten Seite als normaler Ausdruck. Das beeinträchtigt die Lesbarkeit. Wünschenswert ist es, den Ausdruck $x : xs$ nur einmal angeben zu müssen. Mit Hilfe eines sogenannten *As-Pattern* kann dies realisiert werden:

```
duplicateHead :: [a] -> [a]
duplicateHead xs@(x:xs) = x : xs'
```

Ein As-Pattern hat allgemein die Form $var@pat$. Hierbei stellt var eine Variable dar und pat ein Muster. Es erlaubt dem Benutzer, var als Name für den Wert zu verwenden, der zu diesem Muster paßt.

Daneben gibt es noch die sogenannten *Lazy Patterns*. Ein Lazy Pattern hat allgemein die Form $\sim pat$, wobei pat wiederum ein Muster ist. Es zeichnet sich dadurch aus, daß ein Vergleich mit ihm unabhängig von pat zunächst einmal erfolgreich verläuft.

Die Regeln für den Mustervergleich unterscheiden zwischen zwei Gruppen von Mustern:

1. *Irrefutable Patterns (Unwiderlegbare Muster)*
2. *Refutable Patterns (Widerlegbare Muster)*

Ein unwiderlegbares Muster ist [Peyton Jones 1999]:

- Eine Variable
- Eine Wildcard
- Ein Muster der Form $N pat$, wobei N ein mittels `newtype` definierter Konstruktor ist und pat ein unwiderlegbares Muster.
- Ein As-Pattern der Form $var@pat$, wobei pat ein unwiderlegbares Muster ist.
- Ein Lazy Pattern der Form $\sim pat$, wobei es keine Rolle spielt, ob pat ein unwiderlegbares Muster ist oder nicht.

Ein widerlegbares Muster ist ein Muster, das kein unwiderlegbares Muster ist.

Nun zur Beantwortung der zweiten Frage. Ein Mustervergleich kann eines von drei möglichen Ergebnissen haben [Peyton Jones 1999]:

1. Der Mustervergleich scheitert.
2. Der Mustervergleich ist erfolgreich; die in dem betrachteten Muster enthaltenen Variablen werden an die korrespondierenden Werte gebunden.
3. Der Mustervergleich divergiert; das Resultat ist \perp .

Somit bleibt als letztes noch zu klären, wie das betreffende Ergebnis ermittelt wird. Ausführliche Informationen hierüber findet man im Haskell 98 Report. Danach schreitet der Mustervergleich von links nach rechts und von außen nach innen unter Beachtung der nachstehenden Regeln voran:

1. Der Vergleich eines Wertes v mit einem unwiderlegbaren Muster var ist stets erfolgreich. Sofern es sich bei dem Muster nicht um eine Wildcard handelt, wird hierdurch var an v gebunden. Die Bindung impliziert allerdings nicht, daß eine Auswertung erfolgt.
Aus operationaler Sicht heißt das, ein "wirklicher" Vergleich findet im Falle eines unwiderlegbaren Musters so lange nicht statt, bis eine darin enthaltene Variable bei der Reduktion Verwendung findet. Erst zu diesem Zeitpunkt wird das gesamte Muster mit dem Wert verglichen. Sollte der Mustervergleich dann scheitern oder divergieren, gilt das auch für die gesamte Berechnung.
2. `con` sei ein mittels `newtype` definierter Konstruktor. Der Vergleich eines Wertes $con v$ mit dem Muster $con pat$ ist dann äquivalent zu dem Vergleich zwischen v und dem Muster pat .
3. Der Vergleich von \perp mit einem widerlegbaren Muster divergiert stets.
4. Regeln für den Vergleich eines Wertes v , der ungleich \perp ist, mit einem widerlegbaren Muster pat :
 - 4.1. pat sei ein Muster, dessen äußere Komponente ein mittels `data` definierter Konstruktor ist. Der Vergleich von v mit pat scheitert dann, wenn v durch einen anderen Konstruktor gebildet wurde. Sofern beide Konstrukturen übereinstimmen, ist das Ergebnis des Mustervergleichs das Ergebnis

des paarweisen Vergleichs der Teilmuster mit den korrespondierenden Bestandteilen von v , und zwar in der Reihenfolge von links nach rechts. Sind diese Vergleiche alle erfolgreich, gilt das auch für den gesamten Mustervergleich; andernfalls verursacht der erste, der scheitert oder divergiert, ein Scheitern oder Divergieren des gesamten Mustervergleichs.

- 4.2. Numerische Literale werden mit Hilfe der überladenen Gleichheitsfunktion verglichen: (`==`).
- 4.3. Der Vergleich von v mit einem $n+k$ -Pattern ist erfolgreich, wenn $v \geq k$ gilt; n wird in diesem Fall an $v - k$ gebunden. Der Mustervergleich scheitert, wenn $v < k$ gilt.
5. Der Vergleich mit einem Konstruktor, bei dem Field Labels verwendet werden, entspricht vom Grundsatz her dem Vergleich mit einem normalen Konstruktor-Pattern.
6. Das Ergebnis, das aus dem Vergleich eines Wertes v mit einem As-Pattern `var@apat` resultiert, ist das Ergebnis, das aus dem Vergleich von v mit `apat` resultiert, wobei `var` an v gebunden wird. Wenn der Vergleich von v mit `apat` scheitert oder divergiert, gilt das auch für den gesamten Mustervergleich.

2.4 Implementierung nicht-strikter funktionaler Programmiersprachen

2.4.1 Auswertung auf einem Graphen zur schwachen Kopfnormalform

Die bisherigen Ausführungen haben gezeigt, welche Vorteile es hat, einen Ausdruck verzögert auszuwerten. In dem nun folgenden Abschnitt soll die Funktionsweise dieses Mechanismus näher untersucht werden. Zu diesem Zweck wird angenommen, daß ein funktionales Programm in einer einfachen *Kernsprache* geschrieben wird. Das heißt: Ein Programm besteht neben dem zu reduzierenden Ausdruck - nachstehend als `main` bezeichnet - lediglich aus einer Reihe von Superkombinator-Definitionen.

Definition:

Ein **Superkombinator** f mit der Stelligkeit n ist ein Ausdruck der Form

$$f \ x_1 \ x_2 \ \dots \ x_n = e,$$

wobei folgendes gilt:

1. f hat keine freien Variablen.
2. Jede Funktionsanwendung in e stellt einen Superkombinator dar.
3. $n \geq 0$, das heißt, ein Superkombinator kann auch die Stelligkeit 0 haben.

Ein Superkombinator mit der Stelligkeit 0 wird zumeist als *Constant Applicative Form* (CAF) bezeichnet. Nachstehende Ausdrücke sind Beispiele für CAF's:

```
1
+ 1 2
+ 1
```

Die Annahme vereinfacht die nachfolgenden Darstellungen. Sie bedeutet aber keineswegs eine Einschränkung dahingehend, daß der zu beschreibende Auswertungs-

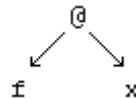
mechanismus nur für Programmiersprachen mit geringer Aussagekraft Gültigkeit besitzt. Wie [Peyton Jones 1992] ausführt, ist es möglich, eine Kernsprache so zu konzipieren, daß Programme, die in einer höheren Programmiersprache wie Miranda geschrieben sind, in diese Sprache übersetzt werden können. Und tatsächlich wird bei der Interpretation bzw. Compilierung nicht-strikter funktionaler Programmiersprachen normalerweise so vorgegangen, daß die höheren Sprachkonstrukte in die Konstrukte einer Kernsprache übersetzt werden. Beispiel: Lambda-Ausdrücke werden mit Hilfe des Lambda-Lifting so transformiert, daß sie keine freien Variablen mehr beinhalten. Anschließend werden sämtliche Lambda-Ausdrücke in gewöhnliche Superkombinator-Definitionen umgewandelt [Peyton Jones 1987].

Als Folge der Beschränkung auf eine Kernsprache handelt es sich bei einem Redex entweder um die Anwendung eines Superkombinators oder einer eingebauten Funktion. Dessen Auswertung beruht aus den in Abschnitt 2.2 genannten Gründen auf der Graphrepräsentation.

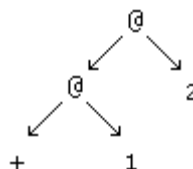
Die Beschreibung dieser Darstellungsform wird dahingehend präzisiert, daß ein Graph, der einen Ausdruck verkörpert, folgende Eigenschaften besitzt:

1. Jeder Knoten hat genau zwei ausgehende Kanten.
2. Die Blätter tragen die Informationen über den darzustellenden Ausdruck (vgl. Abschnitt 2.2).
3. Gemäß der Notation in [Peyton Jones 1987] werden Knoten, die zu einer Funktionsanwendung (Applikation) gehören, mit dem Zeichen @ markiert und in den weiteren Ausführungen als Applikationsknoten bezeichnet.

Die Anwendung einer Funktion f auf ein Argument x wird demgemäß durch folgenden Graphen zum Ausdruck gebracht:



Wie werden Funktionen gehandhabt, die mehr als ein Argument haben? Hierfür bedient man sich der Vorstellung, daß die Anwendung einer Funktion auf ein Argument als Ergebnis eine Funktion liefert, die ebenfalls wiederum auf ein Argument angewendet werden kann, usw. Ein Ausdruck, der die Summe der beiden Zahlen 1 und 2 berechnet, wird zum Beispiel wie folgt notiert:



Dieser Kunstgriff - nach dem Logiker Haskell B. Curry als *Currying* bezeichnet - erlaubt es, eine Funktion immer als einstellige Funktion aufzufassen. Der Ausdruck $(+ 1)$ repräsentiert also eine Funktion, die ihr Argument inkrementiert.

Nun zu dem eigentlichen Auswertungsmechanismus. Die wichtigsten Aspekte seien noch einmal in Erinnerung gerufen:

- Eine Auswertung besteht aus einer Folge von Reduktionen in normaler Reihenfolge.

- Eine Reduktion ersetzt in einem Graphen einen Redex nach einer bestimmten Transformationsregel.
- Eine Auswertung ist beendet, wenn der auszuwertende Ausdruck in WHNF ist.

Die Auswertung auf einem Graphen zur WHNF besteht mithin aus drei Schritten, die so lange wiederholt werden, bis diese Normalform erreicht ist:

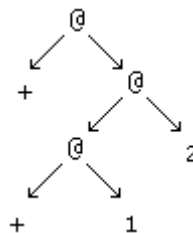
1. Schritt: Wähle den nächsten Redex.
2. Schritt: Reduziere den gewählten Redex.
3. Schritt: Überschreibe die Wurzel des gewählten Redex mit dem Resultat.

Der erste Schritt ist einfach zu realisieren. Bei der Auswertung in normaler Reihenfolge erstreckt sich die nächste Reduktion auf die äußerste Funktionsanwendung. Diese Applikation läßt sich folgendermaßen bestimmen:

1. Beginnend an der Wurzel des Graphen, folgt man so lange dem linken Ast der Applikationsknoten, bis ein Funktionssymbol gefunden wird. Das kann ein Funktionssymbol für einen Superkombinator sein, es kann sich aber auch um ein Funktionssymbol handeln, das eine eingebaute Funktion repräsentiert.

Die Aneinanderreihung der Applikationsknoten, die auf dem Weg zu dem betreffenden Blatt passiert werden, bezeichnet man als *Spine*. Das Entlanglaufen daran selbst wird *Unwinding* genannt.

2. Im Anschluß an das Unwinding wird überprüft, für wieviele Argumente die gefundene Funktion definiert ist. Übersteigt dieser Wert die Anzahl der Applikationsknoten, aus denen sich der Spine zusammensetzt, handelt es sich bei dem gefundenen Ausdruck um eine partielle Funktionsanwendung. Das heißt, es sind nicht genügend Argumente vorhanden. In diesem Fall ist der betrachtete Ausdruck bereits in WHNF und erfordert keine weitere Reduktion. Als Beispiel sei der Ausdruck $+(+ 1 2)$ betrachtet:

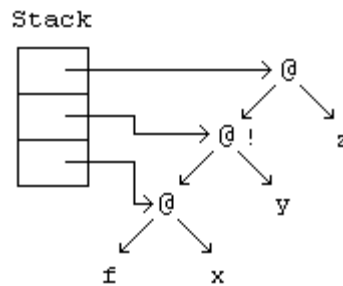


Die Addition ist eine zweistellige Operation, für die hier nur ein Argument zur Verfügung steht. Das ist zwar selbst ein Redex, es wird jedoch im allgemeinen nicht weiter reduziert. Aber: Es gibt auch Evaluatoren, die voraussetzen, daß ein Ausdruck letztendlich zu einem Datenobjekt reduziert. Eine partielle Funktionsanwendung stellt dann einen Fehler dar [Peyton Jones 1987].

Sind für eine n-stellige Funktion hingegen ausreichend Argumente vorhanden, bewegt man sich entlang des Spine um n Applikationsknoten wieder zurück in Richtung Ausgangspunkt - zur Wurzel des Graphen. Der Applikationsknoten, der auf diese Weise erreicht wird, ist die Wurzel der äußersten Funktionsanwendung.

Um das Zurücklaufen zu ermöglichen und Zugriff auf die Argumente der Funktion zu haben, werden die Adressen der Applikationsknoten bereits beim Unwinding auf einem Stack gespeichert, dem sogenannten *Spine Stack*.

Ein Beispiel. Gegeben sei der Ausdruck $f \ x \ y \ z$, wobei f eine zweistellige Funktion repräsentiere. Der Ausdruck $f \ x \ y$ stellt dann die äußerste Funktionsanwendung dar, und der gesamte Ausdruck hat zusammen mit dem Spine Stack folgende Gestalt:



Die Wurzel der äußersten Funktionsanwendung ist mit einem Ausrufezeichen markiert.

Wenn es sich bei der Funktion, die durch das Unwinding gefunden wird, um einen Superkombinator handelt, stellt die äußerste Funktionsanwendung auch gleichzeitig den nächsten zu reduzierenden Redex dar. Schritt 1 ist beendet.

Bei einer eingebauten Funktion kann sich die Situation etwas schwieriger gestalten. Es gibt Fälle, in denen Argumente ausgewertet werden müssen, bevor man die Funktionsanwendung reduzieren kann. Ein Beispiel hierfür ist die eingebaute Addition. So muß in dem Ausdruck

$$+ \ (* \ 1 \ 2) \ 3$$

erst der Wert des inneren Redex $(* \ 1 \ 2)$ berechnet werden, um die Summe bilden zu können. Man sagt, die Funktion $+$ ist strikt in beiden Argumenten.

Definition:

Eine einstellige Funktion f heißt **strikt** in ihrem Argument genau dann, wenn gilt:

$$f \ \perp = \perp$$

Inhaltlich kann diese Definition leicht auf mehrstellige Funktionen übertragen werden. Wenn g zum Beispiel eine Funktion mit drei Argumenten ist, dann heißt g strikt im zweiten Argument genau dann, wenn

$$g \ a \ \perp \ c = \perp$$

gilt, und zwar für alle Werte von a und c [Peyton Jones 1987].

Wenn sich also herausstellt, daß es sich bei dem Ausdruck, der durch das Unwinding gefunden wurde, um die Anwendung einer eingebauten Funktion handelt, die die

Auswertung zumindest eines Arguments erfordert, ist zunächst zu überprüfen, ob das(die) betreffende(n) Argument(e) bereits in WHNF ist(sind). Sollte das nicht der Fall sein, muß der Auswertungsmechanismus rekursiv angewandt werden, um die Transformation(en) in die WHNF zu bewirken, bevor mit der Reduktion der Funktionsanwendung fortgefahren wird.

Die rekursive Auswertung eines Arguments erfordert einen neuen Stack. Dieser kann direkt auf die Spitze des alten Stack aufgesetzt werden, denn

1. der existierende Stack erfährt keine Änderung, solange die Auswertung des Arguments nicht abgeschlossen ist, und
2. der neue Stack kann verworfen werden, wenn die Auswertung des Arguments abgeschlossen ist [Peyton Jones 1987].

Allerdings muß die Tiefe des alten Stack gespeichert werden, um ihn für die eigentliche Reduktion der Funktionsanwendung wiederherstellen zu können. In vielen Implementierungen gibt es zu diesem Zweck einen separaten Stack, der als *Dump* bezeichnet wird. Es ist auch möglich, die Tiefe des alten Stack auf dem Stack selbst zu speichern [Peyton Jones 1987].

Fazit: Im Falle einer eingebauten Funktion stellt die äußerste Funktionsanwendung den nächsten zu reduzierenden Redex dar, falls kein Argument rekursiv ausgewertet werden muß. Ansonsten wird erst ein entsprechender Teilausdruck reduziert.

Jetzt kann die Aufmerksamkeit den Schritten 2 und 3 zugewandt werden.

Sofern der betrachtete Redex zu einem Superkombinator gehört, besteht die Reduktion darin, den Ausdruck durch eine Instanz des Rumpfs zu ersetzen, wobei die formalen Parameter durch die aktuellen Argumente ersetzt werden. Dieser Vorgang zeichnet sich durch verschiedene Merkmale aus [Peyton Jones 1987]:

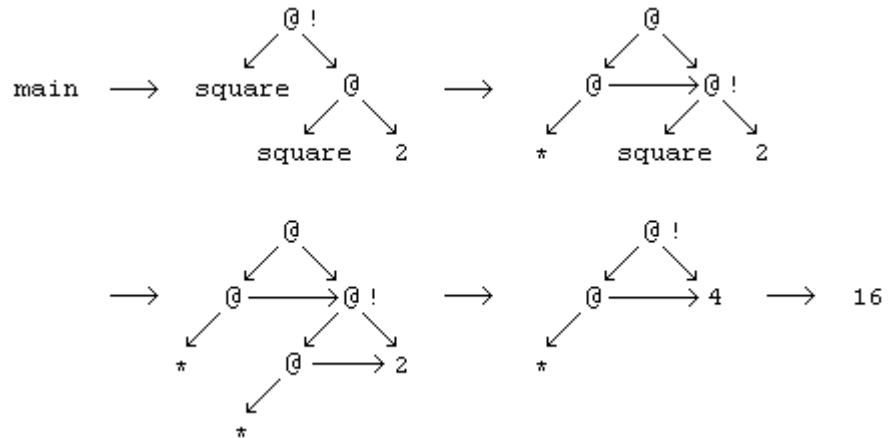
1. Die Ersetzung eines formalen Parameters geschieht nicht dadurch, daß das korrespondierende Argument jedes Mal kopiert wird, wenn der formale Parameter in Erscheinung tritt. Es werden vielmehr Zeiger auf das Argument substituiert. Dies liegt zum einen darin begründet, daß es sich bei dem Argument um einen größeren Ausdruck handeln kann. Das Erstellen mehrerer Kopien desselben Objekts beansprucht dann viel Speicherkapazität. Zum anderen ist es möglich, daß das Argument selber Redexe beinhaltet. In diesem Fall werden die Redexe dupliziert, so daß ein und derselbe Ausdruck mehrfach auszuwerten ist, wenn dessen Wert für die Ergebnisermittlung bekannt sein muß.
2. Ersetzungen werden nicht unmittelbar im Rumpf des betreffenden Superkombinators vorgenommen, sondern in einer Graph-Kopie davon. Damit trägt man dem Umstand Rechnung, daß eine Funktion möglicherweise mehrfach angewendet wird. Der Rumpf dient dann als *Master Template* für die Bildung der Instanz. Er erfährt durch den Kopiervorgang keine Änderung. Das hier zu beschreibende Verfahren wird daher auch als *Template Instantiation (Muster-Instantiierung)* bezeichnet.

Nach Beendigung der Reduktion wird im Schritt 3 die Wurzel des Redex mit dem Ergebnis überschrieben. Sofern der Wurzelknoten mehrere einlaufende Kanten besitzt - man spricht in diesem Zusammenhang von einem sogenannten *Shared Node* -, wird auf diese Weise sichergestellt, daß der betrachtete Ausdruck keine Mehrfachauswertung erfordert.

Wenn man den Spine Stack außer Acht läßt, gestaltet sich die Auswertung des aus dem Abschnitt 2.2 bekannten Ausdrucks

`main = square (square 2)`

zum Beispiel wie folgt:

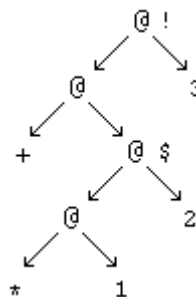


Die Ausrufezeichen signalisieren hier, welcher Knoten die Wurzel des aktuell zu reduzierenden Redex darstellt.

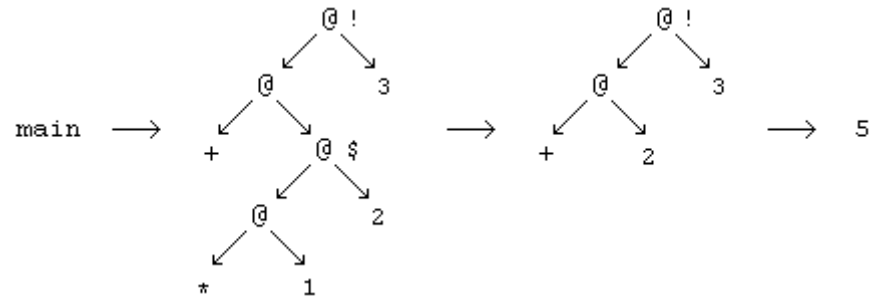
Handelt es sich bei einem Redex nicht um einen Superkombinator, sondern um eine eingebaute Funktion, vollziehen sich die Schritte 2 und 3 in Anlehnung an die bisherigen Ausführungen zu dieser Art von Applikation: Zunächst werden rekursiv alle Argumente ausgewertet, deren Wert für die Ergebnisermittlung bekannt sein muß. Erst danach wird die eigentliche Funktionsanwendung reduziert und die Wurzel des Redex mit dem Resultat überschrieben. Als Beispiel sei noch einmal der Ausdruck

`+ (* 1 2) 3`

betrachtet:



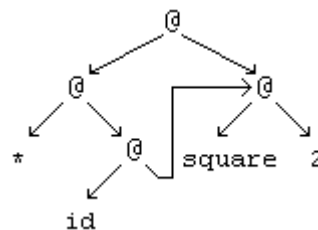
Der mit dem Ausrufezeichen markierte Knoten stellt die Wurzel der äußersten Funktionsanwendung dar. Da es sich hierbei um eine Funktion handelt, die strikt in beiden Argumenten ist, muß das erste Argument rekursiv ausgewertet werden, um es in WHNF zu transformieren - dessen Wurzel ist mit einem Dollarzeichen markiert. Erst wenn dieser Redex durch das Ergebnis der Multiplikation ersetzt wurde, kann die Addition ausgeführt werden:



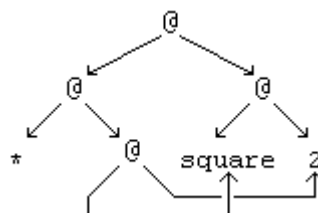
Ziel der verzögerten Auswertung ist es, einen Ausdruck nur einmal zu reduzieren. Das Überschreiben des dazugehörigen Wurzelknotens mit dem Resultat der Auswertung ist dabei von entscheidender Bedeutung. Die bisherige Verfahrensweise reicht jedoch nicht in allen Fällen aus, um redundante Berechnungen zu vermeiden. Das veranschaulicht das nachstehende Programm, welches sich an die Ausführungen in [Peyton Jones 1992] anlehnt:

```
id x = x
f x = * (id x) x
main = f (square 2)
```

Nach der Reduktion des Ausdrucks `f (square 2)` hat der Graph für `main` folgende Gestalt:



Die Funktion `*` ist strikt in beiden Argumenten. Da weder das erste noch das zweite Argument in WHNF ist, müssen beide vor der Ausführung der Multiplikation entsprechend transformiert werden. Geht man davon aus, daß dies zuerst für die Anwendung der Funktion `id` geschieht, handelt es sich bei dem Resultat der nächsten Auswertung um den Ausdruck `square 2`, also um das ursprünglich an die Funktion `f` übergebene Argument, welches sich nicht in WHNF befindet. Folge: Der Ausdruck `square 2` muß zweimal reduziert werden:

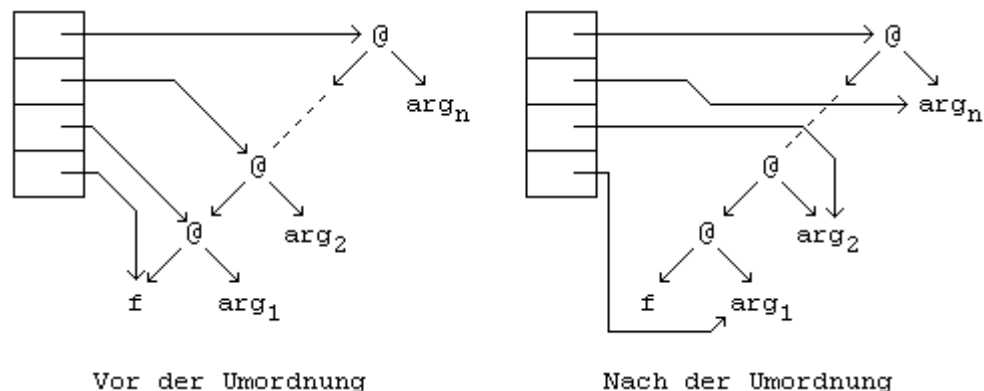


Das steht im Widerspruch zu den Grundsätzen der verzögerten Auswertung. Man kann dieses Problem jedoch durch die Einführung eines neuen Knotentyps vermeiden, der als *Indirection Node* (*Indirektionsknoten*) bezeichnet und mit dem Zeichen `#` markiert wird. Ein Indirektionsknoten fungiert beim Aktualisieren einer Wurzel als Zeiger auf das Resultat der Auswertung. Dadurch wird das Auftreten von Duplikaten einer Applikation vermieden. So auch in dem Beispiel:

Stellt sich durch eine entsprechende Überprüfung heraus, daß für die Anwendung der Funktion genügend Argumente auf dem Stack vorhanden sind, kommt es zu einer Umordnung desselben, und zwar mit dem Ziel, auf die Argumente nicht nur indirekt über die Applikationsknoten, sondern direkt zugreifen zu können. Unter der Voraussetzung, daß sich der Spine aus n Applikationsknoten zusammensetzt, und man die Elemente des Stack relativ zu dessen Spitze adressiert, wobei das oberste Element mit dem Offset 0 angesprochen wird, vollzieht sich diese Umordnung wie folgt:

1. Das $(n - 1)$ -te Stack-Element beinhaltet einen Zeiger auf die Wurzel der äußersten Funktionsanwendung. Dieser Zeiger bleibt erhalten, weil über ihn der Knoten erreichbar ist, der nach Beendigung der Auswertung mit dem Resultat überschrieben wird.
2. Der Zeiger auf das Funktionssymbol an der Spitze des Stack wird durch einen Zeiger auf das erste Argument ersetzt, also durch einen Zeiger auf den rechten Nachfolger des Applikationsknotens, der sich am Ende des Spine befindet.
3. Die verbleibenden $n - 2$ Zeiger, die in den Stack-Elementen 1 bis $n - 2$ gespeichert sind, werden analog zu Punkt 2 durch Zeiger auf die übrigen Argumente ersetzt. Das heißt: Das Stack-Element 1 beinhaltet nach der Umordnung einen Zeiger auf das zweite Argument, das Stack-Element 2 einen Zeiger auf das dritte Argument usw.

Graphisch veranschaulicht, stellt sich der Vorgang wie folgt dar:



Im Anschluß an die Umordnung des Stack erfolgt ein Sprung zur ersten Anweisung der gefundenen Funktion. Die Ausführung des Codes konstruiert, wie erläutert, eine Instanz des Funktionsrumpfs und aktualisiert die Wurzel des Redex. Genau wie bei der in dem vorstehenden Abschnitt beschriebenen Form der Reduktion wiederholt sich dieser Prozeß schließlich so lange, bis der auszuwertende Ausdruck in WHNF ist. Allerdings muß der Zeiger auf den Anfangsgraphen nicht jedes Mal ein Zeiger auf einen Applikationsknoten sein - man erinnere sich an die nullstelligen Superkombinatoren. Dann kann die erste Anweisung auch ohne vorheriges Unwinding und ohne das Umordnen des Stacks angesprungen werden.

Ein Beispiel. Für die Funktion

$$\text{sqrSum } x \ y = (\text{square } x) + (\text{square } y)$$

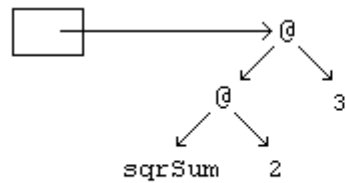
lautet der G-Code:

```
PUSH 1
PUSHFUN square
MKAP
```

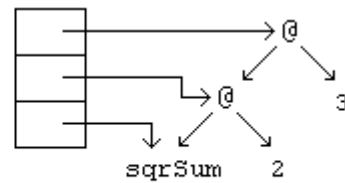
```

PUSH 1
PUSHFUN square
MKAP
PUSHFUN +
MKAP
MKAP
UPDATE 3
POP 2
UNWIND
    
```

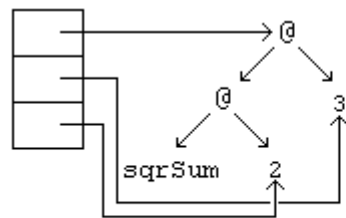
Die nachstehende Abbildung zeigt die Ausführung dieses Codes anhand der Reduktion des Ausdrucks `sqrSum 2 3`.



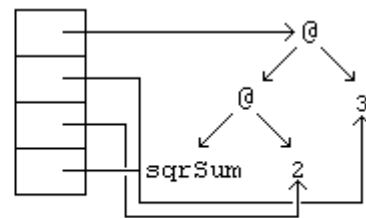
(a)



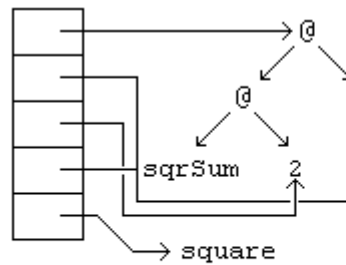
(b) UNWIND



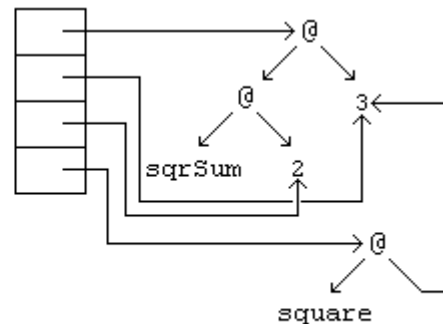
(c) Umordnung



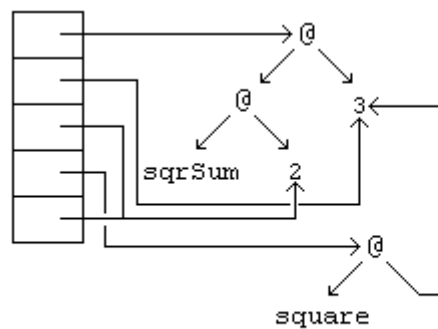
(d) PUSH 1



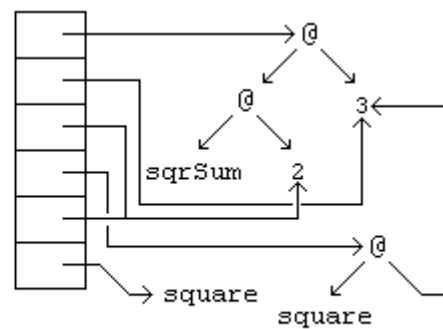
(e) PUSHFUN square



(f) MKAP



(g) PUSH 1



(h) PUSHFUN square

2. Die letzte Teilabbildung beinhaltet nicht mehr den ursprünglichen Redex. Das dient zum einen der Übersichtlichkeit, soll aber insbesondere auch ein Hinweis darauf sein, daß der von den dazugehörigen Applikationsknoten belegte Speicherplatz durch den *Garbage Collector* (die Speicherverwaltung) wieder für eine neue Belegung freigegeben werden kann, falls nicht noch ein anderer Verweis auf sie existiert.

Die in dem Beispiel angewandten Instruktionen haben folgende Bedeutung:

PUSH n	Mit Hilfe dieser Anweisung wird dem Stack eine Kopie des Zeigers, den das $(n + 1)$ -te Stack-Element beinhaltet, als neues Element an der Spitze hinzugefügt.
PUSHFUN f	Mit Hilfe dieser Anweisung wird dem Stack ein Zeiger auf den Superkombinator f als neues Element an der Spitze hinzugefügt.
MKAP	Mit Hilfe dieser Anweisung wird aus den beiden obersten Stack-Elementen ein neuer Applikationsknoten konstruiert. Das Element mit der relativen Adresse 0 bildet dessen linken Nachfolger, das Element mit der relativen Adresse 1 dessen rechten Nachfolger. Beide Elemente werden in diesem Zusammenhang vom Stack genommen und durch einen Zeiger auf den neuen Applikationsknoten ersetzt.
UPDATE n	Mit Hilfe dieser Anweisung wird das $(n + 1)$ -te Stack-Element mit dem obersten Stack-Element überschrieben. Im allgemeinen geschieht das durch die Verwendung eines Indirektionsknotens.
POP n	Mit Hilfe dieser Anweisung werden n Elemente vom Stack entfernt.
UNWIND	Mit Hilfe dieser Anweisung wird die nächste Reduktion initialisiert.

Eingebaute Funktionen werden ebenfalls in eine Instruktionsfolge übersetzt. Dem aus dem vorstehenden Abschnitt bekannten Problem der rekursiven Auswertung von Argumenten begegnet man dabei durch eine zusätzliche G-Code-Anweisung `EVAL` , die das oberste Stack-Element reduziert und das Ergebnis auf dem Stack zurück läßt. Als Beispiel sei der G-Code für die Addition zweier Zahlen betrachtet [Peyton Jones 1987]:

```

PUSH 1
EVAL
PUSH 1
EVAL
ADD
UPDATE 3
POP 2
UNWIND

```

Die ersten vier Anweisungen sorgen dafür, daß die Argumente in ausgewerteter Form auf dem Stack liegen, bevor die eigentliche Addition (`ADD`) ausgeführt wird.

Unter formalen Gesichtspunkten kann man die G-Maschine als Zustandsübergangssystem auffassen, bei dem ein Zustand durch ein 4-Tupel mit den folgenden Komponenten beschrieben wird [Peyton Jones 1987]:

- S - der Stack
- G - der Graph
- C - die noch abzuarbeitende Instruktionssequenz
- D - der Dump

Die G-Code-Instruktionen lassen sich dann als Übergangsregeln beschreiben.

Für weitergehende Informationen über die Implementierung der verzögerten Auswertung - insbesondere auch über die G-Maschine - sei auf die Ausführungen in [Davie 1992], [Peyton Jones 1987] und [Peyton Jones 1992] verwiesen.

3 Verifikation und Fehlersuche mit Hilfe von Zusicherungen (Assertions)

3.1 Testen und Verifizieren von Programmen

Die beiden wichtigsten Methoden, mit denen versucht wird, die Korrektheit eines Programms zu überprüfen, sind das Testen und das Verifizieren. Ein Programm wird als korrekt bezeichnet, sofern es genau die in der Spezifikation definierte Aufgabe erfüllt [Liggesmeyer 1992]. Man unterscheidet dabei zwischen partieller und totaler Korrektheit. Partielle Korrektheit ist gegeben, wenn das Programm die Eigenschaft besitzt, im Falle der Terminierung ein richtiges Ergebnis zu liefern. Totale Korrektheit ist gegeben, wenn das Programm stets terminiert und zugleich partiell korrekt ist.

In der Praxis wird die Frage nach der Softwarequalität mehrheitlich mit Hilfe von Tests beantwortet [Müllerburg 1992]. Das jeweilige Programm wird in einer definierten Umgebung ausgeführt, und die Ausgabedaten werden den aufgrund der Spezifikation zu erwartenden Ergebnissen gegenübergestellt. Liefert der Soll-Ist-Vergleich keine Hinweise auf eine Fehlfunktion, geht man davon aus, daß das Programm fehlerfrei arbeitet. Die Berechtigung dieser Schlußfolgerung hängt allerdings entscheidend von den gewählten Eingabedaten - den Testdaten - ab.

Beispiel:

In der Programmiersprache Haskell sei die folgende Funktionsdefinition für den Quicksort-Algorithmus gegeben:

```
qSort :: Ord a => [a] -> [a]
qSort []      = []
qSort (x:xs) = qSort [y | y<-xs, y<=x] ++
               [x] ++
               qSort [y | y<-xs, y>x]
```

Diese Definition ist fehlerhaft, weil in der letzten Zeile der rekursive Aufruf für die Funktion `qSort` fehlt, so daß die Teilliste der Elemente, die größer als das Pivotelement - in diesem Fall das erste Listenelement - sind, nicht sortiert wird. Die Definition müßte lauten:

```
qSort :: Ord a => [a] -> [a]
qSort []      = []
qSort (x:xs) = qSort [y | y<-xs, y<=x] ++
               [x] ++
               qSort [y | y<-xs, y>x]
```

Aufgrund des Fehlers liefert die Funktion `qSort` nur unter der Voraussetzung ein richtiges Ergebnis, daß die angesprochene Teilliste leer oder bereits sortiert ist. Wählt man für einen Test als Eingabe die Listen `[1, 1, 1, 1]`, `[1, 2, 3, 4]`, `[4, 3, 2, 1]` und `[2, 3, 4, 1]`, bleibt der Fehler mithin unentdeckt:

```

test@hydra:~ > hugs QuickSort
Hugs 98; Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode; Restart with command line option -98 to enable extensions
Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "QuickSort.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
QuickSort.hs
Type :? for help
QuickSort> qSort [1, 1, 1, 1]
[1,1,1,1]
QuickSort> qSort [1, 2, 3, 4]
[1,2,3,4]
QuickSort> qSort [4, 3, 2, 1]
[1,2,3,4]
QuickSort> qSort [2, 3, 4, 1]
[1,2,3,4]
QuickSort>

```

Sicher zielt das Beispiel bewußt darauf ab, daß das Fehlen des rekursiven Funktionsaufrufs für `qSort` unentdeckt bleibt. Es zeigt aber das wesentliche: Auch ein erfolgreich verlaufener Programmtest beweist nicht zwingend die Abwesenheit von Fehlern. Eine zuverlässige Aussage über die Korrektheit ist ausschließlich für die gewählten Eingabedaten möglich. Um durch einen Test die Gewißheit zu erlangen, daß ein Programm in jedem Fall fehlerfrei funktioniert, muß sich der Test auf alle denkbaren Eingabedaten erstrecken. Einen solchen vollständigen Test bezeichnet man als erschöpfend [Myers 1995]. Das Problem ist jedoch: "Die Durchführung eines erschöpfenden Tests ist für reale Programme in der Regel nicht möglich, da oft unendlich viele unterschiedliche Eingaben existieren" [Liggesmeyer 1992]. Und selbst wenn die Eingabedaten in ihrer Gesamtheit eine endliche Menge darstellen, ist die Anzahl der darin enthaltenen Elemente im allgemeinen noch so groß, daß der erschöpfende Test aus wirtschaftlichen und/oder technischen Gründen ausscheidet [Myers 1995].

Entscheidend für die Aussagekraft eines Testvorgangs ist demzufolge die Art und Weise, mit der die Eingaben ermittelt werden. Die gewählte Stichprobe sollte repräsentativ, fehlersensitiv, redundanzarm und ökonomisch sein [Liggesmeyer 1992]. In diesem Zusammenhang bedeutet repräsentativ keineswegs, daß nur typische Eingabedaten Berücksichtigung finden. Das Verhalten eines Programms sollte vielmehr auch in Grenzbereichen beobachtet werden [Hauptmann 1992]. Bei einer Sortierfunktion wie `qSort` ist zum Beispiel auch die Frage von Interesse, wie sie mit der leeren Liste oder einer einelementigen Liste als Argument verfährt.

Das Testen mit intuitiv oder nach dem Zufallsprinzip ausgewählten Daten genügt diesen Anforderungen nur in unzureichendem Maße. In der Literatur werden daher verschiedene systematische Vorgehensweisen für die Ermittlung einer Eingabenstichprobe diskutiert. Zwei seien erwähnt: Das Black-Box-Testen - auch datengetriebenes oder Ein-/Ausgabe-Testen genannt - und das White-Box-Testen - auch logischorientiertes Testen genannt [Myers 1995].

Der Black-Box-Test zeichnet sich dadurch aus, daß Kenntnisse über das interne Verhalten und die Struktur des jeweiligen Programms für seine Durchführung entbehrlich sind. Die Testdaten werden nur anhand der Spezifikation ermittelt [Myers 1995].

Ein Compiler erfordert dann zum Beispiel für den Nachweis der korrekten Übersetzung, daß Testprogramme sämtliche in der Sprachdefinition beschriebenen Sprachkonstrukte abdecken [Hohlfeld 1992].

Bei einem White-Box-Test werden die Testdaten unter Berücksichtigung der Programmstruktur ermittelt. Ziel ist es, möglichst viele Programmteile (Befehle, Zweige oder Pfade) mit den unterschiedlichen Kombinationsmöglichkeiten auszuführen [Spillner 1992]. So wird man für ein `if-then-else`-Konstrukt Testdaten wählen, die sowohl ein Durchlaufen des `then`-Zweigs zur Folge haben, als auch ein Durchlaufen des `else`-Zweigs.

Trotz aller Systematisierungsversuche bleibt aber das Problem bestehen, daß ein Test in der Regel kein geeignetes Mittel ist, um die Korrektheit eines Programms zu beweisen. Ein White-Box-Test, der sämtliche Pfade überdeckt, ist beispielsweise noch kein Garant für das Vorhandensein aller notwendigen Pfade. Dem Problem kann deshalb letztendlich nur durch das Verifizieren von Programmen begegnet werden.

Die Verifikation bedient sich mathematischer Mittel, um die Konsistenz zwischen einem Programm und seiner Spezifikation zu zeigen. Hierfür ist es nicht erforderlich, daß die Spezifikation einen detaillierten Algorithmus oder konkrete Datenstrukturen wiedergibt. Sie muß aber die Aufgabenstellung für das zu entwickelnde Programm präzise beschreiben. Läßt man die Art des Datenbestands außer Acht, kann der Sortiervorgang zum Beispiel wie folgt skizziert werden: Gegeben ist eine Folge von n Objekten a_1, a_2, \dots, a_n . Hierfür ist eine Permutation π der Zahlen von 1 bis n zu finden, und zwar derart, daß die Umordnung der Folge die einzelnen Objekte bezüglich einer Ordnungsrelation \leq in eine aufsteigende Reihenfolge bringt:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$$

Diese eher informelle Beschreibung kann formalisiert werden, indem man die beiden Teilbedingungen für die sortierte Folge mit Hilfe der Prädikatenlogik formuliert:

$$\begin{aligned} & \text{isSorted}(a, a_\pi) \\ \Leftrightarrow & \text{isOrdered}(a_\pi) \wedge \text{isPermutation}(a, a_\pi) \end{aligned}$$

$$\begin{aligned} & \text{isOrdered}(a_\pi) \\ \Leftrightarrow & (\forall i)(1 \leq i \leq n - 1) : (a_{\pi(i)} \leq a_{\pi(i + 1)}) \end{aligned}$$

$$\begin{aligned} & \text{isPermutation}(a, a_\pi) \\ \Leftrightarrow & (\forall x)((\exists i)(1 \leq i \leq n) : a_i = x) : (\#(x, a) = \#(x, a_\pi)) \end{aligned}$$

Hierbei bezeichnet

- n die Anzahl der zu sortierenden Objekte,
- a die zu sortierende Folge,
- a_π die sortierte Folge,
- $\#(x, a)$ die absolute Häufigkeit, mit der das Objekt x in der unsortierten Folge vorkommt, und
- $\#(x, a_\pi)$ die absolute Häufigkeit, mit der das Objekt x in der sortierten Folge vorkommt.

Das Prädikat $\text{isPermutation}(a, a_\pi)$ ist genau dann erfüllt, wenn jedes Objekt, das in der unsortierten Folge mindestens einmal enthalten ist, ebenso oft in der sortierten Folge enthalten ist.

Umordnungsprozesse spielen allerdings bei der Spezifikation eine Sonderrolle, weil sich ihre Eigenschaften häufig aus der Beschreibung des Endzustands erkennen lassen [Endres 1977]. Ohne den Bezug zu einer konkreten Aufgabenstellung kann die Spezifikation eines Programms allgemein als Festlegung einer Funktion

$$f_S: A \rightarrow B$$

charakterisiert werden, wobei A die Menge der in Frage kommenden Eingabedaten und B die Menge der dazugehörigen Ausgabedaten darstellt. Für eine Eingabe $x \in A$ liefert f_S eine korrekte Ausgabe $y = f_S(x) \in B$. Wenn die durch das Programm realisierte Funktion mit f_P bezeichnet wird, ist es somit das Ziel der Verifikation, zu zeigen, daß

$$f_P(x) = f_S(x)$$

für alle $x \in A$ gilt.

Unabhängig von der verwendeten Programmiersprache ist dieser Nachweis im allgemeinen sehr aufwendig und sowohl die Intuition als auch das inhaltliche Verständnis des Programms spielen für ihn eine wichtige Rolle. Als hilfreich erweist sich deshalb der Umstand, daß im Laufe der Zeit verschiedene Methoden entwickelt wurden, die der Verifikation eine formale Grundlage geben. Die vielleicht bekannteste Methode ist der Hoare-Kalkül. "Er erlaubt die Spezifikation und Verifikation von Programmen in imperativen Programmiersprachen des ALGOL-Typs (ALGOL, PASCAL, MODULA-2, ADA)" [Hohlfeld 1992].

Der Hoare-Kalkül beruht darauf, das Verhalten eines Programms mit Hilfe von Prädikaten zu beschreiben. Und zwar wird die Wirkung einer Anweisung oder einer Folge von Anweisungen S durch eine Zeichenkette der Form

$$\{P\} S \{Q\}$$

dargestellt, wobei P und Q Prädikate sind. Diese Notation soll ausdrücken, daß S bezüglich P und Q partiell korrekt ist [Hohlfeld 1992]. Das heißt, wenn vor der Ausführung von S das Prädikat P erfüllt ist, und die Ausführung von S terminiert, dann ist nach der Ausführung von S das Prädikat Q erfüllt. P nennt man *Precondition* (*Vorbedingung*) von S , Q heißt *Postcondition* (*Nachbedingung*) von S [Futschek 1989]. Vor- und Nachbedingung fungieren im Zusammenspiel als Spezifikation. Diese Prädikate gilt es in einem ersten Schritt zu finden und geeignet darzustellen - etwa durch prädikatenlogische Terme. Daß die Programmeigenschaft $\{P\} S \{Q\}$ dann auch tatsächlich gilt, wird formal mit Hilfe verschiedener Verifikationsregeln gezeigt, wodurch der Hoare-Kalkül einen Bezug zur mathematischen Logik erlangt.

Bei funktionalen Programmiersprachen kann sich die Beweisführung einfacher gestalten. Aufgrund der referentiellen Transparenz sind funktionale Programme einer mathematischen Argumentation auch ohne die Zuhilfenahme einer auf der Formulie-

von Prädikaten beruhenden Verhaltensbeschreibung zugänglich. Ein Beispiel hierfür ist die Fakultät.

Für eine natürliche Zahl n sei die Fakultät unter Verwendung der abkürzenden Schreibweise " $n!$ " wie folgt definiert:

$$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n * (n - 1) * \dots * 1, & \text{falls } n > 0 \end{cases}$$

Das korrespondierende Haskell-Programm lautet:

```
fak :: Int -> Int
fak 0 = 1
fak n = n * fak (n - 1)
```

Die Korrektheit dieses Programms kann unmittelbar mit Hilfe des Beweisprinzips der vollständigen Induktion gezeigt werden:

Behauptung:

$$\text{fak } n = n! = n * (n - 1) * \dots * 1 \text{ für alle } n \geq 0$$

Beweis durch vollständige Induktion:

i) Induktionsanfang $n = 0$

$$\text{fak } 0 = 1 = 0!$$

ii) Induktionsschritt $n \rightarrow n + 1$

Sei

$$\text{fak } n = n! = n * (n - 1) * \dots * 1$$

als Induktionsvoraussetzung bereits bewiesen. Es ist zu zeigen, daß

$$\text{fak } (n + 1) = (n + 1)!$$

gilt. Dies sieht man so:

$$\begin{aligned} \text{fak } (n + 1) &= (n + 1) * \text{fak } (n) \\ &= (n + 1) * n * (n - 1) * \dots * 1 \\ &= (n + 1)! \end{aligned} \quad \blacksquare$$

Die Einfachheit des Beweises beruht vor allem auf der Tatsache, daß sich die Implementierung der Funktion im Grunde genommen nicht von der Formulierung der Spezifikation unterscheidet. Ein Umstand, der in der funktionalen Programmierung in vielen Fällen beobachtet werden kann [Davie 1992].

Als Konsequenz der bisherigen Ausführungen stellt sich natürlich die Frage, ob man auf das Testen als Qualitätssicherungsmaßnahme nicht besser verzichten sollte, und zwar unabhängig von der Art der Programmiersprache. Diese Frage ist zu verneinen. In der Regel wird es sinnvoll sein, Verifikationstechniken und Testverfahren gleich-

zeitig einzusetzen. Hierfür gibt es verschiedene Gründe, von denen zwei genannt seien:

1. Eine Beweisführung kann fehlerhaft sein. Es ist daher durchaus von Vorteil, sie durch systematische Tests zu ergänzen, denn auf diese Weise verringert sich die Fehlerwahrscheinlichkeit.
2. Auch ein korrektes Programm kann ein fehlerhaftes Verhalten zeigen. Als mögliche Ursachen ist in diesem Zusammenhang an Fehler in Übersetzern, Betriebssystemen und Hardwarekomponenten zu denken. Ein Programmtest deckt dann zwar nicht nur reine Programmierfehler auf, er vermittelt aber einen Eindruck davon, wie sich die entwickelte Software unter realen Umgebungsbedingungen verhält.

Obwohl man das Testen keinesfalls überbewerten darf, besteht also die Notwendigkeit, gezielt nach Ansätzen zu suchen, die es systematisieren, automatisieren und/oder ergänzen können. Wie in der Einleitung bereits erwähnt wurde, verdient in diesem Zusammenhang der Vorschlag Beachtung, in Programmiersprachen eine zusätzliche Debugginganweisung einzubauen, die während der Programmausführung Zusicherungen überprüft.

Präzise formuliert, besteht die Aufgabe der Debugginganweisung darin, die Programmausführung in Abhängigkeit von dem Wert eines booleschen Ausdrucks entweder nach der Abarbeitung einer Fehlerroutine abubrechen (das geschieht, wenn der boolesche Ausdruck den Wert `False` hat) oder fortzusetzen (das geschieht, wenn der boolesche Ausdruck den Wert `True` hat). In diese Richtung zielt zum Beispiel der `assert`-Makro in der Programmiersprache C. Er erlaubt es dem Anwender, einem Programm eine Zeile der Form

```
assert(expression);
```

hinzuzufügen. Der Ausdruck innerhalb des Klammerpaars repräsentiert einen ganzzahligen Wert. Zeigt die Ausführung von `assert(expression)`, daß es sich hierbei um die Zahl 0 handelt, gibt der `assert`-Makro eine Fehlermeldung aus, die den Dateinamen der Programmquelle und die Zeilennummer des Funktionsaufrufs umfaßt. Anschließend wird das Programm sofort beendet. Hat *expression* dagegen einen von 0 verschiedenen Wert, läuft das Programm normal weiter.

Die in diesem Kontext auftretenden Prädikate sind es, die als *Zusicherungen (Assertions)* bezeichnet werden. An geeigneter Stelle im Programm eingefügt, können sie beim Lokalisieren eines Fehlers sehr hilfreich sein.

Ein Werkzeug, das während der Programmausführung das Eintreffen bestimmter, vorher definierter Bedingungen überprüft, bietet jedoch noch einen anderen wichtigen Vorteil: Beschreibt man die Wirkung eines Programms mit Hilfe von Prädikaten, kann die Spezifikation in den Programmtext integriert werden. Bei einem Test wird dann automatisch festgestellt, ob das Programm auf die gewählten Eingabedaten in der vorgesehenen Weise reagiert. Unter Umständen führt das im Ergebnis sogar zu der Entdeckung von Fehlern, die sich in den Ausgabedaten nicht dokumentieren - ein Beispiel wird dies an anderer Stelle veranschaulichen. Insofern gelingt es durch die Überprüfung von Zusicherungen, dem Testen eine größere Aussagekraft zu verleihen. Das geschieht zwar nicht in dem Sinne, daß der Nachweis der partiellen oder gar der totalen Korrektheit für ein einzelnes Programm automatisch durchgeführt

wird, aber doch immerhin in dem Sinne, daß die Konsistenz zwischen dem Programm und seiner Spezifikation für die Eingabestichprobe sichergestellt ist.

Auch in der objektorientierten Programmiersprache Eiffel besteht die Möglichkeit, Zusicherungen in Programme einzubinden. Für eine Routine kann zum Beispiel eine *Precondition* und eine *Postcondition* angegeben werden [Meyer 1992]:

- Die Precondition - erkennbar an dem Schlüsselwort *require* - bringt Bedingungen zum Ausdruck, die beim Aufruf der Routine erfüllt sein müssen.
- Die Postcondition - erkennbar an dem Schlüsselwort *ensure* - bringt Bedingungen zum Ausdruck, die nach Beendigung des Aufrufs der Routine erfüllt sein müssen.

Im Rahmen der weiteren Ausführungen soll untersucht werden, inwieweit es sinnvoll und in welcher Form es möglich ist, eine Anweisung für die Überprüfung von Zusicherungen - im folgenden als *assert*-Anweisung bezeichnet - in die nicht-strikte funktionale Programmiersprache Haskell zu integrieren. Das erfordert zunächst die Beschäftigung mit der Frage, wie sich die Wirkung eines Programms mit Hilfe von Prädikaten beschreiben läßt. Die Antwort hierauf gibt der nachstehende Abschnitt.

3.2 Programmspezifikation mit Hilfe von Prädikaten

Eine Spezifikation formuliert präzise die von einem Programm zu bewältigende Aufgabe. Wenn sich diese Aufgabe im Hinblick auf die automatische Überprüfung von Zusicherungen auch im Programmtext widerspiegeln soll, muß man hierfür Prädikate als Ausdrucksmittel verwenden können. Wie aber spezifiziert man ein Programm mit Hilfe von Prädikaten? Bei imperativen Programmiersprachen wird zu diesem Zweck der Umstand ausgenutzt, daß Eingabedaten in Variablen gespeichert und weiterverarbeitet werden.

Imperative Programme verwenden nur eine endliche Anzahl von Variablen. Sofern die Möglichkeit, daß undefinierte Variablenwerte vorkommen, einmal außer Acht gelassen wird, hat jede dieser Variablen stets einen Wert aus einem dazugehörigen Wertebereich. In der Programmiersprache C ergibt sich der Wertebereich aus der Zuordnung von Datentypen. So legt der Datentyp `int` zum Beispiel fest, daß die betreffende Variable ganzzahlige Werte annehmen kann; die Wertobergrenze und die Wertuntergrenze hängen von der jeweiligen Implementierung ab.

Der Fortgang einer Berechnung dokumentiert sich in der Veränderung der Variablenwerte. Man spricht in diesem Zusammenhang davon, daß die Wirkung einer einzelnen Anweisung oder auch einer Folge von Anweisungen in der durch sie hervorgerufenen Änderung des Programmzustands - kurz: Zustand - zum Ausdruck kommt, und definiert dementsprechend: "Ein Programmzustand ist eine bestimmte Belegung der Variablen mit Werten" [Futschek 1989]. Aus dieser Sichtweise heraus befindet sich ein imperatives Programm vor Ablauf in einem Anfangszustand, wird durch die Abarbeitung der Anweisungen Schritt für Schritt in Folgezustände überführt und befindet sich nach Ablauf in einem Endzustand.

Beispiel:

Ein C-Programm befinde sich in einem Zustand z , in dem die Variablen x und y den Wert 1 bzw. 2 haben:

$$z: (x, y) = (1, 2)$$

Durch die Abarbeitung der Anweisungen

```
t = x;
x = y;
y = t;
```

wird ein neuer Zustand z' erreicht, in dem die Variablen x und y den Wert 2 bzw. 1 haben:

$$z': (x, y) = (2, 1)$$

In Übereinstimmung mit der inhaltlichen Bedeutung des Begriffs sind Aussagen über einen Programmzustand logische Aussagen über den Wert von Variablen. Sie sind entweder wahr oder falsch. Deshalb ist es möglich, diese Aussagen in der Sprache der Prädikatenlogik zu formulieren. Für das betrachtete Beispiel lauten die Terme:

$$z : x = 1 \wedge y = 2$$

$$z' : x = 2 \wedge y = 1$$

Der Übergang vom Programmzustand z in den Programmzustand z' kann folglich auch mit der aus dem vorstehenden Abschnitt bekannten Notation der Form

$$\{P\} S \{Q\}$$

dokumentiert werden:

$$P: \{x = 1 \wedge y = 2\}$$

```
t = x;
x = y;
y = t;
```

$$Q: \{x = 2 \wedge y = 1\}$$

Aus der isolierten Betrachtung der Vorbedingung und der Nachbedingung kann man in diesem Fall noch nicht erkennen, welchem Zweck die Anweisungsfolge dient, nämlich dem Austausch der Werte zweier Variablen x und y . Wenn zu Beginn das Prädikat

$$x = 1 \wedge y = 2$$

erfüllt ist, und die Anweisungsfolge

```
x = x + 1;
y = y - 1;
```

abgearbeitet wird, ist das Prädikat

$$x = 2 \wedge y = 1$$

am Ende zum Beispiel ebenfalls erfüllt. Sie sind für eine Spezifikation daher ungeeignet. Dieser Sachverhalt ändert sich allerdings, wenn man die beiden Prädikate durch die Verwendung von zwei Hilfsvariablen x und y , die in dem Programm nicht in Erscheinung treten, allgemeiner formuliert:

$$\begin{aligned}x &= X \wedge y = Y \\x &= Y \wedge y = X\end{aligned}$$

Da die Werte der Hilfsvariablen durch das Programm keine Veränderung erfahren können, gelingt es mit ihrer Hilfe, den Zusammenhang zwischen den Anfangs- und Endwerten der Programmvariablen zu beschreiben. Abstrahiert man von der Anweisungsfolge, führt das zu dem Ergebnis, daß die Zeichenkette

$$\{x = X \wedge y = Y\} S \{x = Y \wedge y = X\}$$

als Spezifikation für die Programmentwicklung wie folgt interpretiert werden kann: Finde ein Programmstück S , so daß jedes Mal, wenn vor der Ausführung von S die Variablen x und y den Wert X bzw. Y haben, die Ausführung von S terminiert, und nach der Terminierung die Variablen x und y den Wert Y bzw. X haben, und zwar für beliebige Werte X und Y .

Eine Spezifikation ist jedoch im Idealfall von der Implementierung unabhängig. Insofern ist es nicht erforderlich, die gewünschte Wirkung einzelner Programmfragmente festzulegen. Es reicht für die Spezifikation eines imperativen Programms vielmehr aus, durch die Formulierung geeigneter Vor- und Nachbedingungen die zulässigen Belegungen der Variablen mit Werten vor dem Ablauf des Programms und die gewünschten Belegungen der Variablen mit Werten nach dem Ablauf des Programms zu definieren [Futschek 1989].

Auf der Grundlage solcher Prädikate ist es dann relativ einfach, eine Spezifikation für die automatische Überprüfung von Zusicherungen in den Programmtext einzubinden: Die Prädikate werden im Rahmen der zur Verfügung stehenden Möglichkeiten mit Hilfe von Vergleichsoperatoren ($<$, \leq , $=$, \geq , $>$), logischen Operatoren (and , or , not) usw. in äquivalente programmiersprachliche Konstrukte transformiert und via `assert`-Anweisung als Bedingungen, deren Eintreffen zu überprüfen ist, dem Programmtext an geeigneter Stelle hinzugefügt.

Funktionale Programme können ebenfalls mit Hilfe von Prädikaten spezifiziert werden. Es gibt allerdings Unterschiede zu den imperativen Programmiersprachen, weil Variablen darin keine Speicherplätze bezeichnen, deren Inhalt beliebig geändert werden kann (vgl. Abschnitt 2.2). An einem Beispiel läßt sich das veranschaulichen: Es sei ein Programm `MAX` zu entwickeln, mit dem das Maximum zweier ganzer Zahlen m und n bestimmt werden kann. Das heißt, die Spezifikation muß die Wirkungsweise einer Funktion beschreiben, die folgendermaßen definiert ist:

$$\max(m, n) = \begin{cases} m, & \text{falls } m \geq n \\ n, & \text{falls } m < n \end{cases}$$

Im Fall der imperativen Programmiersprachen taucht in diesem Zusammenhang eine Frage auf, die bei dem eingangs betrachteten Austausch von Variablenwerten keine

Rolle spielt: Wie wird der aus den Argumenten einer Funktion zu berechnende Ergebniswert dargestellt? Hierfür gibt es verschiedene Möglichkeiten:

1. Der Ergebniswert wird einer besonderen Ergebnisvariablen zugewiesen.
2. Der Ergebniswert wird in einer der Variablen zum Ausdruck gebracht, die die Argumente der Funktion beinhalten. In dem hier betrachteten Beispiel sind das zwei Variablen, die mit x und y bezeichnet seien.

Von der Art, wie der Ergebniswert dargestellt wird, hängt die Formulierung der Nachbedingung ab. Entscheidet man sich zum Beispiel für die erste Möglichkeit, wobei

- z die Ergebnisvariable bezeichne, und
- X und Y Hilfsvariablen seien, die die Anfangswerte der Programmvariablen x und y repräsentieren,

lautet die Nachbedingung:

$$(z = X \wedge X \geq Y) \vee (z = Y \wedge X < Y)$$

Das Prädikat sagt nichts über die Eigenschaften von x und y aus. In dieser Form erlaubt es dem Programm, die Werte der beiden Variablen zu verändern. Sollen sie unverändert bleiben, muß die Nachbedingung um das Prädikat

$$x = X \wedge y = Y$$

ergänzt werden, welches dann gleichzeitig die Vorbedingung darstellt. Diese Handhabung der Variablenwerte vorausgesetzt, hat die Spezifikation für das Programm MAX "in der imperativen Welt" also insgesamt folgendes Erscheinungsbild:

$$P: \{x = X \wedge y = Y\}$$

MAX

$$Q: \{((z = X \wedge X \geq Y) \vee (z = Y \wedge X < Y)) \wedge (x = X \wedge y = Y)\}$$

Noch kurz eine Anmerkung zu der Vorbedingung: Sofern man davon ausgehen kann, daß die inhaltliche Bedeutung der Hilfsvariablen durch die Spezifikation auch ohne den darin explizit zum Ausdruck gebrachten Bezug zu den Programmvariablen ersichtlich wird, ist die Angabe des Prädikats in der Form

$$x = X \wedge y = Y$$

etwas umständlich; es handelt sich um eine Invariante. Das Prädikat `true` ist in diesem Fall als Vorbedingung ausreichend. Es signalisiert, daß die Werte der Argumente keinen besonderen Einschränkungen unterliegen [Futschek 1989]. Grundsätzlich gilt für eine Spezifikation: Als Vorbedingung wird immer die sogenannte *Weakest Precondition* (*Schwächste Vorbedingung*) angegeben. Die schwächste Vorbedingung eines Programms S bezüglich einer Nachbedingung Q ist das Prädikat, das die Menge aller Anfangszustände beschreibt, die nach der Ausführung von S zur Erfüllung der Bedingung Q führen [Backhouse 1989].

In der funktionalen Programmierung kann man Prädikate für die Beschreibung einer Aufgabenstellung zum Teil einfacher finden. Variablen werden dort in einer Weise verwendet, wie sie aus der Mathematik vertraut ist. Die Frage, ob und wie die Werte aktueller Parameter durch den Programmablauf verändert werden dürfen oder verändert werden müssen, ist für die Formulierung einer Spezifikation daher nicht von

Belang. Das hat den Vorteil, auf Hilfsvariablen verzichten zu können, die den Wert einer Variablen vor dem Ablauf des Programms repräsentieren. Darüber hinaus ist es nicht erforderlich, sich über die Darstellung des Ergebniswerts einer Funktion in der Form Gedanken zu machen, wie dies bei den imperativen Programmiersprachen der Fall ist. Ohne das Vorhandensein einer Anweisung für Wertzuweisungen im Sinne der Ausführungen des Abschnitts 2.2 reicht es aus, Zusammenhänge zwischen dem Resultat einer Berechnung und den zugrunde liegenden Argumenten mit Hilfe der normalen Funktionsschreibweise darzustellen, also zum Beispiel durch einen Ausdruck der Form:

$$f(x_1, x_2, \dots, x_n) = \text{expression}$$

Sofern die aktuellen Parameter keinen besonderen Beschränkungen unterliegen, kann das Programm MAX deshalb mit Hilfe des folgenden Prädikats spezifiziert werden:

$$(\max(m, n) = m \wedge m \geq n) \vee (\max(m, n) = n \wedge m < n)$$

Eine abschließende Bemerkung, die an die Ausführungen im Zusammenhang mit der Fakultät anknüpft (vgl. Abschnitt 3.1): Das vorstehende Beispiel soll nicht den Eindruck vermitteln, für die Spezifikation eines funktionalen Programms seien bevorzugt Prädikate zu verwenden. Es geht an dieser Stelle lediglich um die Feststellung, daß es im Hinblick auf die Formulierung boolescher Ausdrücke, die bei der Überprüfung des Programmablaufs durch assert-Anweisungen eine Rolle spielen, grundsätzlich möglich ist, Verhaltensweisen funktionaler Programme mit Hilfe von Prädikaten auszudrücken.

Für eine zielgerichtete Programmentwicklung ist zum Beispiel die Funktionsdefinition

$$\max(m, n) = \begin{cases} m, & \text{falls } m \geq n \\ n, & \text{falls } m < n \end{cases}$$

besser geeignet als das Prädikat

$$(\max(m, n) = m \wedge m \geq n) \vee (\max(m, n) = n \wedge m < n)$$

Und zwar aufgrund der Tatsache, daß sie in ihrer Gestalt der entsprechenden programmiersprachlichen Formulierung sehr ähnlich ist:

- Die Funktionsdefinition hat die Form einer Gleichung.
- Auf der linken Seite der Gleichung steht der Funktionsname, gefolgt von einem Klammersymbol, der signalisiert, daß der Ergebniswert aus zwei Argumenten berechnet wird.
- Auf der rechten Seite der Gleichung stehen die für die notwendige Fallunterscheidung relevanten Bedingungen.

Aus diesen Merkmalen kann unmittelbar die folgende Haskell-Definition hergeleitet werden:

```
max :: Int -> Int -> Int
max x y = if x >= y
          then x
          else y
```

Oder alternativ:

```
max :: Int -> Int -> Int
max x y
  | x >= y    = x
  | otherwise = y
```

Es gibt aber natürlich auch Fälle, in denen sich Prädikate besonders eignen, um das gewünschte Verhalten eines funktionalen Programms zu beschreiben: Wenn eine Folge von n Objekten aufsteigend sortiert werden soll, muß die sortierte Folge unter anderem eine Eigenschaft aufweisen, die mit der unsortierten Folge nicht unmittelbar im Zusammenhang steht. Gemäß den Ausführungen in dem vorstehenden Abschnitt muß die sortierte Folge $a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}$ für eine Ordnungsrelation \leq nämlich der Bedingung

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$$

genügen. Und dieser Sachverhalt kann am einfachsten durch ein Prädikat zum Ausdruck gebracht werden (vgl. Abschnitt 3.1).

3.3 Die Überprüfung von Zusicherungen (Assertions) in der Programmiersprache Haskell

3.3.1 Grundkonzept

Haskell ist eine statisch stark getypte Programmiersprache. Das heißt, jeder Ausdruck muß einen Typ haben, und die Einhaltung der Typregeln wird zur Übersetzungszeit überprüft.

Das Typsystem trägt in erheblichem Maße dazu bei, die Betriebssicherheit von Programmen zu erhöhen, weil Fehler, die eine Verletzung der Typregeln zur Folge haben, erkannt und lokalisiert werden, bevor ein Programm ausgeführt wird. Die möglichen Ursachen solcher Fehler sind vielfältig: Eine n -stellige Funktion wird auf mehr als n Argumente angewendet, eine Typdeklaration wird fehlerhaft formuliert, eine Funktion wird auf ein Argument des falschen Typs angewendet usw.

Beispiel:

Eine Funktion `doubleSquare`, die für ein beliebiges $x \in \mathbb{Z}$ den Wert

$$f(x) = 2 * x^2$$

berechnen soll, sei in einem Skript `DoubleSquare.hs` wie folgt definiert worden:

```
doubleSquare :: Int -> Int
doubleSquare x = double square

double :: Int -> Int
double x = 2 * x

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

Bei näherer Betrachtung stellt man fest, daß auf der rechten Seite der Gleichung

```
doubleSquare x = double square
```

das Argument für die Funktion `square` fehlt. Sie müßte lauten:

```
doubleSquare x = double (square x)
```

Das Typsystem erkennt diesen Fehler, denn die Funktion `double` erwartet als Argument einen Ausdruck, der zu einer ganzen Zahl reduziert, nicht einen Bezeichner für eine einstellige Funktion vom Typ `Int -> Int`. Konsequenz: Das betreffende Programm wird abgewiesen:

```

xterm
test@hydrat~$ hugs DoubleSquare
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode; Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "DoubleSquare.hs":
Type checking
ERROR "DoubleSquare.hs" (line 4): Type error in application
*** Expression : double square
*** Term      : square
*** Type      : Int -> Int
*** Does not match : Int

Prelude>
    
```

Ein erfolgreich übersetztes Programm ist jedoch trotz des Typsystems nicht notwendigerweise konsistent zu seiner Spezifikation. Das belegt die im Abschnitt 3.1 betrachtete Funktionsdefinition für den Quicksort-Algorithmus:

```

qSort :: Ord a => [a] -> [a]
qSort []      = []
qSort (x:xs) = qSort [y | y<-xs, y<=x] ++
                [x] ++
                [y | y<-xs, y>x]
    
```

Wegen des in der letzten Zeile fehlenden rekursiven Aufrufs für die Funktion `qSort` bleibt die durch den Ausdruck

```
[y | y<-xs, y>x]
```

repräsentierte Liste mit Elementen, die größer als das Pivotelement sind, unsortiert. Dieser Umstand ist für das Typsystem nicht von Belang, da eine Liste in Haskell stets eine Folge von Elementen des gleichen Typs ist, so daß deren Anordnung für die Typisierung keine Rolle spielt. Der Programmfehler dokumentiert sich erst nach Beendigung einer Auswertung im Resultat des Sortiervorgangs.

Auch die nachstehenden Programmfehler haben keine Verletzung der Typregeln zur Folge:

- In der obigen Funktion `double`, die für ein gegebenes $x \in \mathbb{Z}$ den Wert

$$f(x) = 2 * x$$

berechnen soll, wird eine falsche multiplikative Konstante verwendet - anstelle der Zahl 2 etwa die Zahl 3:

```
double :: Int -> Int
double x = 3 * x
```

- In der obigen Funktion `square`, die für ein gegebenes $x \in \mathbb{Z}$ den Wert

$$f(x) = x^2$$

berechnen soll, wird ein falscher Operator verwendet - anstelle des Operators für die Multiplikation etwa der Operator für die Addition:

```
square :: Int -> Int
square x = x + x
```

Diese Beispiele zeichnen sich durch eine Gemeinsamkeit aus: Die jeweilige Funktion liefert ein Ergebnis, das - von Ausnahmefällen abgesehen, die etwa bei der Reduktion der Ausdrücke

- `qSort [1, 2, 3, 4]`,
- `double 0` und
- `square 2`

beobachtbar sind - nicht mit dem auf Grund der Spezifikation zu erwartenden Resultat übereinstimmt. Abweichungen von der Norm können sich in Haskell aber auch in anderer Weise manifestieren. Hierbei ist insbesondere an die Nicht-Terminierung einer Auswertung zu denken.

Das Problem der Nicht-Terminierung läßt sich anhand der Fakultätsfunktion veranschaulichen:

```
fak :: Int -> Int
fak 0 = 1
fak n = n * fak (n - 1)
```

Die Korrektheit der Funktionsdefinition wurde im Abschnitt 3.1 mit Hilfe des Beweisprinzips der vollständigen Induktion für Argumente $n \in \mathbb{N}_0$ gezeigt. Zur Erinnerung: Die verwendete Spezifikation sieht eine Berechnung der Fakultät ausschließlich für natürliche Zahlen vor. Was geschieht bei der Anwendung der Fakultätsfunktion auf eine negative ganze Zahl?

In Ermangelung eines Datentyps für die Menge natürlichen Zahlen wird in der Typdeklaration der Fakultätsfunktion der Datentyp `Int` verwendet. Dieser Datentyp repräsentiert die Menge der ganzen Zahlen. Somit stellt ein Ausdruck der Form

```
fak (-1)
```

ebenso wenig einen Typfehler dar wie ein Ausdruck der Form

```
fak 1
```

Präziser formuliert: Obwohl es im Widerspruch zur Spezifikation steht, akzeptiert das Typsystem als Argument der Fakultätsfunktion jeden Ausdruck, der zu einer negativen ganzen Zahl reduziert. Und da eine Variable ein unwiderlegbares Muster ist, wird das Ergebnis einer entsprechenden Applikation anhand der Gleichung

```
fak n = n * fak (n - 1)
```

ermittelt. Mit welchem Problem das verbunden ist, erkennt man unmittelbar aus der Definition: Die Auswertung terminiert nicht, weil das Argument der Fakultätsfunktion bei jedem rekursiven Aufruf um den Wert 1 vermindert wird, und für negative ganze Zahlen keine Abbruchbedingung vorhanden ist, die dem Prozeß ein Ende bereitet:

```
fak (-1)
-> (-1) * fak (-2)
-> (-1) * (-2) * fak (-3)
-> ...
```

Es gibt verschiedene Möglichkeiten, dem Problem zu begegnen. Denkbar ist unter anderem eine Erweiterung der Funktionsdefinition, die bewirkt, daß für negative ganze Zahlen der Wert 0 als Ergebnis geliefert wird:

```
fak n
| n == 0    = 1
| n > 0    = n * fak (n - 1)
| otherwise = 0
```

Aufgrund der fehlenden inhaltlichen Übereinstimmung mit der Spezifikation

$$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n * (n - 1) * \dots * 1, & \text{falls } n > 0 \end{cases}$$

ist diese Lösung jedoch unbefriedigend. Eher sachgerecht erscheint die Ausgabe einer Fehlermeldung mit Hilfe der vordefinierten Funktion `error`:

```
fak :: Int -> Int
fak n
| n == 0    = 1
| n > 0    = n * fak (n - 1)
| otherwise = error ("Funktion fak ist nur fuer " ++
                    "natuerliche Zahlen definiert")
```

Wie ersichtlich, handelt es sich bei der Funktion `error` um eine einstellige Funktion mit einem Argument vom Typ `String`:

```
error :: String -> a
```

Ihr Aufruf bewirkt einen Programmabbruch, der mit der Wiedergabe der dazugehörenden Zeichenkette einhergeht. Das heißt, wenn die Fakultät für eine negative ganze Zahl nach der Maßgabe der vorstehenden Definition berechnet wird, verhindert die Funktion `error` die Nicht-Terminierung und weist den Anwender explizit auf das Vorhandensein eines Fehlers hin. So auch bei dem Versuch, den Ausdruck

```
fak (-1)
```

auswerten zu lassen:


```

xterm
test@hydra:~ > hugs Fakultaet
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "Fakultaet.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Fakultaet.hs
Type :? for help
Fakultaet> fak (-1)

Program error: Funktion fak ist nur fuer natuerliche Zahlen definiert
Fakultaet>
    
```

Gleiches kann eine assert-Anweisung leisten: Bei einem Aufruf der Fakultätsfunktion mit einem Argument $n \in \mathbb{Z}$ wird die Programmausführung davon abhängig gemacht, daß die Bedingung $n \geq 0$ erfüllt ist. Richtet man das Augenmerk auf die Übersichtlichkeit der Programmgestaltung, ist eine assert-Anweisung sogar das geeignetere Mittel, um der Nicht-Terminierung entgegenzuwirken. Sie erlaubt nämlich eine klare Trennung zwischen der eigentlichen Funktionsdefinition und der Bedingung, deren Eintreffen zu überprüfen ist.

Die wichtigsten Voraussetzungen für die Steuerung des Programmablaufs mit Hilfe von Zusicherungen erfüllt die Programmiersprache Haskell:

- Es stehen ausreichende Mittel für die Formulierung boolescher Ausdrücke zur Verfügung; im einzelnen:
 - Ein vordefinierter Datentyp `Bool`, dessen Konstruktoren `True` und `False` die gleichlautenden booleschen Werte repräsentieren:

```

data Bool
  = False | True
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
    
```

- Verschiedene Vergleichsoperatoren: `<`, `<=`, `==`, `/=`, `>=`, `>`

In normaler Funktionsschreibweise ausgedrückt, handelt es sich hierbei um die Operatoren `<`, `≤`, `=`, `≠`, `≥`, `>`.

- Verschiedene logische Operatoren: `&&`, `|`, `not`

Diese Operatoren repräsentieren die logischen Funktionen \wedge , \vee , \neg .

- Mit der Funktion `error` steht eine Fehleroutine für den gegebenenfalls erforderlichen Programmabbruch zur Verfügung.

Hierauf aufbauend kann man für die Überprüfung von Zusicherungen zunächst folgende Funktion definieren:

```
assert :: String -> Bool -> a -> a
assert msg cond exp
  = case cond of
      True  -> exp
      False -> error msg
```

Das Ergebnis der Funktion `assert` hängt von dem Wert des booleschen Ausdrucks `cond` ab, der ihr als zweites Argument übergeben wird:

- Hat der boolesche Ausdruck den Wert `True`, liefert die Funktion `assert` als Ergebnis den Ausdruck `exp` - ihr drittes Argument.
- Hat der boolesche Ausdruck den Wert `False`, kommt es zu einem Abbruch der Programmausführung mit Hilfe der Funktion `error`, wobei als Fehlermeldung das erste Argument der Funktion `assert` verwendet wird.

Dieser Sachverhalt erlaubt es dem Anwender, die Auswertung eines beliebigen Ausdrucks `exp` von dem Eintreffen einer Bedingung abhängig zu machen, indem er in einem Programmtext an seiner Stelle einen Ausdruck der Form

```
assert message condition exp
```

einfügt. In diesem Zusammenhang bezeichnet *condition* die vordefinierte Bedingung, deren Eintreffen es zu überprüfen gilt, und *message* die inhaltlich auf das Nicht-Eintreffen dieser Bedingung bezogene - vom Anwender zu formulierende - Fehlermeldung. Das bedeutet konkret für die Berechnung der Fakultät: In einem Programmtext wird anstelle eines Ausdrucks der Form

```
fak n
```

ein Ausdruck der Form

```
assert message (n >= 0) (fak n)
```

verwendet.

Beispiel:

Unter Zuhilfenahme der Funktion `square` sei eine Funktion `squareFak` zu definieren, die für eine natürliche Zahl `n` den Wert

$$f(n) = (n!)^2$$

berechnet.

Definition ohne Überprüfung der Zusicherung, daß an die Fakultätsfunktion nur eine natürliche Zahl als Argument übergeben wird:

```
squareFak :: Int -> Int
squareFak n = square (fak n)
```

```
fak :: Int -> Int
fak 0 = 1
fak n = n * fak (n - 1)
```

Definition mit Überprüfung der Zusicherung, daß an die Fakultätsfunktion nur eine natürliche Zahl als Argument übergeben wird:

```
squareFak :: Int -> Int
squareFak n = square (assert msg (n >= 0) (fak n))
  where
    msg = "Funktion fak ist nur fuer " ++
          "natuerliche Zahlen definiert"

fak :: Int -> Int
fak 0 = 1
fak n = n * fak (n - 1)
```

Auswertung des Ausdrucks

```
squareFak (-1)
```

auf der Grundlage der letztgenannten Definition:

```
test@hydra:~$ hugs SquareFak
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode; Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "SquareFak.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
SquareFak.hs
Type :? for help
SquareFak> squareFak (-1)

Program error: Funktion fak ist nur fuer natuerliche Zahlen definiert
SquareFak>
```

Es liegt im Rahmen des Möglichen, daß die Fakultätsfunktion in verschiedenen Programmen Verwendung findet, denn der Gültigkeitsbereich einer Funktionsdefinition ist in der Programmiersprache Haskell nicht notwendigerweise auf ein einzelnes Skript beschränkt, oder daß die Fakultätsfunktion in ein und demselben Programm mehrfach angewendet wird. Unter solchen Umständen bedeutet es eine Arbeitserleichterung und einen Vorteil im Hinblick auf die Übersichtlichkeit der Programmgestaltung, wenn man die in dem vorstehenden Beispiel zum Ausdruck gebrachte Anwendung der `assert`-Funktion in die rechte Seite der Funktionsdefinition für die Fakultät integriert - etwa unter Zuhilfenahme einer lokalen Funktionsdefinition:

```
fak :: Int -> Int
fak n = assert msg (n >= 0) (fak' n)
  where
    fak' :: Int -> Int
    fak' 0 = 1
    fak' n = n * fak (n - 1)
    msg = "Funktion fak ist nur fuer " ++
          "natuerliche Zahlen definiert"
```

Der Grund liegt auf der Hand: Dem Problem der Nicht-Terminierung wird entgegen-
gewirkt, ohne die für die Berechnung der Fakultät relevante Zusicherung noch ein-
mal an irgendeiner anderen Stelle in einem Programmtext wiedergeben zu müssen.

Die Funktion `assert` kann zudem verwendet werden, um das Ergebnis einer Aus-
wertung auf seine Richtigkeit hin zu überprüfen. Als Beispiel sei der im Abschnitt
3.1 beschriebene Sortiervorgang betrachtet: Eine Folge von n Objekten ist bezüglich
einer Ordnungsrelation \leq genau dann in aufsteigender Reihenfolge sortiert, wenn das
Prädikat `isSorted(a, a π)` erfüllt ist, wobei a die zu sortierende Folge und a_π
die sortierte Folge bezeichnet. Das heißt, der Erfolg eines Sortiervorgangs mißt sich
an der Beantwortung der Frage, ob das Resultat im Widerspruch zu einer der Eigen-
schaften steht, die durch das Prädikat `isSorted(a, a π)` formalisiert werden.
Dieser Sachverhalt läßt sich relativ einfach auf der programmiersprachlichen Ebene
zum Ausdruck bringen:

```
isSorted :: Ord a => [a] -> [a] -> Bool
isSorted x y = isOrdered y && isPermutation x y

isOrdered :: Ord a => [a] -> Bool
isOrdered x = foldr1 (&&) (convert x)
  where
    convert :: Ord a => [a] -> [Bool]
    convert []      = [True]
    convert [x]    = [True]
    convert y@(x:xs) = zipWith (<=) y xs

isPermutation :: Ord a => [a] -> [a] -> Bool
isPermutation [] [] = True
isPermutation (x:xs) z@(y:ys) = isPermutation xs (delete x z)
isPermutation _ _ = False
```

In Anlehnung an die Funktion `qSort` beruht die Definition der Funktion
`isSorted` auf der Annahme, daß es sich bei einem Sortiervorgang um eine listen-
verarbeitende Operation handelt. Übergibt man ihr als erstes Argument die unsorti-
erte Liste und als zweites Argument die sortierte Liste, stellt sie fest,

- ob für die in der sortierten Liste enthaltenen Elemente x_i ($1 \leq i \leq n$) die Ei-
genschaft

$$x_i \leq x_{i+1}, \text{ für } i = 1, \dots, n - 1$$

gilt - das geschieht durch den Aufruf der Funktion `isOrdered` und entspricht
der Feststellung, ob das im Abschnitt 3.1 formulierte Prädikat `isOrdered(a π)`
erfüllt ist -, und

- ob jedes Element, das in der unsortierten Liste mindestens einmal enthalten ist,
ebenso oft in der sortierten Liste enthalten ist - das geschieht durch den Aufruf
der Funktion `isPermutation` und entspricht der Feststellung, ob das im Ab-
schnitt 3.1 formulierte Prädikat `isPermutation(a, a π)` erfüllt ist.

Wenn beides zutrifft, liefert die Funktion `isSorted` entsprechend dem Term

$$\text{isSorted}(a, a_\pi) \Leftrightarrow \text{isOrdered}(a_\pi) \wedge \text{isPermutation}(a, a_\pi)$$

als Ergebnis den Wert `True`, ansonsten den Wert `False`. Hierdurch eröffnet sich die Möglichkeit, eine fehlerhafte Sortierung - etwa im Fall der Funktion `qSort` - mit Hilfe einer zu überprüfenden Zusicherung automatisch zu erkennen:

```
qSort :: Ord a => [a] -> [a]
qSort x = assert msg (isSorted x res) res
  where
    msg = "Funktion qSort fehlerhaft definiert"
    res = qSort' x
    qSort' :: Ord a => [a] -> [a]
    qSort' [] = []
    qSort' (x:xs) = qSort' [y | y<-xs, y<=x] ++
                    [x] ++
                    qSort' [y | y<-xs, y>x]
```

Diese Definition wirkt aufgrund der zweimaligen Verwendung des Ausdrucks `res` noch recht umständlich. Sie läßt sich in einfacherer Form darstellen, wenn für Testzwecke eine weitere Funktion

```
assertF :: String -> (a -> Bool) -> a -> a
assertF msg func exp = assert msg (func exp) exp
```

zur Verfügung gestellt wird, die die Tatsache ausnutzt, daß die Programmiersprache Haskell die Handhabung von Funktionen höherer Ordnung erlaubt: Der Funktion `assertF` muß als zweites Argument eine Funktion vom Typ

```
a -> Bool
```

übergeben werden. Das heißt, sie akzeptiert als solches auch eine partielle Funktionsanwendung des gleichen Typs - mit dem Ergebnis, daß die Definition für die Funktion `qSort` folgendermaßen umformuliert werden kann:

```
qSort :: Ord a => [a] -> [a]
qSort x = assertF msg (isSorted x) (qSort' x)
  where
    msg = "Funktion qSort fehlerhaft definiert"
    qSort' :: Ord a => [a] -> [a]
    qSort' [] = []
    qSort' (x:xs) = qSort' [y | y<-xs, y<=x] ++
                    [x] ++
                    qSort' [y | y<-xs, y>x]
```

Ein abschließendes Beispiel, das noch einmal den Nutzen der `assert`-Funktion unterstreicht: Die Funktion `qSort` werde ohne Zusicherung als Hilfsmittel für die Definition einer Funktion `tailProduct` verwendet, deren Aufgabe es ist, alle Elemente einer Liste vom Typ `[Int]` mit Ausnahme des kleinsten Elements aufzusummieren und die Summe mit ebendiesem Element zu multiplizieren:

```
tailProduct :: [Int] -> Int
tailProduct x = head x' * sum (tail x')
  where
    x' = qSort x
```

Unter dieser Voraussetzung liefert die Funktion `tailProduct` auch korrekte Ergebnisse, wenn der Sortiervorgang durch den zu Beginn des Abschnitts 3.1 beschriebenen Fehler beeinträchtigt wird. Und zwar aus zwei Gründen:

Überprüfung von Zusicherungen zu deaktivieren, wenn die Testphase für ein Programm beendet ist. Im Fall der Programmiersprache C können die Aufrufe des `assert`-Makros zum Beispiel durch die Präprozessor-Anweisung

```
#define NDEBUG
```

unwirksam gemacht werden. Sie bewirkt, daß der Compiler die Aufrufe als Kommentar betrachtet.

Eine vergleichbare Lösung erscheint für die Programmiersprache Haskell ungeeignet, weil funktionale Programme nicht aus einer Folge von Anweisungen bestehen, die sequentiell abgearbeitet werden, und die Funktionen `assert` und `assertF` dementsprechend nicht darauf ausgerichtet sind, Zusicherungen in einer separaten Programmzeile zum Ausdruck zu bringen, die im Bedarfsfall ignoriert werden kann. Der Auswertungsmechanismus muß stets in der Lage sein, in einer Applikation der Form

```
assert message condition expression
```

bzw. der Form

```
assertF message function expression
```

auf den Ausdruck *expression* zuzugreifen. Für die angestrebte Flexibilisierung bietet es sich deshalb vielmehr an, der Funktion `assert` - und aufgrund ihrer Definition automatisch auch der Funktion `assertF` - zwei unterschiedliche Bedeutungen zuzuordnen: Während der Testphase überprüft sie die Programmausführung, indem das Eintreffen von bestimmten, vorher definierten Bedingungen beobachtet wird. Nach Beendigung der Testphase verhält sie sich im Sinne der Definition

```
assert :: String -> Bool -> a -> a
assert msg cond exp = exp
```

wie eine Projektion auf das dritte Argument.

Mit Hilfe des im Abschnitt 2.3.2 beschriebenen Mechanismus der Überladung läßt sich dies allerdings nicht realisieren, denn die Beantwortung der Frage, welches Verhalten der Funktion `assert` für die Auswertung im Einzelfall maßgebend ist, hängt nicht vom Typ ihrer Argumente ab. Erforderlich sind Änderungen in der Implementierung der Programmiersprache Haskell: Die Funktion `assert` muß systemabhängig als eingebaute Funktion definiert werden, deren Verhalten der Anwender über die - dem System auf einem geeigneten Weg mitzuteilende - Information steuern kann, ob eine Testphase durchlaufen wird.

Darüber hinaus weist das Grundkonzept noch einen Mangel auf, der schwerer wiegt als der soeben beschriebene: Die Überprüfung von Zusicherungen soll die Konsistenz zwischen einem Programm und seiner Spezifikation sicherstellen, keinesfalls aber die Ausgabedaten ändern. Diesem wichtigen Prinzip widerspricht das Grundkonzept. So kann es durchaus vorkommen, daß die Ausführung eines Programms terminiert und einen korrekten Ergebniswert liefert, wenn auf die Überprüfung von Zusicherungen in ihrer bisherigen Form verzichtet wird, ansonsten jedoch nicht terminiert. Mit anderen Worten: Wenn man die Bedeutung eines Programms mit dem

Wert assoziiert, den seine Ausführung als Ergebnis liefert, verändert sich durch die Einbindung von Zusicherungen in den Quellcode unter Umständen dessen Semantik.

Um diesen Sachverhalt zu veranschaulichen, sei eine einfache Aufgabenstellung betrachtet: Gegeben ist eine Folge von n positiven natürlichen Zahlen a_1, a_2, \dots, a_n . Für jede Zahl a_i ($1 \leq i \leq n$) aus dieser Folge soll der Wert

$$s_i = \sum_{k=1}^{a_i} k$$

berechnet werden.

Stellt man sowohl die Zahlenfolge a_1, a_2, \dots, a_n als auch die Zahlenfolge

$$s_1 = \sum_{k=1}^{a_1} k, \quad s_2 = \sum_{k=1}^{a_2} k, \quad \dots, \quad s_n = \sum_{k=1}^{a_n} k$$

in Form einer Liste dar, läßt sich die spezifizierte Aufgabe zum Beispiel mit Hilfe der folgenden Funktion bewältigen:

```
gauss :: [Int] -> [Int]
gauss xs = map gSum xs
  where
    gSum :: Int -> Int
    gSum 1 = 1
    gSum n = n + gSum (n - 1)
```

Auch diese Definition nutzt die Tatsache aus, daß die Programmiersprache Haskell die Handhabung von Funktionen höherer Ordnung erlaubt: Die in der [Prelude] vordefinierte Funktion `map` erwartet als erstes Argument eine Funktion f , als zweites Argument eine Liste `xs` und wendet die Funktion f auf alle Elemente der Liste `xs` an. Um die gesuchte Zahlenfolge s_1, s_2, \dots, s_n zu ermitteln, wird ihr als erstes Argument eine Funktion übergeben, die für eine positive natürliche Zahl n die Summe der natürlichen Zahlen von 1 bis n zu berechnen vermag - die Funktion `gSum` -, und als zweites Argument das Argument der Funktion `gauss`.

Die Definition der Funktion `gSum` besteht aus zwei Gleichungen. Beide beinhalten auf der linken Seite ein Muster, um zwischen der Zahl 1 und einer Zahl $n \neq 1$ ($n \in \mathbb{Z}$) als Argument unterscheiden zu können. Das heißt, ein Aufruf der Funktion `gSum` bedingt einen Mustervergleich, der gemäß den im Abschnitt 2.3.3 dargelegten Regeln mit Hilfe der überladenen Gleichheitsfunktion (`==`) zunächst überprüft, ob der aktuelle Parameter zum Muster der ersten Gleichung paßt. Es ist offensichtlich, daß der aktuelle Parameter in diesem Zusammenhang gegebenenfalls reduziert werden muß. Infolgedessen terminiert die Auswertung des Ausdrucks

```
gSum (infinite 1)
```

nicht, falls die Funktion `infinite` als

```
infinite :: a -> a
infinite x = infinite x
```


definiert wird, denn die Auswertung des an die Funktion `gSum` übergebenen Arguments terminiert nicht:

```
infinite 1
-> infinite 1
-> infinite 1
-> ...
```

Auf den ersten Blick mag hierdurch der Eindruck entstehen, daß auch eine Auswertung nicht terminiert, bei der das Argument der Funktion `gauss` einen Redex der Form `infinite expression` beinhaltet - oder einen beliebigen anderen Redex, dessen Auswertung nicht terminiert. Dieser Eindruck stellt sich jedoch als falsch heraus, wenn man zum Beispiel den Ausdruck

```
take 2 (gauss [1, 2, infinite 3])
```

auswertet:

```
test@hydra:~ > hugs Gauss
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs";
Reading file "Gauss.hs";

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Gauss.hs
Type :? for help
Gauss> take 2 (gauss [1, 2, infinite 3])
[1,3]
Gauss> █
```

Erklärung: Aufgrund der Nicht-Striktheit der Programmiersprache Haskell wird das Argument einer Funktion nur reduziert, wenn sein Wert für die Ermittlung des Gesamtergebnisses bekannt sein muß (und dann auch nur einmal). Ist das Argument in irgendeiner Form strukturiert, handelt es sich also zum Beispiel um ein Tupel oder um eine Liste, beschränkt sich die Reduktion zudem auf die für das Gesamtergebnis relevanten Teile (vgl. Abschnitt 2.2).

Im Fall des Ausdrucks

```
take 2 (gauss [1, 2, infinite 3])
```

kommt diese Beschränkung voll zum Tragen. Für die Ermittlung des Gesamtergebnisses muß der Wert des Ausdrucks `gSum (infinite 3)` nicht berechnet werden, weil die Funktion `take` als erstes Argument eine nicht-negative ganze Zahl `n` erwartet, als zweites Argument eine Liste `xs` und als Ergebnis eine Liste liefert, die aus den ersten `n` Elementen der Liste `xs` besteht, so daß eine Summenbildung durch die Funktion `gSum` nur für die beiden ersten Elemente der Liste

```
[1, 2, infinite 3]
```

erforderlich ist. Mithin terminiert die Auswertung.

Dem kann die Überprüfung einer Zusicherung in ihrer bisherigen Form zuwiderlaufen. In Ermangelung eines Datentyps für die Menge der positiven natürlichen Zahlen beruht die Typdeklaration der Funktion `gSum` auf dem Datentyp `Int`. Obwohl es im Widerspruch zur Spezifikation der Funktion `gauss` steht, akzeptiert das Typsystem als ihr Argument also jeden Ausdruck, der eine ganze Zahl $n < 1$ repräsentiert. Das stellt ein Problem dar, mit dem in ähnlicher Form auch die Fakultätsfunktion behaftet ist: Wenn das Argument der Funktion `gauss` einen derartigen Ausdruck beinhaltet, terminiert die Auswertung nicht, weil die Summe für das entsprechende Element anhand der Gleichung

$$gSum\ n = n + gSum\ (n - 1)$$

bestimmt wird, sich das Argument der Funktion `gSum` infolgedessen bei jedem rekursiven Aufruf um den Wert 1 vermindert, und für ganze Zahlen $n < 1$ keine Abbruchbedingung vorhanden ist, die dem Prozeß ein Ende bereitet:

```
gSum 0
-> 0 + gSum (-1)
-> 0 + (-1) + gSum (-2)
-> ...
```

Man kann diesem Problem entgegenwirken, indem mit Hilfe der Funktion

```
allPositive :: (Num a, Ord a) => [a] -> Bool
allPositive [] = False
allPositive l@(x:xs) = all (0 <) l
```

überprüft wird, ob es sich bei jedem Element der fraglichen Liste um eine positive ganze Zahl handelt:

```
gauss :: [Int] -> [Int]
gauss xs = map gSum (assertF msg allPositive xs)
  where
    msg = "Funktion gauss: Summenbildung " ++
          "fuer eine Zahl n <= 0"
    gSum :: Int -> Int
    gSum 1 = 1
    gSum n = n + gSum (n - 1)
```

Hier das Ergebnis für die Auswertung des Ausdrucks `gauss [1, 2, 0]`:

```
test@hydra:~ > hugs Gauss
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode: Restart with command line option -98 to enable extensions
Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "Gauss.hs":
Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Gauss.hs
Type :? for help
Gauss> gauss [1, 2, 0]
Program error: Funktion gauss: Summenbildung fuer eine Zahl n <= 0
Gauss> █
```

Allerdings ist die zweistellige Funktion (`<`) strikt in ihren Argumenten. Für die Überprüfung der durch die Funktion `allPositive` zum Ausdruck gebrachten Zusicherung bedeutet das: Die Auswertung terminiert nicht, wenn das Argument der Funktion `gauss` einen Redex der Form

```
infinite expression
```

beinhaltet; und zwar selbst dann nicht, wenn dieser Redex wie im Fall des Ausdrucks

```
take 2 (gauss [1, 2, infinite 3])
```

für die Ermittlung des Gesamtergebnisses ohne Belang ist. Der Grund: Die Funktion `all` übergibt ihr erstes Argument gemäß der Definition

```
all :: (a -> Bool) -> [a] -> Bool
all p = and . map p
```

an die oben beschriebene Funktion `map`, so daß sich die Anwendung der partiellen Applikation (`0 <`) auf alle Elemente der Liste

```
[1, 2, infinite 3]
```

erstreckt. Also wird auch der Wert des Ausdrucks (`infinite 3`) berechnet, um festzustellen, ob er größer als 0 ist.

Die Nicht-Terminierung ist zwar vermeidbar, indem die Zusicherung in die lokal definierte Funktion `gSum` integriert wird:

```
gauss :: [Int] -> [Int]
gauss xs = map gSum xs
  where
    gSum :: Int -> Int
    gSum n = assert msg (n > 0) (gSum' n)
    gSum' :: Int -> Int
    gSum' 1 = 1
    gSum' n = n + gSum' (n - 1)
    msg = "Funktion gauss: Summenbildung " ++
          "fuer eine Zahl n <= 0"
```

Ein derartiger Ausweg setzt jedoch voraus, daß der Anwender erkennt, inwieweit die Kontrolle des Programmablaufs im Widerspruch zur Nicht-Striktheit der Programmiersprache Haskell steht. Für komplexere Aufgabenstellungen als die hier betrachtete wäre das ein mit erheblichem Aufwand verbundener Nachteil des Grundkonzepts. Im folgenden Abschnitt wird deshalb untersucht, auf welche andere Weise dem Mangel begegnet werden kann.

3.3.3 Prinzipien einer semantikerhaltenden Überprüfung von Zusicherungen (Assertions)

Die Einbindung von Zusicherungen in den Quellcode eines Programms dient nicht dem Zweck, seine Bedeutung zu verändern. Mit ihrer Hilfe soll lediglich die Frage beantwortet werden, ob das Programm im Einklang mit der zugrundeliegenden Spezifikation steht. Das heißt, wenn ein Programm `P` durch die Einbindung von Zusicherungen in den Quellcode in ein Programm `P'` transformiert wird, sollten beide Pro-

gramme semantisch äquivalent sein. Anders ausgedrückt: Die Semantik von P sollte erhalten bleiben.

Was bedeutet diese Forderung konkret?

Das Bindeglied zwischen zwei Programmen P und P' , die sich in der vorstehend bezeichneten Weise voneinander unterscheiden, ist das Ergebnis ihrer Ausführung. Um Aussagen darüber treffen zu können, welche Wirkung die Überprüfung einer Zusicherung erzielen darf bzw. welche nicht, ist es deshalb naheliegend, die Bedeutung eines Programms mit dem Ergebnis zu assoziieren, das die Reduktion des auszuwertenden Ausdrucks zur WHNF liefert. Betrachtet man das Programm P , können in diesem Zusammenhang grundsätzlich zwei Fälle unterschieden werden:

1. Die Programmausführung terminiert und liefert ein Ergebnis $x \neq \perp$, das unter Umständen von dem aufgrund der Spezifikation zu erwartenden Ergebnis abweicht.
2. Das Ergebnis der Programmausführung ist \perp . Das heißt, die Programmausführung terminiert nicht, oder es ist ein Fehler aufgetreten, der eine Reduktion zur WHNF verhindert, etwa eine Division durch die Zahl 0 (vgl. Abschnitt 2.2).

Was der erste Fall für die Forderung nach dem Erhalt der Semantik bedeutet, wenn das Programm P im Einklang mit der zugrundeliegenden Spezifikation steht, liegt auf der Hand: Die Ausführung des Programms P' muß ebenfalls terminieren und dasselbe Ergebnis $x \neq \perp$ liefern, das die Ausführung des Programms P liefert.

Diesem Anspruch wird die Überprüfung von Zusicherungen mit Hilfe der im Abschnitt 3.3.1 definierten Funktionen `assert` und `assertF` nur zum Teil gerecht. Angesichts der Tatsache, daß sich beide Funktionen wie eine Projektion auf ihr drittes Argument verhalten, wenn die Programmausführung im Einklang mit der zugrundeliegenden Spezifikation steht, ist zwar sichergestellt, daß die Ausführung der Programme P und P' dasselbe Ergebnis $x \neq \perp$ liefert, wenn letztere terminiert. Genau dies muß jedoch nicht zwangsläufig der Fall sein. Wie die Funktion `gauss` gezeigt hat, terminiert die Ausführung des Programms P' zum Beispiel nicht, falls die beiden folgenden Voraussetzungen erfüllt sind:

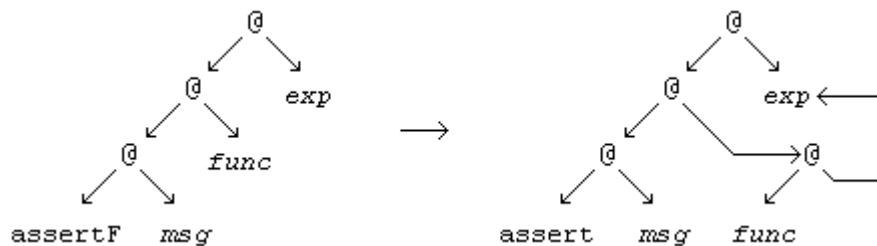
1. Die Überprüfung einer Zusicherung stößt die Reduktion eines Ausdrucks an, dessen Wert für die Ermittlung des Gesamtergebnisses ohne Belang ist, der also aufgrund der Nicht-Striktheit der Programmiersprache Haskell ansonsten - im Programm P - unausgewertet bliebe.
2. Die durch die Überprüfung der Zusicherung angestoßene Reduktion terminiert nicht.

Hieraus leitet sich die Notwendigkeit ab, von der Funktion `assert` - und aufgrund ihrer Definition automatisch auch von der Funktion `assertF` - ein Verhalten zu verlangen, das auf der programmiersprachlichen Ebene nicht realisierbar ist: Die Überprüfung einer Zusicherung darf grundsätzlich keine Auswertung erzwingen, sondern beruht auf der Inaugenscheinnahme bereits ausgewerteter Ausdrücke. Das heißt, unter Umständen ist es erforderlich, einen Aufruf der Funktion `assert` (`assertF`) ohne Resultat abubrechen.

Zweifellos bedeutet diese Forderung nicht, daß für einen Ausdruck der Form

`assertF msg func exp`

bereits die Reduktion



unzulässig ist oder etwa die Anwendung der Funktion *func* auf den Ausdruck *exp* - ein Aufruf der Funktion `assertF` würde sonst unabhängig von der Gestalt des Ausdrucks *exp* niemals ein Ergebnis liefern. Gleiches gilt für einen Ausdruck der Form

`assert msg (exp_1 == exp_2) exp_3`

Er verlangt, die Anwendung des Gleichheitsoperators zuzulassen.

Andererseits kann man das Auswertungsverbot nicht auf das dritte Argument der Funktion `assert` (`assertF`) beschränken. Eine solche Strategie wäre zwar im Fall der Funktion `gauss` erfolgreich, sie beruht jedoch auf einer falschen Voraussetzung. Nämlich auf der Voraussetzung, daß jede während des Programmablaufs zu überprüfende Zusicherung eine Eigenschaft widerspiegelt, die das dritte Argument der Funktion `assert` (`assertF`) auszeichnet - bzw. einen darin enthaltenen Teilausdruck. Das dies nicht notwendigerweise der Fall ist, zeigt die folgende Definition:

```
foo :: Int -> Int -> Int -> Int
foo x y z = assert msg (x /= 0) (y + z)
```

Bleibt der Bereich, in dem nach einem Aufruf der Funktion `assert` (`assertF`) keine Auswertung vorgenommen werden darf, auf deren drittes Argument beschränkt, terminiert die Reduktion des Ausdrucks

`foo (infinite 1) 2 3`

nicht, weil der auf der rechten Seite der Definition verwendete Ungleichheitsoperator im Rahmen der Überprüfung der Zusicherung eine nicht-terminierende Reduktion des ersten Arguments der Funktion `foo` zur Folge hat - implizit vorausgesetzt, daß sich die Funktion `infinite` gemäß der im vorstehenden Abschnitt angegebenen Definition verhält:

```
infinite 1
-> infinite 1
-> infinite 1
-> ...
```

Präzise formuliert lautet deshalb die obige Restriktion, der das Verhalten der Funktionen `assert` und `assertF` unterworfen werden muß, um der Forderung nach dem Erhalt der Semantik gerecht zu werden:

grammausdrucks anstößt. Dementsprechend würde auch die Auswertung des Ausdrucks `foo (infinite 1) 2 3` terminieren. Problematisch sind im Hinblick auf die Forderung nach dem Erhalt der Semantik allerdings keineswegs nur Programmausdrücke: Wie bereits dargelegt wurde, reduziert ein Ausdruck der Form

```
assertF msg func exp
```

aufgrund der Definition der Funktion `assertF` zu einem Ausdruck der Form

```
assert msg (func exp) exp
```

und setzt notwendigerweise voraus, daß für die Überprüfung der durch ihn repräsentierten Programmeigenschaft die Anwendung der Funktion `func` reduziert werden darf. Hierdurch besteht das Risiko, die Semantik eines fehlerfreien Programms selbst dann zu verändern, wenn die bisher geforderten Restriktionen beachtet werden. Der Ausdruck

```
assertF msg infinite exp
```

liefert hierfür ein einfaches Beispiel - wenn nach der Transformation in den Ausdruck

```
assert msg (infinite exp) exp
```

die Anwendung der Funktion `infinite` reduziert wird, terminiert die Programmausführung nicht:

```
infinite exp
-> infinite exp
-> infinite exp
-> ...
```

Von dieser Problematik sind analog sämtliche Ausdrücke betroffen, deren Reduktion zulässig ist, wenn ein Aufruf der Funktion `assert` (`assertF`) abgearbeitet wird. Will man der Forderung nach dem Erhalt der Semantik vollkommen gerecht werden, folgt daraus:

Der Überprüfung einer Zusicherung müssen auch Beschränkungen auferlegt werden, die sicherstellen, daß die Semantik eines Programms nicht aufgrund des Anstoßens einer erlaubten Reduktion verändert wird.

Denkbar ist zum Beispiel eine zeitliche Limitierung (Timeout) der Überprüfung oder eine zahlenmäßige Beschränkung der Reduktionen, die sie anstoßen darf. Solche Lösungen entbehren jedoch nicht einer gewissen Willkür. Im Fall der Implementierung sollte der Anwender deshalb zumindest die Möglichkeit haben, die entsprechenden Vorgabewerte selbst zu wählen, und/oder die Beschränkung ganz aufzuheben.

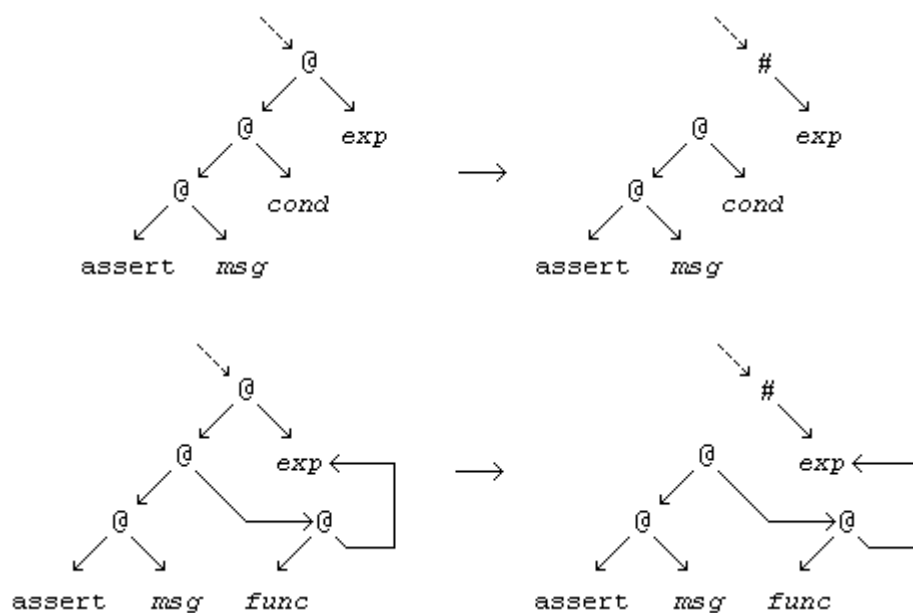
Es stellt sich jetzt die Frage, welche Konsequenzen es hat, wenn die Überprüfung einer Zusicherung tatsächlich ergebnislos abgebrochen werden muß.

Angesichts der Tatsache, daß eine derartige Notwendigkeit nicht in dem Erkennen eines Fehlers begründet liegt, kann die Antwort auf diese Frage zunächst einmal nur

lauten: Die normale Programmausführung wird fortgesetzt, indem der Auswertungsmechanismus Zugriff auf das dritte Argument der Funktion `assert` (`assertF`) erlangt. Genau diesen Ausdruck liefert sie nämlich als Ergebnis, wenn die Programmausführung im Einklang mit der zugrundeliegenden Spezifikation steht. Das heißt mit anderen Worten:

Wenn die Überprüfung einer Zusicherung abgebrochen wird, muß dieser Vorgang im Ergebnis wie eine Projektion auf das dritte Argument der Funktion `assert` bzw. `assertF` wirken.

Um dem gerecht zu werden, reicht es aus, die fragliche Applikation mit einem Indirektionsknoten zu überschreiben, der als Zeiger auf das bezeichnete Argument fungiert:



Ein wichtiger Aspekt darf in diesem Zusammenhang jedoch nicht außer Acht bleiben: Es ist möglich, daß die normale Programmausführung eine Reduktion anstößt, die zuvor im Rahmen der Überprüfung einer Zusicherung als unzulässig erachtet wurde.

Beispiel:

Gegeben sei die aus dem Abschnitt 3.3.1 bekannte Definition der Fakultätsfunktion:

```
fak :: Int -> Int
fak n = assert msg (n >= 0) (fak' n)
  where
    fak' :: Int -> Int
    fak' 0 = 1
    fak' n = n * fak (n - 1)
    msg = "Funktion fak ist nur fuer " ++
          "natuerliche Zahlen definiert"
```

Die rechte Seite dieser Definition ist so formuliert worden, daß die Funktion `assert` unmittelbar nach einem Aufruf der Fakultätsfunktion überprüft, ob deren Argument eine natürliche Zahl darstellt. Erst nach Beendigung dieser Überprüfung

berechnet die lokal definierte Funktion `fak'` den eigentlichen Ergebniswert. Das hat für die Auswertung des Ausdrucks

```
fak (1 - 2)
```

zur Folge: Wenn man der Überprüfung einer Zusicherung die dargelegten Restriktionen auferlegt, muß der Aufruf der Funktion `assert` ergebnislos abgebrochen werden, weil das an die Fakultätsfunktion übergebene Argument einen Programmausdruck darstellt, der für die Beantwortung der Frage, ob es sich um eine natürliche Zahl handelt, nicht reduziert werden darf. Es ist offensichtlich, daß der Wert des Ausdrucks `(1 - 2)` aber bekannt sein muß, um entscheiden zu können, welche der beiden Gleichungen, die die Funktion `fak'` definieren, für die Berechnung der Fakultät maßgebend ist. Dementsprechend wird er reduziert, nachdem sich die Funktion `assert` wie eine Projektion auf ihr drittes Argument verhalten hat, und zwar durch den Aufruf der Funktion `fak'`.

Dieser Sachverhalt impliziert:

Wenn die normale Programmausführung eine Reduktion bewirkt hat, die zuvor im Rahmen der Überprüfung einer Zusicherung als unzulässig erachtet wurde, muß diese Zusicherung erneut überprüft werden.

Was es im Fall der Fakultätsfunktion bedeuten würde, wenn dieser Grundsatz nicht beachtet wird, liegt auf der Hand: Die Auswertung des Ausdrucks

```
fak (1 - 2)
```

terminiert nicht, weil der Ausdruck `(1 - 2)` eine negative ganze Zahl darstellt, so daß die Berechnung des Ergebniswerts mit Hilfe der Funktion `fak'` aus den im Abschnitt 3.3.1 dargelegten Gründen nicht terminiert. Genau betrachtet, würde die Überprüfung der für die Fakultätsfunktion maßgebenden Zusicherung nicht einmal ein Ergebnis liefern, wenn ihr als Argument ein ganzzahliges Literal wie 0 oder 1 übergeben wird, das heißt, sie wäre also letztendlich vollkommen nutzlos. Worin dies begründet liegt, offenbart der Haskell 98 Report - [Peyton Jones 1999]: "An integer literal represents the application of the function `fromInteger` to the appropriate value of type `Integer`."

Der soeben formulierte Grundsatz wirft natürlich verschiedene Fragen auf, zum Beispiel:

1. Wie erkennt ein System im normalen Programmablauf, ob eine durchgeführte Reduktion im Rahmen der Überprüfung einer Zusicherung als unzulässig erachtet wurde - und vor allem: im Rahmen der Überprüfung welcher Zusicherung?
2. Wann soll der Auswertungsmechanismus dazu veranlaßt werden, eine Zusicherung erneut zu überprüfen - sofort nachdem der ursprünglich für den Abbruch ihrer Überprüfung maßgebende Ausdruck reduziert wurde, oder erst zu einem späteren - genau zu definierenden - Zeitpunkt?
3. Wie kann der Auswertungsmechanismus überhaupt dazu veranlaßt werden, eine Zusicherung erneut zu überprüfen?
4. Was geschieht, wenn auch die nochmalige Überprüfung einer Zusicherung abgebrochen werden muß?

Solche Fragen sind technischer Natur. Sie sprechen Probleme an, die gelöst werden müssen, wenn das an dieser Stelle zu erarbeitende Konzept in einer Implementierung umgesetzt werden soll. Insofern erscheint es gerechtfertigt, hier auf nähere Erläuterungen zu verzichten, und die Aufmerksamkeit zwei Problemen zuzuwenden, die grundsätzlicher Natur sind:

1. Dem "Informationsverlust", der entsteht, wenn ein Aufruf der Funktion `assert` (`assertF`) vorzeitig beendet wird.
2. Der Handhabung von Constant Applicative Forms (CAF's).

Zunächst einige Anmerkungen Punkt 1 betreffend: Es wurde bereits dargelegt, daß der Vorgang, mit dem die Überprüfung einer Zusicherung abgebrochen wird, im Ergebnis wie eine Projektion auf das dritte Argument der Funktion `assert` bzw. `assertF` wirken muß. Systemintern entsteht hierdurch der Eindruck, daß die Programmausführung im Einklang mit der zugrundeliegenden Spezifikation steht, und die fragliche Auswertung normal beendet wurde. Dies führt einerseits zu dem gewünschten Effekt, daß der Auswertungsmechanismus Zugriff auf das dritte Argument der Funktion `assert` (`assertF`) erlangt. Andererseits wird hierdurch der dazugehörige Stack verworfen (vgl. Abschnitt 2.4.1). Wenn es erforderlich ist, die Überprüfung einer Zusicherung erneut anzustoßen, bedeutet dies ein gewisses Maß an Informationsverlust, weil derselbe Stack dann neu aufgebaut werden muß.

Um dem Problem zu begegnen, könnte man den Mechanismus, mit dem die Überprüfung einer Zusicherung abgebrochen wird, so ausgestalten, daß der dazugehörige Stack-Inhalt für die eventuelle Wiederherstellung in geeigneter Form - zum Beispiel mittels separatem Stack - aufbewahrt wird. Dies bedeutet allerdings zusätzlichen Aufwand, der vergebens ist, wenn die normale Programmausführung jene Reduktion nicht anstößt, die als unzulässig erachtet wurde.

Eine Alternative ist die "Wiederherstellung einer Zusicherung durch die Rückgängigmachung der Projektion" gemäß der nachstehend beschriebenen Vorgehensweise.

Gegeben sei eine Zusicherung Z , für die folgendes gilt:

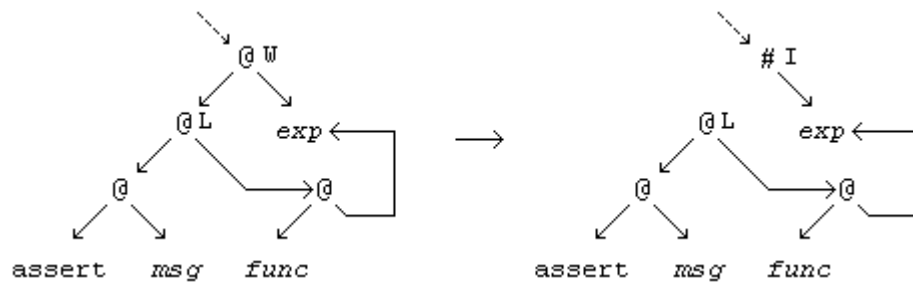
- Die Zusicherung Z wird durch einen Ausdruck der Form

$$\text{assertF } \textit{msg func exp}$$

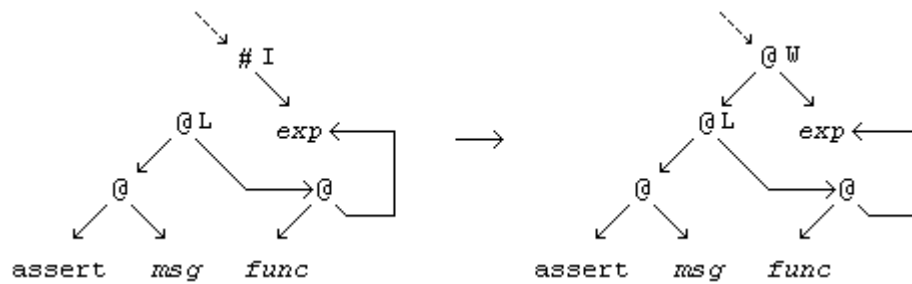
repräsentiert. Der Wurzelknoten dieser Funktionsanwendung sei mit W bezeichnet, der linke Nachfolger des Knotens W sei mit L bezeichnet.

- Die Überprüfung der Zusicherung Z muß abgebrochen werden, weil ein Programmausdruck A nicht reduziert werden darf.
- Die Überprüfung der Zusicherung Z wird vorzeitig beendet, indem der Knoten W mit einem Indirektionsknoten I überschrieben wird, der als Zeiger auf den Ausdruck exp fungiert.

Schematisch dargestellt, gestaltet sich der Abbruch der Überprüfung wie folgt:



Wenn der Ausdruck A im Rahmen der normalen Programmausführung reduziert wird, kann dieser Abbruch sehr einfach rückgängig gemacht werden - der Indirektionsknoten I wird durch einen Applikationsknoten ersetzt, dessen linker Nachfolger der Knoten L ist:



Die erneute Überprüfung der Zusicherung Z kann nun veranlaßt werden, indem man den Applikationsknoten W an den Unwind-Mechanismus übergibt.

Ein solches Verfahren hat zwar zur Folge, daß der Stack neu aufgebaut wird, aber ursprünglich vollendete Reduktionsschritte wie



bedürfen keiner Wiederholung, weil sie vom Ergebnis her im Graphen erhalten bleiben. Der Informationsverlust, den dieses Verfahren mit sich bringt, hält sich also in vertretbaren Grenzen, weshalb ihm an dieser Stelle aufgrund des geringen Aufwands, mit dem es verbunden ist, der Vorzug gegenüber anderen Methoden gegeben wird. Ein Gesichtspunkt darf in diesem Zusammenhang allerdings nicht außer Acht gelassen werden: Wenn die zugrundeliegende Implementierung über eine automatische Garbage Collection verfügt, muß nach dem Abbruch der Überprüfung der Zusicherung Z sichergestellt sein, daß der Speicherplatz, den der am Knoten L beginnende Teilgraph in Anspruch nimmt, nicht für eine neue Belegung freigegeben wird.

Nun Anmerkungen Punkt 2 betreffend: Als CAF wird ein Superkombinator mit der Stelligkeit 0 bezeichnet (vgl. Abschnitt 2.4.1). Die Definition

`incr = (+) 1`

repräsentiert zum Beispiel eine solche CAF.

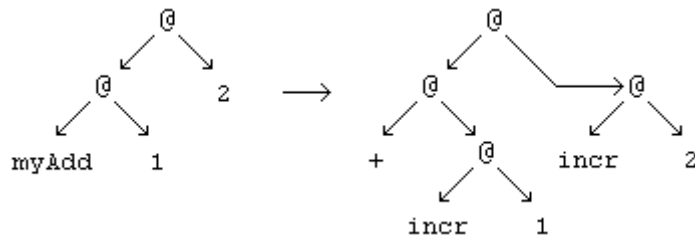
Da nullstellige Superkombinatoren keine formalen Parameter haben, erfahren sie in der Regel eine besondere Handhabung. In welcher Art und Weise, sei mit Hilfe der folgenden Funktionsdefinition veranschaulicht:

```
myAdd :: Int -> Int -> Int
myAdd x y = incr x + incr y
```

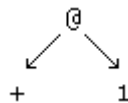
Bei einer Auswertung des Ausdrucks

```
myAdd 1 2
```

wird die Funktion `incr` zweimal aufgerufen - die Additionsfunktion ist strikt in beiden Argumenten:



Das bedeutet jedoch nicht, auch den Graphen



zweimal bilden zu müssen. Und zwar aus folgendem Grund: Nullstellige Superkombinatoren repräsentieren konstante Ausdrücke. Eine Implementierung ist deshalb darauf ausgerichtet, den Graphen einer CAF nur beim ersten Aufruf der entsprechenden Funktion zu bilden, ihn separat zu speichern und bei nachfolgenden Funktionsaufrufen über einen Shared Node auf genau diesen einen Graphen zuzugreifen [Peyton Jones 1987] - vgl. auch Abschnitt 2.4.1. Sofern der Graph einer CAF einen Redex repräsentiert bzw. einen Redex als Teilausdruck enthält, wird hierdurch sichergestellt, daß der betreffende Ausdruck nicht mehrfach ausgewertet werden muß.

Für die Überprüfung einer Zusicherung ergeben sich aus diesem Sachverhalt zwei Konsequenzen:

1. Die Verwendung nullstelliger Superkombinatoren sollte im Rahmen der Überprüfung von Zusicherungen nicht eingeschränkt werden, um auch den Graphen einer CAF inspizieren zu können. Das hat gegebenenfalls zur Folge, daß die Bildung eines solchen Graphen auch außerhalb des normalen Programmablaufs angestoßen wird.
2. Wenn die Verwendung nullstelliger Superkombinatoren im Sinne der vorstehenden Aussage nicht eingeschränkt wird, muß die Unterscheidung zwischen Zusicherungs- und Programmknoten auch auf die Graphen der CAF's ausgedehnt werden. Der Grund hierfür ist bereits angeklungen: Der Graph einer CAF kann einen Redex repräsentieren bzw. einen Redex als Teilausdruck enthalten. Das belegt in einfacher Form die Definition

```
zero = 1 - 1
```

Vom Grundsatz her bedeutet die letztgenannte Forderung allerdings nicht, daß jeder nullstellige Superkombinator einen Programmausdruck repräsentiert. Bei der Auswertung des Ausdrucks

```
1 / assertF message ((/=) zero) 0
```

entspricht die CAF `zero` einem Teilausdruck, dessen Graph aus Zusicherungsknoten besteht.

Abschließend bleibt noch eine Frage zu klären: Was bedeutet die Forderung nach dem Erhalt der Semantik für ein Programm, das nicht im Einklang mit der zugrundeliegenden Spezifikation steht?

Sicher ist eines: Die Überprüfung einer Zusicherung darf weder die Ursache für die Nicht-Terminierung der Programmausführung sein, noch darf sie den Ergebniswert verfälschen, falls ein Fehler nicht erkannt wird - letzteres begründet sich unter anderem aus der Tatsache heraus, daß Prädikate, mit denen das Programmverhalten beschrieben wird, selbst fehlerhaft sein können. Beiden Anforderungen wird das hier vorgestellte Konzept gerecht.

Natürlich hat die Forderung nach dem Erhalt der Semantik in dem hier definierten Sinn aber keinen Bestand, wenn die Überprüfung einer Zusicherung Anlaß gibt, die Programmausführung abubrechen.

3.3.4 Aussagekraft einer semantikerhaltenden Überprüfung von Zusicherungen (Assertions)

Ein Werkzeug, das dem Zweck dient, während der Programmausführung automatisch Zusicherungen zu überprüfen, hat in einer nicht-strikten Programmiersprache wie Haskell aus den dargelegten Gründen eher beobachtenden Charakter. Dies sollte keineswegs zu der Annahme verleiten, seine Aussagekraft sei derart eingeschränkt, daß es keinen Nutzen erbringt. Orientiert man sich bei der Implementierung des Werkzeugs an den Grundsätzen, die der vorstehende Abschnitt bezeichnet, droht im Fall der Fakultätsfunktion weder eine Nicht-Terminierung der Berechnung, wenn die Überprüfung der Zusicherung erst einmal abgebrochen werden muß, weil ihr Argument einen Redex darstellt, noch bleibt im Fall der Funktion `qSort` ein Sortiervorgang unbeanstandet, dessen Ergebnis im Widerspruch zum Prädikat `isSorted` steht. Die Aussagekraft hat allerdings Grenzen. Ein Programmfehler bleibt nämlich unentdeckt, wenn die beiden folgenden Voraussetzungen erfüllt sind:

1. Der Aufruf der Funktion `assert (assertF)`, mit dessen Hilfe sich der Fehler eigentlich aufdecken ließe, muß ergebnislos abgebrochen werden, weil die Reduktion eines Ausdrucks `A` als unzulässig erachtet wird.
2. Der Ausdruck `A` erfährt im Rahmen des normalen Programmablaufs keine Reduktion.

Ein Beispiel soll diesen Sachverhalt veranschaulichen: Es sei eine Funktion `prim` zu definieren, die als erstes Argument eine ganze Zahl `n` erwartet, und als Ergebnis eine

Liste liefert, die aus den ersten n Primzahlen besteht. Mit Hilfe der aus dem Abschnitt 3.3.2 bereits bekannten Funktion `take` kann dies wie folgt geschehen:

```

prim n = take n (assertF msg allPrim (prim' [2..]))
  where
    msg = "Das Ergebnis der Funktion prim beinhaltet " ++
          "auch Nicht-Primzahlen"
    prim' :: [Int] -> [Int]
    prim' (x:xs)
      = x : [ y | y <- xs , y `rem` x /= 0 ]

-- !! Die Definition der Funktion prim' ist fehlerhaft !!
-- Ihre letzte Zeile muesste lauten:
--      = x : prim' [ y | y <- xs , y `rem` x /= 0 ]

allPrim :: [Int] -> Bool
allPrim [] = False
allPrim l@(x:xs) = all isPrim l

isPrim :: Int -> Bool
isPrim x = x == minDiv x

minDiv :: Integral a => a -> a
minDiv x = findDiv x 2

findDiv :: Integral a => a -> a -> a
findDiv x y
  | y^2 > x      = x
  | x `rem` y == 0 = y
  | otherwise    = findDiv x (y + 1)

```

Um das gewünschte Ergebnis zu berechnen, wird der Funktion `take` als erstes Argument das Argument der Funktion `prim` übergeben und als zweites Argument ein Ausdruck, der die unendliche Liste aller Primzahlen repräsentiert:

```
(assertF msg allPrim (prim' [2..]))
```

Zu diesem Ausdruck ist folgendes anzumerken:

1. Der Grundfehler des Ausdrucks ist die lokal definierte Funktion `prim'`, denn die Liste aller Primzahlen wird letztendlich nur durch den in ihm enthaltenen Teilausdruck

```
prim' [2..]
```

repräsentiert - die Definition der Funktion `prim'` implementiert im Zusammenspiel mit dem Argument, das ihr übergeben wird, ein altes Verfahren für die Bestimmung von Primzahlen: das "Sieb des Eratosthenes".

2. Die Anwendung der Funktion `assertF` soll sicherstellen, daß es sich bei den Elementen der Liste, die der Ausdruck `prim' [2..]` repräsentiert, tatsächlich um Primzahlen handelt.
3. Der gemäß der Kommentierung des Programmtextes in der Definition der Funktion `prim'` enthaltene Fehler bewirkt, daß aus der Liste `[2..]` nicht nur Primzahlen herausgefiltert werden, sondern auch alle natürlichen Zahlen, die nicht durch 2 teilbar sind.

Unter der Voraussetzung, daß man auf den Primzahltest verzichtet, liefert die Funktion `prim` trotz des Programmfehlers ein richtiges Ergebnis, wenn ihr Argument


```
xterm
test@hydra:~ > hugs +A Prim
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "Prim.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Prim.hs
Type !? for help
Prim> prim 4
[2,3,5,7]
Prim> prim 5
[2,3,5,7,9]

*** Program error: Assertion failed
*** Das Ergebnis der Funktion prim beinhaltet auch Nicht-Primzahlen

Prim> █
```


4 Modifizierung des Haskell-Interpreters Hugs für die Überprüfung von Zusicherungen (Assertions)

4.1 Überblick

Der dritte Abschnitt dieser Arbeit hat ein Konzept für die Überprüfung von Zusicherungen dargelegt, das es erlaubt, die Ausführung eines Haskell-Programms ohne Einflußnahme auf dessen Semantik zu kontrollieren. Die nachfolgenden Ausführungen beschreiben die Umsetzung dieses Konzepts in der Implementierung des Haskell-Interpreters Hugs, Version: Mai 1999 - im Weiteren kurz als Hugs-Anpassung bezeichnet.

Zwei einleitende Anmerkungen hierzu:

1. Die Funktion `assert` wird aufgrund der Ausführungen im Rahmen des Abschnitts 3.3.2 systemabhängig als eingebaute Funktion definiert. In diesem Zusammenhang gilt es zu beachten, daß der Name einer eingebauten Funktion in der Programmiersprache Haskell grundsätzlich mit dem Präfix `prim` beginnt. Um dem Anwender die Möglichkeit einzuräumen, bei der Einbindung von Zusicherungen in ein Programm weitgehend einheitlich bezeichnete Funktionen zu verwenden, beruht die Hugs-Anpassung deshalb auf folgender Festlegung:
 - Die zu definierende eingebaute Funktion wird als `primAssert` bezeichnet.
 - Die Funktion `assert` wird in `assertB` umbenannt und als Synonym für die Funktion `primAssert` verwendet. Ähnlich wie der Name `assertF` durch den letzten Buchstaben signalisiert, daß es sich bei dem zweiten Argument der Funktion `assertF` um eine Funktion handelt, soll der Name `assertB` durch den letzten Buchstaben signalisieren, daß es sich bei dem zweiten Argument der durch ihn bezeichneten Funktion um einen booleschen Ausdruck handelt.

Hier die entsprechenden Definitionen, soweit sie sich in der Programmiersprache Haskell zum Ausdruck bringen lassen:

```
primitive primAssert :: String -> Bool -> a -> a

assertB :: String -> Bool -> a -> a
assertB = primAssert

assertF :: String -> (a -> Bool) -> a -> a
assertF msg func exp = assertB msg (func exp) exp
```

2. Die weiteren Ausführungen gliedern sich in drei Hauptabschnitte:
 - Der Abschnitt 4.2 erläutert Grundlagen der Implementierung des Haskell-Interpreters Hugs, die für das Verständnis der daran vorgenommenen Änderungen von Bedeutung sind - insbesondere für das Studium des Anhangs A zu dieser Arbeit.
 - Im Rahmen des Abschnitts 4.3 wird dargelegt, inwieweit sich die Handhabung des modifizierten Haskell-Interpreters Hugs von der des ursprünglichen Interpreters unterscheidet. Dieser Abschnitt dient nicht zuletzt auch als Leitfaden, der dem Leser vermitteln soll, wie das Verhalten der Funktionen `assertB` und `assertF` beeinflusst werden kann - Stichwort: Übergang von der Testphase zum normalen Programmbetrieb -, und welche Beschränkungen bei der Verwendung dieser Funktionen beachtet werden müssen.

- Der Abschnitt 4.4 beschreibt schließlich Einzelheiten der Hugs-Anpassung. Da es den Rahmen dieser Arbeit übersteigen würde, alle Änderungen an der Implementierung des Haskell-Interpreters Hugs ausführlich darzustellen, beschränken sich die Ausführungen auf einige wesentliche Aspekte:
 - Die Unterscheidung zwischen Zusicherungsknoten und Programmknoten
 - Die Funktion `primAssert`
 - Die Wiederherstellung einer Zusicherung
 - Die Vorbereitung einer Auswertung
 - Die Modifikation der Fehlerbehandlung
 - Die Modifikation der Garbage Collection

Leser, die an detaillierteren Informationen interessiert sind, finden im Anhang A zu dieser Arbeit sämtliche Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick; und zwar gegliedert nach den Quellcode-Dateien, die sie betreffen. Diese Änderungen sind mit ausführlichen Kommentaren versehen worden.

4.2 Zentrale Aspekte und Komponenten der Implementierung des Haskell-Interpreters Hugs

4.2.1 Implementierungssprache

Der Haskell-Interpreter Hugs ist der Nachfolger von Gofer, einem experimentellen, funktionalen Programmiersystem. Aus diesem Umstand leitet sich auch der Name des Interpreters ab: Die Bezeichnung Hugs wurde ursprünglich als Abkürzung für "Haskell users' Gofer system" gewählt. Wie sein Vorgänger, ist der Haskell-Interpreter Hugs in der Programmiersprache C implementiert.

Für die Standardisierung der Programmiersprache C hat die vom American National Standards Institute (ANSI) vorgelegte Sprachdefinition einen wesentlichen Beitrag geleistet. Dieser Standard unterscheidet sich von früheren Versionen insbesondere im Hinblick auf die Deklaration und Definition von Funktionen [Kernighan 1990]. Er sieht im Gegensatz zum "alten Stil" beispielsweise vor, in der Deklaration einer Funktion den Typ der formalen Parameter anzugeben.

Beispiel:

Nach "altem Stil" wäre eine Funktion für die Berechnung der Fakultät folgendermaßen deklariert worden:

```
int fak();
```

Hier dieselbe Funktionsdeklaration nach ANSI-Standard:

```
int fak(int);
```

Die standardisierte Form der Deklaration macht es einem Compiler einfacher, Fehler bei der Anzahl oder dem Typ von Funktionsargumenten zu erkennen [Kernighan 1990]. Auch im Quellcode des Haskell-Interpreter Hugs wird von ihr Gebrauch gemacht. Dennoch kann er mit Compilern übersetzt werden, die dem ANSI-Standard in

dieser Hinsicht nicht genügen. Das gelingt durch die Zuhilfenahme eines Präprozessor-Makros `Args`:

```
#if HAVE_PROTOTYPES
#define Args(x) x
#else
#define Args(x) ()
#endif
```

Dem Makro `Args` wird in den Funktionsdeklarationen des Hugs-Quellcodes als Argument der Klammerausdruck übergeben, der die Parameter beschreibt. Aufgrund der bedingten Anweisungen, die seine Definition kontrollieren, wirkt sich dies wie folgt aus:

- Unterstützt der verwendete Compiler Prototypen - in diesem Fall ist für die Definition des Makros `Args` die Anweisung

```
#define Args(x) x
```

maßgebend -, werden Typangaben im Sinne des ANSI-Standards in die Funktionsdeklarationen einbezogen.

- Unterstützt der verwendete Compiler keine Prototypen - in diesem Fall ist für die Definition des Makros `Args` die Anweisung

```
#define Args(x) ()
```

maßgebend -, werden keine Typangaben in die Funktionsdeklarationen einbezogen.

Sämtliche im Anhang A zu dieser Arbeit dokumentierten Funktionen sind an die Verwendung des Makros `Args` angepaßt. Die Deklaration der Funktion `canReduce`, mit der im Rahmen der Überprüfung einer Zusicherung festgestellt wird, ob ein Ausdruck reduziert werden darf, lautet zum Beispiel:

```
static Bool local canReduce Args((Cell));
```

Im Zusammenhang mit der Handhabung von Fehlern spielt der C-Präprozessor ebenfalls eine besondere Rolle: Um Fehlermeldungen anzuzeigen und eingebaute Fehlerbehandlungs-routinen anzusteuern, werden im Hugs-Quellcode grundsätzlich Makros verwendet, die eigens für diese Zwecke in der Datei `errors.h` vordefiniert wurden. Einige der Makros sind aufgrund ihrer Parameter in der Lage, Zeilenangaben, Ausdrücke und Typinformationen in eine Fehlermeldung einzubinden. Der Code

```
ERRMSG(0) "Garbage collection fails to reclaim sufficient space"
EEND;
```

signalisiert dem Anwender zum Beispiel, daß für die Programmausführung nicht mehr genügend freier Speicher vorhanden ist, und bricht diese ab.

Um eine einheitliche Darstellung zu gewährleisten, werden die in der Datei `errors.h` definierten Makros auch für die im Anhang A zu dieser Arbeit dokumentierten Änderungen an der Implementierung des Haskell-Interpreters Hugs verwendet. Ein Beispiel: Wenn es erforderlich ist, die Überprüfung einer Zusicherung ergebnislos abzubrechen, werden mit Hilfe der in der Datei `storage.c` definierten

Funktion `saveDebugInfo` bestimmte Informationen gespeichert, die es erlauben, diese Überprüfung zu einem späteren Zeitpunkt erneut anzustoßen. Reichen die vorhandenen Kapazitäten hierfür nicht, wird der Code

```
ERRMSG(0) "Cannot allocate storage space " ETHEN
ERRTEXT  "for \"Debugging with Assertions\".\n" ETHEN
ERRTEXT  "Assignment: Check List"
EEND;
```

ausgeführt.

4.2.2 Programmstruktur

Die Hauptkomponenten des Haskell-Interpreters Hugs verteilen sich auf separate Quelldateien:

- Die Datei `input.c` beinhaltet Funktionen für die lexikalische Analyse und das Parsen.
- Die Datei `static.c` beinhaltet Funktionen für die statische Analyse.
- Die Datei `type.c` beinhaltet Funktionen für die Typüberprüfung. Zur Erinnerung: Haskell ist eine statisch stark getypte Programmiersprache.
- Die Datei `compiler.c` beinhaltet Funktionen für die Übersetzung des typgeprüften Codes in eine Menge von Superkombinator-Definitionen (vgl. Abschnitt 2.4.1).
- Die Datei `machine.c` beinhaltet Funktionen, die die Programmausführung auf einer abstrakten Maschine simulieren.

Alle Komponenten des Systems tragen für die Initialisierung und Pflege der Variablen und Datenstrukturen, von denen sie abhängen, selbst die Verantwortung. Zu diesem Zweck beinhalten sie eigenständige Kontrollfunktionen, die in der Lage sind, verschiedene Steuersignale zu verarbeiten. Im Fall der statischen Analyse ist das beispielsweise die Funktion `staticAnalysis`.

Darüber hinaus gibt es eine übergeordnete Kontrollfunktion `everybody`, mit deren Hilfe sämtlichen Kontrollfunktionen gleichzeitig ein Signal übermittelt werden kann:

```
Void everybody(what)
Int what; {
    machdep(what);
    storage(what);
    substitution(what);
    ...
    builtIn(what);
    controlFuns(what);
    plugins(what);
}
```

Der Aufruf `everybody(MARK)` veranlaßt zum Beispiel die Kennzeichnung der Speicherbereiche, die durch die Garbage Collection noch nicht für eine neue Belegung freigegeben werden dürfen.

Wie die Definition der Funktion `everybody` erkennen läßt, ist die obige Aufzählung der Systemkomponenten keineswegs vollständig. Unerwähnt geblieben sind

beispielsweise die Speicherverwaltung, deren Implementierung sich auf die Dateien `storage.c` und `storage.h` verteilt, sowie die Definition der Schnittstelle zum Benutzer, die in der Datei `hugs.c` enthalten ist.

Es würde den Rahmen dieser Arbeit übersteigen, auf alle Komponenten des Haskell-Interpreters Hugs einzugehen. In den beiden nachfolgenden Abschnitten werden daher lediglich einige Aspekte der Speicherverwaltung und des Auswertungsmechanismus näher betrachtet, die für den Untersuchungsgegenstand der vorliegenden Arbeit von Bedeutung sind.

4.2.3 Speicherverwaltung

Im Rahmen der Implementierung einer verzögert auswertenden Programmiersprache stellt sich unter anderem die Frage, wie ein Graph dargestellt werden soll. Der Haskell-Interpreter Hugs verwendet hierfür einen Heap, der sich aus zwei separaten Vektoren `heapFst` und `heapSnd` mit jeweils `heapSize` Elementen vom Datentyp `Cell` zusammensetzt. Und zwar in der Form, daß das *i*-te Element des Vektors `heapFst` zusammen mit dem *i*-ten Element des Vektors `heapSnd` das *i*-te Element des Heaps bildet.

Gemäß den Vereinbarungen

```
typedef int Int;
typedef Int Cell;
```

in den Dateien `prelude.h` und `storage.h` wird der Datentyp `Cell` als Synonym für den Datentyp `int` der Programmiersprache C verwendet. Jedes Element des Heaps besteht demnach aus zwei Komponenten, die einen ganzzahligen Wert repräsentieren, und in der "Hugs-Terminologie" ein *Pair* (*Paar*) darstellen. Letzteres dokumentiert sich insbesondere in der Namensgebung der folgenden Funktionsdefinition (vgl. Datei `storage.c`):

```
Cell pair(l,r)
Cell l, r; {
    ...
    fst(c)  = l;
    snd(c)  = r;
    ...
}
```

Die Funktion `pair` erwartet zwei Argumente `l` und `r` vom Typ `Cell` und konstruiert daraus ein Paar, indem der Wert `l` mit Hilfe des Selektors `fst` in das erste freie Element des Vektors `heapFst` eingetragen wird, und der Wert `r` mit Hilfe des Selektors `snd` in das korrespondierende Element des Vektors `heapSnd` eingetragen wird. Sollte der Heap vollständig belegt sein, veranlaßt die Funktion `pair` zuvor die Durchführung einer Garbage Collection.

Die Aufgabe der Garbage Collection besteht darin, sämtliche Elemente des Heaps zu ermitteln, deren Inhalt für die weitere Programmausführung nicht mehr von Bedeutung ist, und in einer verketteten Liste - der sogenannten `freeList` - zusammenzufassen. Nach Beendigung dieses Vorgangs, den die Speicherverwaltung mit Hilfe einer Technik realisiert, die als *Mark-scan Garbage Collection* bekannt ist [Peyton Jones

1987], greift die Funktion `pair` auf das erste Element der `freeList` zu und belegt es mit den neuen Werten.

Insgesamt betrachtet dient der Heap der Repräsentation sehr verschiedenartiger Objekte - nicht nur im Rahmen einer Auswertung. Als Beispiele seien Applikationsknoten, die Namen von Variablen und Operatoren, String-Literale, Wildcard-Pattern, Indirektionsknoten und Tupel-Konstrukturen erwähnt. Letztendlich liegt deren Darstellung die Idee zugrunde, Objekte unterschiedlicher Art - Objekte vom Datentyp `Cell` - mit Hilfe ganzer Zahlen aus unterschiedlichen Bereichen zu repräsentieren. Die Grenzen der Bereiche legt eine Reihe von `#define`-Anweisungen in der Datei `storage.h` fest:

```
#define NIL      0
#define TAGMIN  1
#define BCSTAG  30
#define SPECMIN 101
#define TUPMIN  201
...
#define NAMEMIN (TYCMIN+NUM_TYCON)
#define INSTMIN (NAMEMIN+NUM_NAME)
...
```

Im Sinne dieser Systematik repräsentieren zum Beispiel

- ganze Zahlen n mit der Eigenschaft $n < \text{NIL}$ Paare - in den weiteren Ausführungen wird eine ganze Zahl n mit der Eigenschaft $-\text{heapSize} \leq n < \text{NIL}$ als Heap-Adresse bezeichnet -,
- ganze Zahlen n mit der Eigenschaft $\text{TAGMIN} \leq n < \text{BCSTAG}$ sogenannte *Tags*, die als Identifikationsmerkmal eines Paares in den Vektor `heapFst` eingetragen werden, wenn das Paar ein sogenanntes *Boxed Object* darstellt (vgl. [Peyton Jones 1987]), und
- ganze Zahlen n mit der Eigenschaft $\text{NAMEMIN} \leq n < \text{INSTMIN}$ Funktionsnamen, kurz: Namen.

Aufgrund der Tatsache, daß negative ganze Zahlen Paare repräsentieren, beruht die Implementierung der Selektoren `fst` und `snd` darauf, in den Vektoren `heapFst` und `heapSnd` rückwärts zu indizieren:

```
Void storage(what)
Int what; {
  Int i;

  switch (what) {
    ...
    case INSTALL : heapFst = heapAlloc(heapSize);
                  heapSnd = heapAlloc(heapSize);
    ...
                  heapTopFst = heapFst + heapSize;
                  heapTopSnd = heapSnd + heapSize;
    ...
  }
}

#define fst(c)      heapTopFst[c]
#define snd(c)      heapTopSnd[c]
```

Neben diesen Selektoren und der Funktion `pair` stehen für die Verwendung von `Cell`-Werten noch eine ganze Reihe weiterer Makros und Funktionen zur Verfügung. Darunter zum Beispiel die in der Datei `storage.h` als

```
#define ap(f,x) pair(f,x)
#define fun(c)  fst(c)
#define arg(c)  snd(c)
```

definierten Makros `ap`, `fun` und `arg` für die Handhabung von Applikationsknoten sowie die in der Datei `storage.c` definierte Funktion `whatIs`.

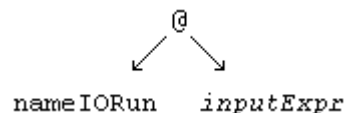
Die Funktion `whatIs` erwartet als Argument einen `Cell`-Wert `c` und liefert als Ergebnis einen Schlüssel, der darüber Auskunft gibt, welche Art von Objekt `c` repräsentiert. Handelt es sich bei dem Argument zum Beispiel um einen `Cell`-Wert, der einen Applikationsknoten, einen Indirektionsknoten bzw. einen Namen repräsentiert, lautet das Ergebnis `AP`, `INDIRECT` bzw. `NAME`.

Weitergehende Informationen über die Speicherverwaltung können dem Research Report "The implementation of the Gofer functional programming system" entnommen werden [Jones 1994]. Viele der darin beschriebenen Grundsätze besitzen auch für den Haskell-Interpreter Hugs noch Gültigkeit.

4.2.4 Auswertungsmechanismus

Der in der Datei `machine.c` enthaltene Quellcode des Haskell-Interpreters Hugs implementiert für die Programmausführung eine abstrakte Maschine, die sich eng an die Grundprinzipien der G-Maschine anlehnt, die im Abschnitt 2.4.2 vorgestellt wurde. Für das Verständnis der weiteren Ausführungen ist es deshalb nicht erforderlich, die Funktionsweise der abstrakten Maschine im Überblick darzustellen. Die Aufmerksamkeit kann vielmehr einigen Details des Auswertungsmechanismus zugewandt werden:

1. Um eine Auswertung durchzuführen, konstruiert die in der Datei `hugs.c` definierte Funktion `evaluator` zunächst den Anfangsgraphen, der grundsätzlich eine Applikation der Funktion `nameIORun` repräsentiert - `nameIORun` ist der systemintern verwendete Name für die in der [Prelude] definierte Funktion `hugsIORun`):



2. Um den durch den Anfangsgraphen repräsentierten Ausdruck in WHNF zu transformieren, wird der Funktion `eval` die Heap-Adresse seines obersten Applikationsknotens als Argument übergeben. Die Funktion `eval` wird in der Datei `machine.c` wie folgt definiert:

Kapitel 4 – Modifizierung des Haskell-Interpreters Hugs für die Überprüfung von Zusicherungen (Assertions)

```

Void eval(n)                                /* Graph reduction evaluator */
Cell n; {
    StackPtr base = sp;
    Int ar;

    ....

unw:switch (whatIs(n)) {                    /* unwind spine of application */

    case AP : push(n);
              n = fun(n);
              goto unw;

    case INDIRECT : n = arg(n);
                    allowBreak();
                    goto unw;

    case NAME : allowBreak();
                ...
                if (!isCfun(n) && (ar=name(n).arity)<=(sp-base)) {
*** 1 ***                                if (ar>0) { /* fn with args*/
                                        StackPtr root;

                                        push(NIL); /* rearrange */
                                        ...
                                        if (name(n).primDef) /* reduce */
                                            (*name(n).primDef)(root);
                                        else
                                            run(name(n).code,root);
                                        ...

                                        sp = root; /* continue... */
                                        n = pop();
                }
*** 2 ***                                else { /* CAF */
                                        if (isNull(name(n).defn)) { /* build CAF */
                                            StackPtr root = sp;
                                            push(n); /* save CAF */
                                            ...
                                            if (name(n).primDef)
                                                (*name(n).primDef)(sp);
                                            else
                                                run(name(n).code,sp);
                                            ...
                                            name(n).defn = top();
                                            sp = root; /* drop CAF */
                                        }
                                        n = name(n).defn; /*already built*/
                                        if (sp>base)
                                            fun(top()) = n;
                }
                ...
                goto unw;
            }
        }
        break;

    ...
}

whnfHead = n; /* rearrange components of term on */
...
}

```

Die Funktion `eval` implementiert den im Abschnitt 2.4.2 beschriebenen Reduktionsvorgang. Sollte es erforderlich sein, kann sie nach der Übergabe des Anfangsgraphen weitere Male aufgerufen werden - etwa um Funktionsargumente auszuwerten.

3. Zu der angegebenen Definition der Funktion `eval` sei folgendes angemerkt:

- Die in der mit der Zeichenfolge `*** 1 ***` markierten Programmzeile enthaltene Bedingung

```
if (ar>0)
```

wird abgefragt, wenn die `switch`-Anweisung, zum Beispiel nach Beendigung des Unwind-Prozesses, einen Superkombinator `f` vorgefunden hat, dessen Stelligkeit - ausgedrückt in der Variablen `ar` - nicht größer als die Anzahl der auf dem Stack vorhandenen Argumente ist. Sollte die Bedingung erfüllt sein, handelt es sich bei dem Superkombinator `f` nicht um eine CAF, der Stack wird im Sinne der Ausführungen des Abschnitts 2.4.2 umgeordnet, und die Funktion `eval` ruft auf der Grundlage des folgenden `if-else`-Konstrukts den Code für `f` auf:

```
if (name(n).primDef)           f ist eine eingebaute Funktion
    (*name(n).primDef)(root);
else                            f ist keine eingebaute Funktion
    run(name(n).code,root);
```

Der "Stack Pointer" `root` bezeichnet bei einem solchen Aufruf stets das Stack-Element, das den Zeiger auf den Wurzelknoten der zu reduzierenden Applikation beinhaltet. Nach Ausführung des Codes wird dieser Knoten durch die Anweisungsfolge

```
sp = root;                      /* continue... */
n  = pop();
```

vom Stack genommen und erneut dem Unwind-Mechanismus übergeben, so daß der auszuwertende Ausdruck weiter reduziert wird, falls er noch nicht in WHNF ist - zur Erläuterung: `sp` stellt einen "Stack Pointer" dar, der stets das oberste Stack-Element bezeichnet.

- Wenn die Bedingung

```
if (ar > 0)
```

nicht erfüllt ist, hat die `switch`-Anweisung einen nullstelligen Superkombinator `f` vorgefunden. Wie im Falle jedes anderen Namens, sind über `f` systemintern in einer Struktur vom Typ `struct strName` bestimmte Informationen verfügbar (vgl. Datei `storage.h`):

```
struct strName {
    Text text;
    Int  line;
    ...
    Int  arity;
    ...
    Cell defn;
    ...
};
```

Die Komponente `defn` gibt zum Beispiel darüber Auskunft, ob der Code für den Superkombinator `f` schon einmal aufgerufen wurde - hat sie den Wert `NIL`, ist das noch nicht der Fall gewesen.

Diese Information spielt für den mit der Zeichenfolge `*** 2 ***` kenntlich gemachten `else`-Teil in der Definition der Funktion `eval` eine entscheidende Rolle: Wenn die Bedingung

```
if (isNull(name(n).defn)
```

erfüllt ist, hat die Komponente `defn` in der "Namens"-Struktur von `f` den Wert `NIL`, und die Funktion `eval` ruft auf der Grundlage des folgenden `if-else`-Konstrukts den Code für `f` auf:

```
if (name(n).primDef)           f ist eine eingebaute Funktion
    (*name(n).primDef)(sp);
else                            f ist keine eingebaute Funktion
    run(name(n).code, sp);
```

Nach Ausführung des Codes wird dieser Komponente durch die Anweisung

```
name(n).defn = top();
```

der `Cell`-Wert zugewiesen, den das oberste Stack-Element beinhaltet, so daß sie das Resultat der Auswertung von `f` widerspiegelt, etwa in Gestalt der Heap-Adresse eines Applikationsknotens.

Hat die Komponente `defn` in der "Namens"-Struktur von `f` dagegen einen von `NIL` verschiedenen Wert, ruft die Funktion `eval` den Code für `f` nicht mehr auf; es wird unmittelbar die Anweisung

```
n = name(n).defn;
```

ausgeführt, mit der der Auswertungsmechanismus auf die vormals konstruierte Instanz zugreift. Das heißt, der Haskell-Interpreter Hugs wurde im Sinne der Ausführungen des Abschnitts 3.3.3 so konzipiert, daß von dem Graphen einer CAF keine unnötigen Kopien erstellt werden.

4.3 Der modifizierte Haskell-Interpreter Hugs - Optionen und Restriktionen bei der Überprüfung von Zusicherungen (Assertions)

Das Verhalten des Haskell-Interpreters Hugs kann mit Hilfe mehrerer Optionen bedarfsgerecht angepaßt werden. In der Regel handelt es sich bei diesen Optionen um einfache Toggles; das heißt, sie lassen sich entweder aktivieren oder deaktivieren - und zwar dergestalt, daß der entsprechenden Option ein Pluszeichen bzw. ein Minuszeichen vorangestellt wird. Die gewünschten Einstellungen können dabei auf verschiedenen Wegen festgelegt werden, etwa beim Start des Interpreters über Argumente aus der Kommandozeile oder nach dem Start des Interpreters mit Hilfe eines eigens dafür vorgesehenen Befehls, des `:set`-Befehls.

Beispiel:

Die Anweisung

```
hugs +s
```

startet den Haskell-Interpreter Hugs und aktiviert die Option `s`, die ihn veranlaßt, nach Beendigung einer Auswertung nicht nur das ermittelte Resultat anzuzeigen, sondern unter anderem auch die Zahl der erforderlich gewesen Reduktionen. Durch die Eingabe des Befehls

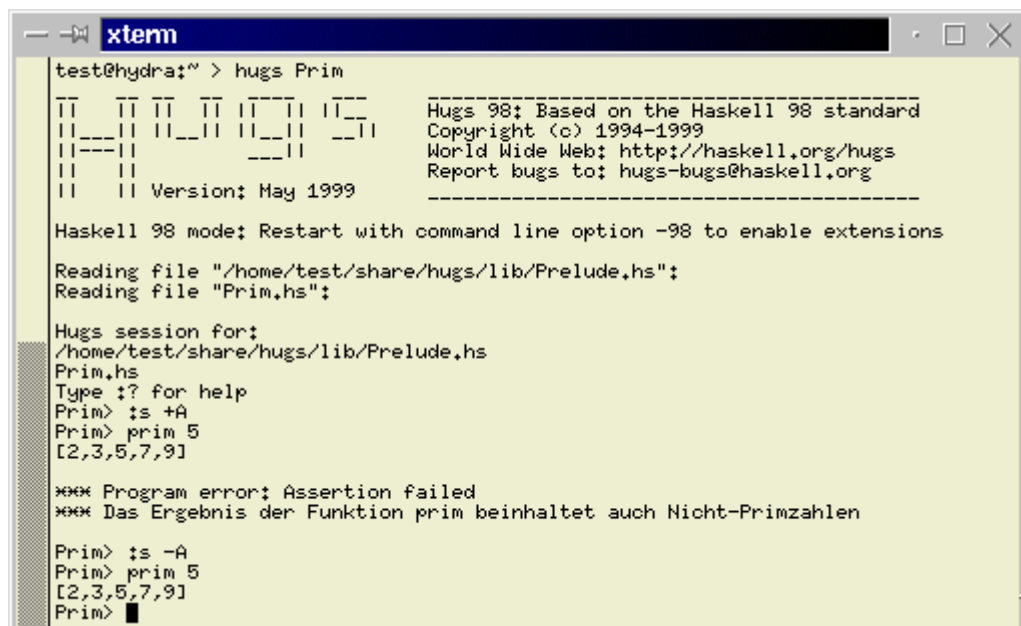
```
:set -s (oder kurz: :s -s)
```

am Hugs-Prompt kann dies wieder rückgängig gemacht werden.

Damit der Anwender die Möglichkeit hat, die Überprüfung von Zusicherungen zu unterbinden, sobald die Testphase für ein Programm beendet ist, wird ihm durch die Hugs-Anpassung eine weitere Option zur Verfügung gestellt - die Option `A`:

- Wenn die Option `A` deaktiviert ist - dies ist die Standard-Einstellung, mit der der modifizierte Haskell-Interpreter Hugs gestartet wird - verhält sich die Funktion `primAssert` wie eine Projektion auf das dritte Argument.
- Wenn die Option `A` aktiviert ist, überprüft die Funktion `primAssert` die Programmausführung, indem das Eintreffen der vom Anwender definierten Bedingungen beobachtet wird.

Die nachfolgende Abbildung veranschaulicht die Wirkungsweise der Option `A` anhand zweier Auswertungen für die aus dem Abschnitt 3.3.4 bekannte - fehlerhaft definierte - Funktion `prim`:



```
xterm
test@hydra:~ > hugs Prim
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode; Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "Prim.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Prim.hs
Type :? for help
Prim> :s +A
Prim> prim 5
[2,3,5,7,9]

*** Program error: Assertion failed
*** Das Ergebnis der Funktion prim beinhaltet auch Nicht-Primzahlen

Prim> :s -A
Prim> prim 5
[2,3,5,7,9]
Prim>
```

Systemintern bewirkt das Aktivieren bzw. Deaktivieren der Option `A`, daß sich der Wert der Variablen

```
debugging
```

verändert:

- Wenn die Option `A` deaktiviert ist, hat die Variable `debugging` den Wert `FALSE` - in der Implementierung des Haskell-Interpreters Hugs gleichbedeutend mit dem Wert `0`.

- Wenn die Option `A` aktiviert ist, hat die Variable `debugging` den Wert `TRUE` - in der Implementierung des Haskell-Interpreters Hugs gleichbedeutend mit dem Wert `1`.

Dementsprechend beinhalten viele der im Anhang A zu dieser Arbeit dokumentierten Funktionen und Anweisungsfolgen `if`-Konstrukte der Form

```
if (debugging)
    statement
```

oder `if-else`-Konstrukte der Form

```
if (debugging)
    statement_1
else
    statement_2
```

Die Option `A` steht allerdings nur zur Verfügung, wenn der Name

```
DEBUGGING_WITH_ASSERTIONS
```

in der Konfigurations-Datei `options.h.in` mit dem Wert `1` definiert wird, da die Hugs-Anpassung die Fähigkeit des C-Präprozessors zur bedingten Übersetzung ausnutzt. Infolgedessen werden die für die Kontrolle des Programmablaufs erforderlichen Änderungen am C-Quellcode des Interpreters fast ausnahmslos mit einer Bedingung der Form

```
#if DEBUGGING_WITH_ASSERTIONS
...
#endif
```

umgeben. So auch die Vereinbarung der Variablen `debugging` in der Datei `machine.c`:

```
#if DEBUGGING_WITH_ASSERTIONS
Bool  debugging      = FALSE;           /* TRUE => Die Funktion primAssert */
                                           /*       verhaelt sich nicht      */
                                           /*       ausschliesslich wie     */
                                           /*       eine Projektion auf das  */
                                           /*       dritte Argument         */
...
#endif
```

Eine der wenigen Ausnahmen stellt die `#define`-Anweisung

```
#define returnThdArg() updateRoot(primArg(1))
```

dar, die der Datei `builtin.c` hinzugefügt wird. Sie definiert einen Makro, mit dessen Hilfe die Funktion `primAssert` ihr drittes Argument als Ergebnis liefert, falls kein Grund für einen Programmabbruch vorhanden ist. Der Verzicht auf eine bedingte Übersetzung erklärt sich in diesem Zusammenhang aus der Tatsache, daß die Funktion `primAssert` auch definiert sein muß - und damit ein Ergebnis liefern muß - wenn der Name

```
DEBUGGING_WITH_ASSERTIONS
```

in der Konfigurations-Datei `options.h.in` mit dem Wert `0` definiert wird, denn die Hugs-Anpassung bindet den Haskell-Quellcode

```
primitive primAssert :: String -> Bool -> a -> a

assertB :: String -> Bool -> a -> a
assertB = primAssert

assertF :: String -> (a -> Bool) -> a -> a
assertF msg func exp = assertB msg (func exp) exp
```

in die [Hugs-Prelude] ein, um einen automatischen Import der Funktionen `assertB` und `assertF` in alle Module zu gewährleisten.

Die Grundstruktur der Definition der Funktion `primAssert` liegt damit auf der Hand:

```
primFun(primAssert) {
  #if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
      ...
    }
    else
      returnThdArg();
  #else
    returnThdArg();
  #endif
}
```

Zurück zu den Optionen: Die Option `A` wird durch eine Option `d` ergänzt, die für die Handhabung der CAF's von Bedeutung ist.

Der vorstehende Abschnitt hat gezeigt, daß der Code für einen nullstelligen Superkombinator im Rahmen einer Auswertung nur einmal ausgeführt werden muß. Die Implementierung des Haskell-Interpreters Hugs leistet sogar noch mehr - die durch die Ausführung des Codes konstruierte Instanz bleibt für wiederholte Zugriffe so lange erhalten, bis das Skript, dem der entsprechende nullstellige Superkombinator zuzuordnen ist, nach einer Änderung erneut geladen wird oder etwa durch das Laden eines anderen Skripts seine Gültigkeit verliert. Sie überdauert also unter Umständen mehrere Auswertungen.

Im Hinblick auf die Aussagekraft von Zusicherungen ist dieser Sachverhalt problematisch. Um das zu veranschaulichen, seien folgende Definitionen betrachtet:

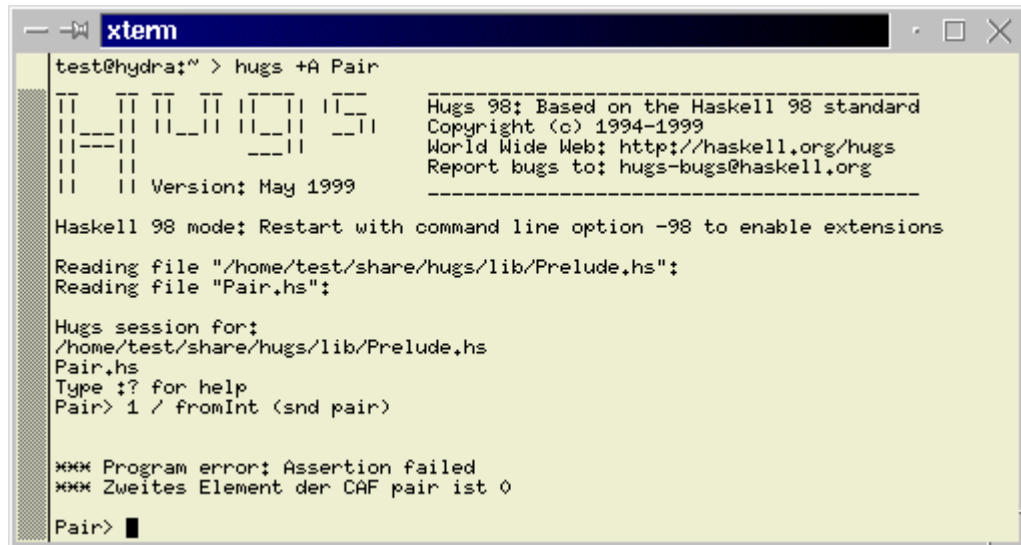
```
pair :: (Int, Int)
pair = assertF msg (not . isEqSnd) (1, 0)
  where
    msg = "Zweites Element der CAF pair ist 0"

isEqSnd :: (Int, Int) -> Bool
isEqSnd (_, 0) = True
isEqSnd (_, _) = False
```

Wenn der Graph der CAF `pair` aufgrund der Auswertung des Ausdrucks

```
1 / fromInt (snd pair)
```

gebildet wird, bricht die damit einhergehende Anwendung der Funktion `assertF` die Programmausführung ab:



```
test@hydra:~ > hugs +A Pair
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "Pair.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Pair.hs
Type :? for help
Pair> 1 / fromInt (snd pair)

*** Program error: Assertion failed
*** Zweites Element der CAF pair ist 0

Pair> █
```

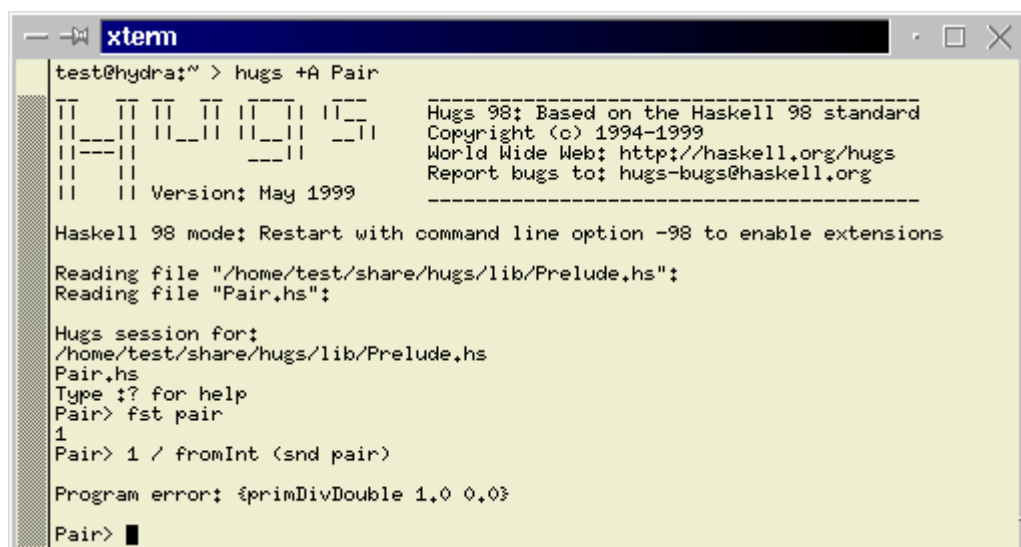
Anders verhält sich die Situation jedoch, wenn der Graph der CAF `pair` aufgrund der Auswertung des Ausdrucks

```
fst pair
```

gebildet wird, und der Ausdruck

```
1 / fromInt (snd pair)
```

erst im Anschluß daran eine Reduktion erfährt:



```
test@hydra:~ > hugs +A Pair
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "Pair.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Pair.hs
Type :? for help
Pair> fst pair
1
Pair> 1 / fromInt (snd pair)

Program error: {primDivDouble 1.0 0.0}

Pair> █
```

Die Erklärung für die unterschiedlichen Ergebnisse ist relativ einfach: Wenn der Graph der CAF `pair` aufgrund der Auswertung des Ausdrucks

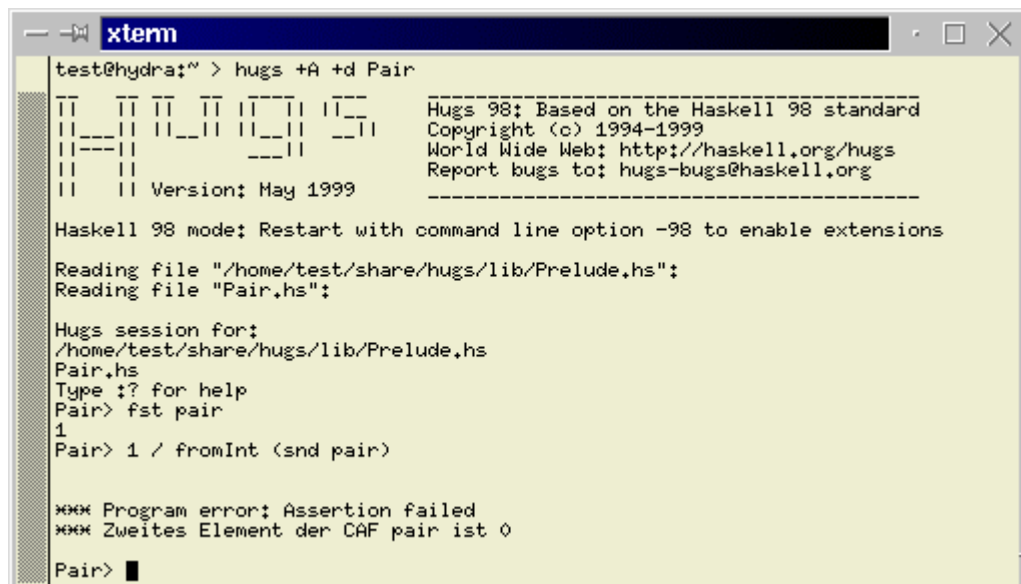
```
fst pair
```

gebildet wird, muß die Überprüfung der in ihrer Definition eingebundenen Zusicherung ergebnislos abgebrochen werden, weil das zweite Element des Paares (1, 0) einen Programmausdruck darstellt. Die Hugs-Anpassung sieht als Reaktion darauf vor, die Applikation der Funktion `primAssert` mit einem Indirektionsknoten zu überschreiben, der als Zeiger auf deren drittes Argument fungiert. Infolgedessen ist die Anwendung der Funktion `assertF` in dem Graphen der CAF `pair` nicht mehr präsent, wenn der Ausdruck

```
1 / fromInt (snd pair)
```

reduziert wird, so daß auch nicht mehr überprüft wird, ob das zweite Element des Paares (1, 0) die Zahl 0 ist.

Um diesem Problem entgegenzuwirken, kann der Haskell-Interpreter Hugs durch die Aktivierung der Option `d` veranlaßt werden, vor jeder neuen Auswertung die Graphen der nullstelligen Superkombinatoren zu löschen - in den weiteren Ausführungen wird dieser Vorgang als "Löschen der CAF-Definitionen" bezeichnet:



```
xterm
test@hydra:~$ hugs +A +d Pair
-----
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
-----
Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/home/test/share/hugs/lib/Prelude.hs":
Reading file "Pair.hs":

Hugs session for:
/home/test/share/hugs/lib/Prelude.hs
Pair.hs
Type :? for help
Pair> fst pair
1
Pair> 1 / fromInt (snd pair)

*** Program error: Assertion failed
*** Zweites Element der CAF pair ist 0

Pair> █
```

Systemintern bewirkt das Aktivieren bzw. Deaktivieren der Option `d`, daß sich der Wert der Variablen

```
deleteCAF
```

verändert: Sie hat den Wert `FALSE`, wenn die Option `d` deaktiviert ist, ansonsten den Wert `TRUE`.

Diese Variable wird ebenfalls in der Datei `machine.c` vereinbart:

```
#if DEBUGGING_WITH_ASSERTIONS
...
Bool  deleteCAF      = FALSE;          /* TRUE => Vor einer neuen      */
                                           /* Auswertung werden die      */
                                           /* CAF-Definitionen          */
                                           /* gelöscht                  */
...
#endif
```

Im Umgang mit den Optionen `A` und `d` ist folgendes zu beachten:

1. Die Aktivierung der Option `d` wird nur wirksam, wenn gleichzeitig die Option `A` aktiviert wird, bzw. die Option `A` bereits zu einem früheren Zeitpunkt aktiviert wurde. Diese Einschränkung stellt die normale Handhabung der nullstelligen Superkombinatoren sicher, wenn keine Zusicherungen überprüft werden.
2. Die Hugs-Anpassung sieht vor, nach der Aktivierung bzw. Deaktivierung der Option `A` sowie nach der Aktivierung bzw. Deaktivierung der Option `d` sämtliche CAF-Definitionen zu löschen - im Fall der Option `d` allerdings nur unter der Voraussetzung, daß die Option `A` aktiviert ist. Hierdurch werden einerseits die Auswertungen unter verschiedenen Optionseinstellungen klar voneinander abgegrenzt, andererseits werden Probleme im Zusammenhang mit nullstelligen Superkombinatoren vermieden, denen der Anwender mit Hilfe der Option `d` nicht begegnen kann. Folgendes Beispiel, das auf die vorstehend betrachteten Auswertungen Bezug nimmt, soll dies veranschaulichen: Wenn der Graph der CAF `pair`

- aufgrund der Auswertung des Ausdrucks

```
fst pair
```

gebildet wird, während die Variable `debugging` den Wert `FALSE` besitzt,

- anschließend die Option `A` aktiviert wird, ohne dafür Sorge zu tragen, daß vor der nächsten Auswertung die CAF-Definitionen gelöscht werden, und
- dann der Ausdruck

```
1 / fromInt (snd pair)
```

eine Reduktion erfährt,

ist die Anwendung der Funktion `assertF` in dem Graphen der CAF `pair` nicht mehr präsent, wenn die Division ausgeführt wird. Welche Konsequenzen sich daraus ergeben, ist bereits dargelegt worden.

Um sicherzustellen, daß die Überprüfung einer Zusicherung nach Aktivierung der Option `A` verlässliche Resultate liefert, muß die Funktion `assertB` auf mindestens zwei Argumente angewendet werden. Gleiches gilt für die Funktion `assertF`. Das heißt, ein Ausdruck

expression

darf in einem Programmtext durch einen Ausdruck der Form

- `assertB message condition expression`,
- `assertF message function expression`, oder
- `assert expression`

ersetzt werden, wobei der Bezeichner `assert` einen Ausdruck der Form

- `assertB message condition` bzw.
- `assertF message function`

repräsentieren muß.

Beispiel:

Die Funktion `qSort` wurde im Rahmen der Ausführungen des Abschnitts 3.3.1 folgendermaßen definiert:

```
qSort :: Ord a => [a] -> [a]
qSort x = assertF msg (isSorted x) (qSort' x)
  where
    msg = "Funktion qSort fehlerhaft definiert"
    qSort' :: Ord a => [a] -> [a]
    qSort' [] = []
    qSort' (x:xs) = qSort' [y | y<-xs, y<=x] ++
                    [x] ++
                    qSort' [y | y<-xs, y>x]
```

Zulässig wäre auch diese Definition:

```
qSort :: Ord a => [a] -> [a]
qSort x = assert (qSort' x)
  where
    assert = assertF msg (isSorted x)
    msg = "Funktion qSort fehlerhaft definiert"
    qSort' :: Ord a => [a] -> [a]
    qSort' [] = []
    qSort' (x:xs) = qSort' [y | y<-xs, y<=x] ++
                    [x] ++
                    qSort' [y | y<-xs, y>x]
```

Aus Gründen der Praktikabilität sieht die Hugs-Anpassung nicht vor, die Überprüfung einer Zusicherung zeitlich zu limitieren oder die Reduktionen, die sie anstoßen darf, zahlenmäßig zu beschränken.

4.4 Änderungen an der Implementierung des Haskell-Interpreters Hugs im Detail

4.4.1 Unterscheidung zwischen Zusicherungsknoten und Programmknoten

Für die Unterscheidung zwischen Zusicherungsknoten und Programmknoten sieht die Hugs-Anpassung die Verwendung von Markierungen vor:

- Ein Programmknoten wird mit der Markierung `NON_ASSERTION` versehen.
- Ein Zusicherungsknoten wird mit der Markierung `ASSERTION` versehen, wenn es sich um den Wurzelknoten einer Anwendung der Funktion `assertB`, `assertF` bzw. `primAssert` handelt. Alle übrigen Zusicherungsknoten bleiben unmarkiert.

Gesetzt werden die Markierungen mit Hilfe der Funktion `markExpression` (definiert in der Datei `storage.c`):

Kapitel 4 – Modifizierung des Haskell-Interpreters Hugs für die Überprüfung von Zusicherungen (Assertions)

```
Void markExpression(n)
Cell n; { /* Cell-Wert */
  Cell left;
  Cell right;

  if (!isGenPair(n)) {
    if (isCAF(n))
      restrictCAF(n);
    else
      return;
  }
  else {
    if (debugMark(n) == NOT_MARKED || debugMark(n) == CAF) {
      left = fst(n);
      right = snd(n);
      if (isAssertion(n)) {
        debugMark(n) = ASSERTION;
        markExpression(right);
      }
      else if (isGenPair(left)) {
        debugMark(n) = NON_ASSERTION;
        markExpression(left);
        markExpression(right);
      }
      else if (isNull(left) || left == INDIRECT) {
        debugMark(n) = NON_ASSERTION;
        markExpression(right);
      }
      else {
        debugMark(n) = NON_ASSERTION;
        if (left >= BCSTAG) {
          if (isCAF(left))
            restrictCAF(left);
          markExpression(right);
        }
        else
          return;
      }
    }
    else
      return;
  }
}
```

Anmerkungen zu der Funktion `markExpression`:

1. Als Argument erwartet die Funktion `markExpression` einen beliebigen Cell-Wert; zum Beispiel einen Wert, der einen Applikationsknoten repräsentiert.
2. Der Graph eines auszuwertenden Ausdrucks wird mit Hilfe der Funktion `markExpression` noch vor dessen Übergabe an die Funktion `eval` markiert.
3. Die Markierungen der Knoten werden mit Hilfe des Makros `debugMark` in einen Vektor `heapFourth` eingetragen, der genau wie die Vektoren `heapFst` und `heapSnd` definiert ist. Dementsprechend lautet seine Definition:

```
#define debugMark(n) heapTopFourth[n]
```

4. Wenn das Argument der Funktion `markExpression` ein Paar repräsentiert, wird mit Hilfe der Funktion `isAssertion` zunächst überprüft, ob es sich dabei um den Wurzelknoten A einer Anwendung der Funktion `assertB`, `assertF` bzw. `primAssert` handelt. Sollte das der Fall sein, erstreckt sich die weitere Markierung nur auf den rechten Nachfolger von A, ansonsten auch auf den linken Nachfolger - sofern es sich dabei ebenfalls um ein Paar handelt.

5. Wenn das Argument der Funktion `markExpression` kein Paar repräsentiert, wird mit Hilfe des Makros `isCAF` überprüft, ob es sich dabei um einen nullstelligen Superkombinator handelt:

```
#define isCAF(n) (whatIs(n) == NAME && name(n).arity == 0)
```

Sollte das der Fall sein, ruft die Funktion `markExpression` die Funktion `restrictCAF` auf und übergibt ihr als Argument den vorgefundenen Namen.

Die Funktion `restrictCAF` wird in der Datei `storage.c` wie folgt definiert:

```
static Void local restrictCAF(n)
Name n; { /* Funktionssymbol */
    if (name(n).assignment == NIL && !isDfun(n)) {
        if (name(n).defn == NIL)
            name(n).assignment = NON_ASSERTION;
        else {
            if (isClosure(name(n).defn))
                name(n).assignment = ASSERTION;
            else {
                name(n).assignment = NON_ASSERTION;
                markExpression(name(n).defn);
            }
        }
    }
    else
        doNothing();
}
```

Anmerkungen zu der Funktion `restrictCAF`:

1. Die Funktion `restrictCAF` dient dem Zweck, eine CAF, bei der es sich nicht um eine partielle Anwendung der Funktion `assertB`, `assertF` bzw. `primAssert` handelt, als Programmausdruck zu qualifizieren, wenn die Funktion `markExpression` deren Namen über einen Programmknoten erreicht hat.
2. Der Graph der als Programmausdruck zu qualifizierenden CAF muß zum Zeitpunkt des Aufrufs der Funktion `restrictCAF` noch nicht gebildet worden sein. In diesem Fall wird er für eine spätere Markierung vorgemerkt, indem lediglich die Komponente `assignment` in der dazugehörigen Struktur von Typ `struct strName` den Wert `NON_ASSERTION` erhält:

```
struct strName {
    ...
    Cell defn;
#ifdef DEBUGGING_WITH_ASSERTIONS
    Int assignment; /* NIL | ASSERTION | NON_ASSERTION */
    Int debugMarks; /* NOT_DETERMINED | DETERMINED */
#endif
    ...
};
```

Eine CAF wird spätestens dann endgültig markiert, wenn die Funktion `eval` auf sie zugreift, und zwar mit Hilfe der in der Datei `storage.c` definierten Funktion `markCAF`. Hier der entsprechende Ausschnitt aus dem Programmtext:

Kapitel 4 – Modifizierung des Haskell-Interpreters Hugs für die Überprüfung von Zusicherungen (Assertions)

```

Void eval(n)                                /* Graph reduction evaluator      */
Cell n; {
    ...

unw:switch (whatIs(n)) {                    /* unwind spine of application      */
    ...

*** 2 ***                                  /* CAF                               */
    else {                                  /* CAF                               */
        if (isNull(name(n).defn)) { /* build CAF                        */
            ...
        }
        n = name(n).defn;                /*already built*/
#ifdef DEBUGGING_WITH_ASSERTIONS
        if (debugging) {
            markCAF(saveName);
        }

        if (sp>base) {
            fun(top()) = n;
            if (debugging &&
                name(saveName).assignment ==\
                ASSERTION)
                updateDebugMarks(top());
        }
#else
        if (sp>base)
            fun(top()) = n;
#endif
    }
    ...
    goto unw;
}
...
}

```

Die Funktion `markCAF` überprüft mit Hilfe der in der Datei `storage.c` definierten Funktion `isClosure` zunächst, ob es sich bei der fraglichen CAF um eine partielle Anwendung der Funktion `assertB`, `assertF` bzw. `primAssert` auf zwei Argumente handelt. Sollte das der Fall sein, wird sie als `ASSERTION` qualifiziert und nicht markiert:

```

Void markCAF(n)
Name n; {                                  /* Funktionssymbol                */
    if (isClosure(name(n).defn)) {
        if (name(n).assignment != ASSERTION)
            name(n).assignment = ASSERTION;
        else
            doNothing();
    }
    else
        switch (name(n).assignment) {
            case NON_ASSERTION:
                if (name(n).debugMarks != DETERMINED) {
                    markExpression(name(n).defn);
                    name(n).debugMarks = DETERMINED;
                }
                else
                    doNothing();
            case NIL:
                if (interimCheck && !isDfun(n))
                    updateTempCAFMarks(n);
                else
                    doNothing();
        }
}

```

Ansonsten veranlaßt die Funktion `markCAF` das Setzen der entsprechenden Markierungen durch einen Aufruf der Funktion `markExpression`.

Da ein Graph während der Programmausführung laufend Veränderungen erfährt, ist es erforderlich, dessen Markierungen nach jeder Update-Anweisung zu aktualisieren. Hierfür werden sämtliche dieser Anweisungen um einen Aufruf der in der Datei `storage.c` definierten Funktion `updateDebugMarks` ergänzt:

```
Void updateDebugMarks(n)
Cell n; {                               /* Cell-Wert          */
    switch (debugMark(n)) {
        case NOT_MARKED    : doNothing();
                           break;
        case ASSERTION     : doNothing();
                           break;
        case NON_ASSERTION: if (isAssertion(n)) {
                           debugMark(n) = ASSERTION;
                           markExpression(snd(n));
                           }
                           else {
                           markExpression(fst(n));
                           markExpression(snd(n));
                           }
                           break;
        case CAF           : doNothing();
                           break;
    }
}
```

Der Funktion `updateDebugMarks` wird als Argument der `Cell`-Wert übergeben, der den durch die Update-Anweisung überschriebenen Applikationsknoten repräsentiert.

Zwei abschließende Bemerkungen zu der Unterscheidung zwischen Zusicherungsknoten und Programmknoten:

1. Die Beschränkungen, denen die Verwendung der Funktionen `assertB` und `assertF` unterliegt (vgl. Abschnitt 4.3), stellen sicher, daß unmarkiert bleibende Zusicherungsknoten nicht irrtümlich mit der Markierung `NON_ASSERTION` versehen werden. Wäre zum Beispiel eine Definition der Form

```
myAssert = assertF
```

zulässig, würde die Funktion `markExpression` den Ausdruck

```
myAssert msg (not . isEqSnd) (1, 0)
```

als Programmausdruck qualifizieren. Erst nach dessen Transformation in den Ausdruck

```
assertF msg (not . isEqSnd) (1, 0)
```

wäre erkennbar, daß lediglich das Paar `(1, 0)` einen Programmausdruck darstellt. Die fehlerhafte Markierung müßte also rückgängig gemacht werden. Auf entsprechende Vorkehrungen wird im Rahmen der Hugs-Anpassung verzichtet, weil die erwähnten Beschränkungen nicht die Aussagekraft einer zu überprüfenden Zusicherung mindern.

2. Gegeben seien folgende Definitionen:

```
pair :: (Int, Int)
pair = (1, 0)

foo :: Int -> Int
foo x = fst pair + x
```

Auszuwerten sei folgender Ausdruck:

```
1 + assertB message (snd pair /= 0) (foo 1)
```

Dieser Ausdruck ist für die Unterscheidung zwischen Zusicherungsknoten und Programmknoten mit Hilfe von Markierungen problematisch, weil die Zusicherung vor der Auswertung des Ausdrucks `foo 1` überprüft wird. Zu diesem Zeitpunkt ist für die Funktion `markExpression` noch nicht erkennbar gewesen, daß die CAF `pair` auch in der Definition von `foo` vorkommt und eigentlich einen Programmausdruck darstellt. Schlußfolgerung: Die Überprüfung einer Zusicherung müßte vom Grundsatz her zurückgestellt werden, bis eine Auswertung beendet ist, um einen unberechtigten Programmabbruch, wie er in diesem Beispiel droht, zu verhindern.

Insbesondere im Hinblick auf umfangreiche Berechnungen sieht die Hugs-Anpassung in diesem Zusammenhang jedoch einen Kompromiß vor: Zusicherungen werden zwischenzeitlich auch nach Beendigung einer Garbage Collection überprüft. Hierbei ist es

- zulässig, den Graphen einer CAF zu bilden, aber
- die Graphen aller CAF's, die nicht als Programmausdruck qualifiziert sind, werden temporär mit einer Markierung versehen, die signalisiert, daß eine Auswertung unzulässig ist - mit der Markierung CAF.

Für das Setzen der temporären Markierungen werden die in der Datei `storage.c` definierten Funktionen `markEveryUnrestrictedCAF` und `markEveryUnrestrictedDefinition` verwendet. Nach Überprüfung aller Zusicherungen werden diese Markierungen wieder entfernt.

4.4.2 Die Funktion `primAssert`

Die Funktion `primAssert` wird in der Datei `builtin.c` wie folgt definiert:

```
primFun(primAssert) {
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        /* Option: +A */
        if (projectionMode()) {
            /* Anmerkung 1 */
            saveDebugInfo(root);
            returnThdArg();
        }
        else {
            /* Anmerkung 2 */
            /* Anmerkung 2.1 */
            allocateEnv();

            if (setjmp(jumpBuffer->env) == 0) {
                /* Anmerkung 2.2 */
                /* Anmerkung 2.2.1 */
                eval(primArg(2));
                checkBool();
                if (whnfHead == nameTrue) {
                    /* Anmerkung 2.2.2 */
                    returnThdArg();
                    wrapUp(root);
                }
                else
                    /* Anmerkung 2.2.3 */
                    terminate(root);
            }
            else
                /* Anmerkung 2.3 */
                interrupt(root);
        }
    }
    else
        /* Option: -A */
        returnThdArg();
#else
    returnThdArg();
#endif
}
```

Aufgrund der Tatsache, daß die Grundstruktur der Funktion `primAssert` bereits im Rahmen des Abschnitts 4.3 dargelegt wurde, können sich die Erläuterungen zu ihrer Definition auf die als Kommentare angedeuteten Anmerkungen 1 - 2.3 beschränken. Sie beschreiben das Verhalten der Funktion `primAssert` nach Aktivierung der Option A.

Zu Anmerkung 1:

Zusicherungen werden nur überprüft, wenn die Variable `interimCheck` oder die Variable `finalCheck` den Wert `TRUE` hat:

```
Bool interimCheck = FALSE;
/* TRUE => Nach Beendigung der
/*          Garbage Collection
/*          werden die in der Check
/*          List verzeichneten
/*          Zusicherungen
/*          ueberprueft
Bool finalCheck = FALSE;
/* TRUE => Nach Beendigung der
/*          normalen Auswertung
/*          werden die in der Check
/*          List verzeichneten
/*          Zusicherungen
/*          ueberprueft
```

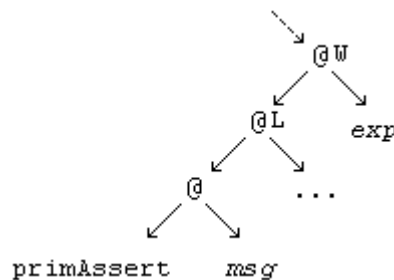
Zu jedem anderen Zeitpunkt, befindet sich die Funktion `primAssert` im "Projektionsmodus", erkennbar daran, daß die Bedingung

```
if (projectionMode())
```

nicht erfüllt ist:

```
#define projectionMode() (!interimCheck && !finalCheck)
```

Projektionsmodus bedeutet: Die Funktion `primAssert` verhält sich wie eine Projektion auf ihr drittes Argument, speichert zuvor aber mit Hilfe der Funktion `saveDebugInfo` (definiert in der Datei `storage.c`) Informationen, die für eine spätere Wiederherstellung der Zusicherung erforderlich sind, nämlich die Heap-Adresse der Applikationsknoten `W` und `L`:



Präzise formuliert: Die Funktion `saveDebugInfo` speichert die Heap-Adresse der Applikationsknoten `W` und `L` einer Struktur vom Typ `struct strAssertion`:

```
struct strAssertion {
    Cell root;
    Cell leftSubgraph;
    struct strAssertion *next;
};
```

Diese Struktur wird in eine verkettete Liste eingefügt, die `checkList`:

```
InfoRecord *checkList; /* Verkettete Liste, in der die */
/* Zusicherungen verzeichnet */
/* werden, deren Ueberpruefung */
/* zurueckgestellt bzw. abgebrochen */
/* wurde */
```

Zu Anmerkung 2:

Die Funktion `primAssert` befindet sich nicht im Projektionsmodus.

Zu Anmerkung 2.1:

Wenn die Überprüfung einer Zusicherung vorzeitig beendet werden muß, geschieht das durch einen direkten Rücksprung zur Funktion `primAssert` mit Hilfe einer `longjmp`-Anweisung. Die Funktion `allocateEnv` (definiert in der Datei `storage.c`) stellt in diesem Zusammenhang Speicher für den `setjmp`-Makro bereit.

Zu Anmerkung 2.2:

Direkter Aufruf von `set jmp`.

Zu Anmerkung 2.2.1:

Auswertung des zweiten Arguments der Funktion `primAssert` durch einen Aufruf der `eval`-Funktion.

Zu Anmerkung 2.2.2:

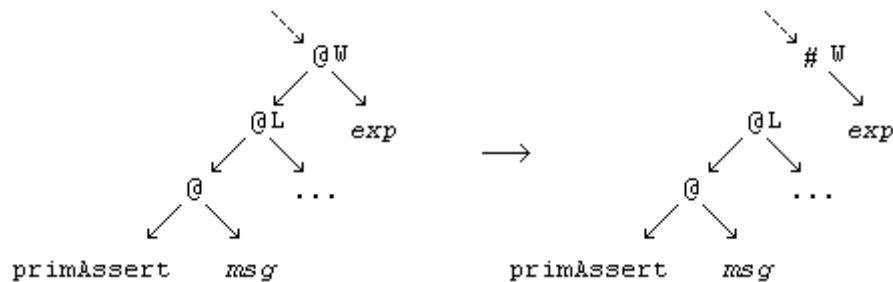
Wenn die Überprüfung der Zusicherung nicht abgebrochen werden mußte, und der Aufruf der `eval`-Funktion den Wert `True` ergeben hat, liefert die Funktion `primAssert` ihr drittes Argument als Ergebnis. Der anschließende Aufruf der Funktion `wrapUp` (definiert in der Datei `builtin.c`) dient unter anderem dem Zweck, die eventuell in der `checkList` gespeicherten Informationen zu löschen.

Zu Anmerkung 2.2.3:

Wenn die Überprüfung der Zusicherung nicht abgebrochen werden mußte, und der Aufruf der `eval`-Funktion den Wert `False` ergeben hat, bewirkt der Aufruf der Funktion `terminate` (definiert in der Datei `builtin.c`) einen Programmabbruch, dem die Wiedergabe der entsprechenden Fehlermeldung vorausgeht.

Zu Anmerkung 2.3:

Die Funktion `interrupt` (definiert in der Datei `builtin.c`) wird aufgerufen, wenn die Überprüfung der Zusicherung abgebrochen werden mußte. Sie dient insbesondere dem Zweck, die Wurzel des dazugehörigen Graphen mit einem Indirektionsknoten zu überschreiben, so daß die normale Programmausführung fortgesetzt werden kann:



Sie sorgt aber zum Beispiel auch dafür, daß über die Zusicherung Informationen im Sinne der Ausführungen zu Anmerkung 1 gespeichert werden, falls ihre Überprüfung erstmalig veranlaßt wurde - Voraussetzung: Die normale Auswertung wurde noch nicht beendet.

Wie wird entschieden, ob eine Auswertung im Rahmen der Überprüfung einer Zusicherung vorgenommen werden darf?

Zu diesem Zweck modifiziert die Hugs-Anpassung die Funktion `eval`. Und zwar dergestalt, daß das `if-else`-Konstrukt, auf dessen Grundlage sie den Code für einen mindestens einstelligen Superkombinator aufruft, durch folgendes Konstrukt ersetzt wird:

```
#if DEBUGGING_WITH_ASSERTIONS
    if (restrictEval())
        restrictedCodeExecution(root, n);
    else
        normalCodeExecution(root, n);
#else
    if (name(n).primDef) /* reduce */
        (*name(n).primDef)(root);
    else
        run(name(n).code, root);
#endif
```

Die Bedingung

```
    if (restrictEval())
```

ist erfüllt, wenn sich die Funktion `primAssert` nicht im Projektionsmodus befindet. Die Entscheidung über die Zulässigkeit einer Reduktion wird dann in Anlehnung an die Ausführungen des Abschnitts 3.3.3 anhand der Markierung des Wurzelknotens der betreffenden Applikation entschieden:

```
static Void local restrictedCodeExecution(root, n)
StackPtr root; /* Zeiger auf ein Stack-Element */
Name n; { /* Funktionssymbol */
    Cell rootElement;

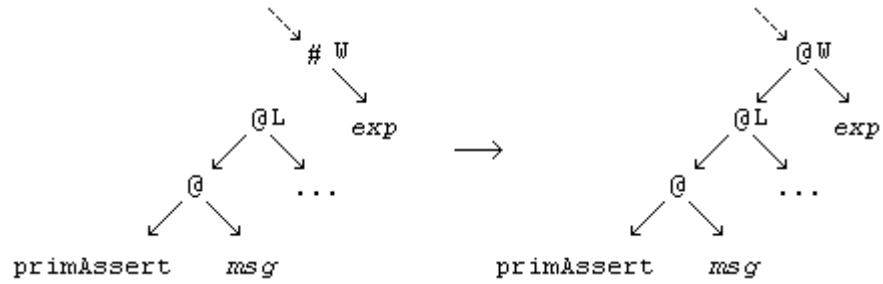
    rootElement = stack(root); /* Wurzel des Redex */
    if (canReduce(rootElement) || isDfun(n))
        normalCodeExecution(root, n);
    else
        longjmp(jumpBuffer->env, 1);
}

static Bool local canReduce(n)
Cell n; { /* Cell-Wert */
    switch (debugMark(n)) {
        case NOT_MARKED : return TRUE;
                          break;
        case ASSERTION : return TRUE;
                          break;
        case NON_ASSERTION: return FALSE;
                          break;
        case CAF : if (isAssertion(n))
                    return TRUE;
                  else
                    return FALSE;
                  break;
    }
}
```

Sowohl die Funktion `restrictedCodeExecution` als auch die Funktion `canReduce` wird in der Datei `machine.c` definiert.

4.4.3 Wiederherstellung einer Zusicherung

Für das Wiederherstellen einer Zusicherung sieht die Hugs-Anpassung das im Rahmen des Abschnitts 3.3.3 vorgestellte Verfahren der "Wiederherstellung durch die Rückgängigmachung der Projektion" vor:



Eine solche Wiederherstellung wird nur nach Beendigung einer Garbage Collection bzw. der gesamten Auswertung durchgeführt, und zwar nacheinander für alle Zusicherungen, die in der `checkList` verzeichnet sind. Maßgebend ist hierfür die in der Datei `machine.c` definierte Funktion `checkAssertion`:

```
static Void local checkAssertion(StackPtr root)
StackPtr root; {
    if (interimCheck) {
        if (!emptyCheckList()) {
            reducedExpr = stack(root);
            if (!gcRequired)
                gcRequired = TRUE;
            if (!haveTempCAFMarks)
                markEveryUnrestrictedCAF();
            restoreAssertion(root);
        }
        else {
            if (!emptyClipboard())
                restoreCheckList();
            if (gcRequired) {
                garbageCollect();
                gcRequired = FALSE;
            }
            if (haveTempCAFMarks)
                unmarkEveryUnrestrictedCAF();
            interimCheck = FALSE;
        }
    }
    else {
        if (!emptyCheckList()) {
            if (debugMessageRequired) {
                printDebugMessage();
                debugMessageRequired = FALSE;
            }
            if (!resultMessageRequired)
                resultMessageRequired = TRUE;
            restoreAssertion(root);
        }
        else {
            finalCheck = FALSE;
            if (traceError)
                traceError = FALSE;
            if (debugMessageRequired)
                debugMessageRequired = FALSE;
            if (resultMessageRequired) {
                printResultMessage();
                resultMessageRequired = FALSE;
            }
        }
    }
}
```

Der Aufruf der Funktion `checkAssertion` erfolgt grundsätzlich, bevor der Zeiger auf den Wurzelknoten W^* einer reduzierten Applikation durch die Anweisungsfolge

```
sp = root;                /* continue... */
n  = pop();
```

in der Funktion `eval` vom Stack genommen und erneut dem Unwind-Mechanismus übergeben wird. Hier der entsprechende Ausschnitt aus dem Programmtext:

```
Void eval(n)                /* Graph reduction evaluator */
Cell n; {
    ...

    unw:switch (whatIs(n)) {                /* unwind spine of application */
        ...

        case NAME : allowBreak();
            ...
            if (!isCfun(n) && (ar=name(n).arity)<=(sp-base)) {
*** 1 ***                if (ar>0) {                /* fn with args*/
                            StackPtr root;

                                push(NIL);                /* rearrange */
                                ...
                                if (name(n).primDef)        /* reduce */
                                    (*name(n).primDef)(root);
                                else
                                    run(name(n).code,root);
                                ...

                                sp = root;                /* continue... */

#ifdef DEBUGGING_WITH_ASSERTIONS
                                    if (debugging) {
                                        if (saveName == nameIOCase) {
                                            finalCheck = TRUE;
                                            debugMessageRequired = TRUE;
                                        }

                                        if (!restrictEval() &&
                                            !projectionMode())
                                            checkAssertion(root);
                                    }
                                }

                                n = pop();
*** 2 ***                } else {                /* CAF */
                            ...
                            }
                            ...
                            goto unw;
                        }
    }
}
```

Die Funktion `checkAssertion` bewirkt bei ihrem Aufruf, daß der Zeiger auf den Knoten W^* vom Stack genommen und durch den Zeiger auf den Wurzelknoten der wiederhergestellten Anwendung der Funktion `primAssert` ersetzt wird. Folge: Der Unwind-Mechanismus stößt automatisch die Überprüfung der Zusicherung an.

Dieser Vorgang wiederholt sich analog für jede wiederherzustellende Zusicherung, das heißt, die in der `checkList` verzeichneten Zusicherungen werden streng sequentiell abgearbeitet.

4.4.4 Vorbereitung einer Auswertung

Unmittelbar vor einer neuen Auswertung wird die in der Datei `hugs.c` definierte Funktion `prepareEvaluation` ausgeführt:

```
static Void local prepareEvaluation(e)
Cell e; {
    /* Wurzel des zu reduzierenden      */
    /* Ausdrucks                        */

    if (firstRun) {
        if (!debugging)
            doNothing();
        else {
            saveCAFHandling();
            mainRoot = e;
            markInputExpr(e);
            cleanUpRequired = TRUE;
        }
    }
    else {
        if (!debugging) {
            if (!cleanUpRequired)
                doNothing();
            else {
                deleteAllDebugMarks();
                deleteEveryCAF();
                cleanUpRequired = FALSE;
            }
        }
        else {
            if (!cleanUpRequired) {
                deleteEveryCAF();
                cleanUpRequired = TRUE;
            }
            else {
                if (deleteCAF)
                    deleteEveryCAF();
                else {
                    if (newCAFHandling())
                        deleteEveryCAF();
                    else
                        unassignEveryCAF();
                }
            }
            saveCAFHandling();
            mainRoot = e;
            markInputExpr(e);
        }
    }
}
```

Diese Funktion dient unter anderem folgenden wichtigen Aufgaben:

1. Wenn die Option `A` aktiviert ist, veranlaßt sie die Markierung des Anfangsgraphen durch einen Aufruf des in der Datei `storage.h` definierten Makros `markInputExpr`:

```
#define markInputExpr(n) markExpression(n)
```

2. Wenn die Option `A` aktiviert und die Option `d` deaktiviert ist, veranlaßt sie das Löschen der Markierungen in den Graphen der CAF's durch einen Aufruf der in der Datei `storage.c` definierten Funktion `unassignEveryCAF`.
3. Wenn sowohl die Option `A` als auch die Option `d` aktiviert ist, veranlaßt sie das Löschen der CAF-Definitionen durch einen Aufruf der in der Datei `storage.c` definierten Funktion `deleteEveryCAF`.

4.4.5 Modifikation der Fehlerbehandlung

Wenn es nicht möglich ist, die Reduktion eines Ausdrucks ordnungsgemäß zu beenden, ruft der Auswertungsmechanismus des Haskell-Interpreters Hugs eine eingebaute Fehlerbehandlungsroutine auf, und zwar die in der Datei `machine.c` definierte Funktion `evalFails`. Was dieser Aufruf zum Beispiel bei einer Division durch die Zahl 0 bewirkt, hat die Auswertung des Ausdrucks

```
1 / fromInt (snd pair)
```

im Abschnitt 4.3 gezeigt: Die Programmausführung wird mit der Ausgabe der Fehlermeldung

```
Program error: {primDivDouble 1.0 0.0}
```

abgebrochen.

Die Hugs-Anpassung modifiziert die in der Datei `hugs.c` definierte Funktion `interpreter` so, daß nach einem Programmabbruch die in der `checkList` verzeichneten Zusicherungen einer Überprüfung unterzogen werden:

```
static Void local interpreter(argc,argv)/* main interpreter loop          */
Int      argc;
String argv[]; {
    Int errorNumber = setjmp(catch_error);

    ...

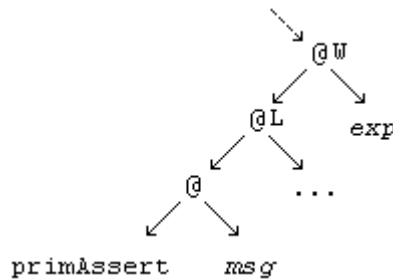
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging && errorNumber) {
        if (!restrictEval()) {
            if (!emptyCheckList()) {
                finalCheck = TRUE;
                traceError = TRUE;
                printDebugMessage();
                resultMessageRequired = TRUE;
                clearStack();
                restoreAssertion(++sp);
                eval(pop());
            }
            else
                doNothing();
        }
        else
            doNothing();

        systemCheckRequired = TRUE;
    }
    else
        doNothing();
#endif
    ...
    cmd = readCommand(cmds, (Char)':' , (Char) '!');
    ...
    switch (cmd) {
        case EDIT : editor();
                 break;
        ...
    }
    ...
}
```

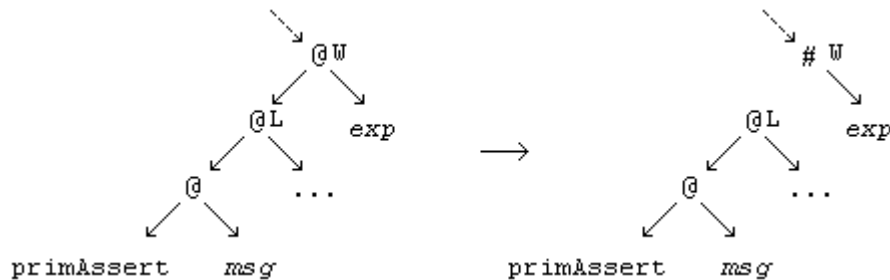
Diese Modifikation stellt zum Beispiel sicher, daß Anwendungen der Funktion `assertB` bzw. `assertF` noch einmal wiederhergestellt werden, die bis zum Aufruf der Funktion `evalFails` lediglich als Projektion behandelt wurden.

4.4.6 Modifikation der Garbage Collection

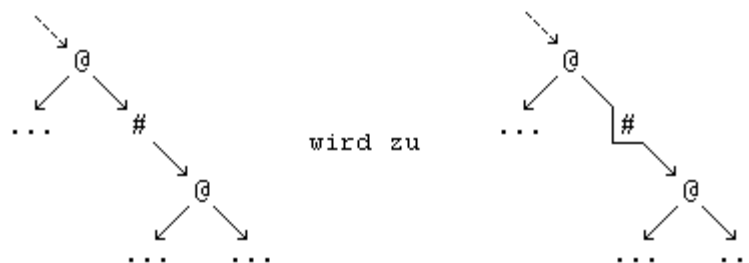
Schematisch sei eine Anwendung der Funktion `primAssert` wie folgt dargestellt:



Wenn es erforderlich ist, die Reduktion dieser Anwendung vorzeitig zu beenden, sorgt die Funktion `interrupt` dafür, daß der Applikationsknoten `W` mit einem Indirektionsknoten überschrieben wird, der als Zeiger auf den Ausdruck `exp` fungiert:



Da der Haskell-Interpreter Hugs über eine automatische Garbage Collection verfügt, muß nach diesem Vorgang sichergestellt sein, daß der Speicherplatz, den der am Knoten `L` beginnende Teilgraph in Anspruch nimmt, nicht für eine neue Belegung freigegeben wird. Gleiches gilt für den Indirektionsknoten `W`, denn im Rahmen der Garbage Collection werden Zeiger auf Indirektionsknoten grundsätzlich durch Zeiger auf deren Nachfolger ersetzt:



Um diesen Anforderungen gerecht zu werden, sieht die Hugs-Anpassung vor, daß im Rahmen der Markierungsphase der Garbage Collection (vgl. Abschnitt 4.2.2) die in der Datei `storage.c` definierten Funktionen `restorePendingAssertions` und `markPendingAssertions` aufgerufen werden:

Kapitel 4 – Modifizierung des Haskell-Interpreters Hugs für die Überprüfung von Zusicherungen (Assertions)

```

static Void local markPendingAssertions() {
    InfoRecord *temp;

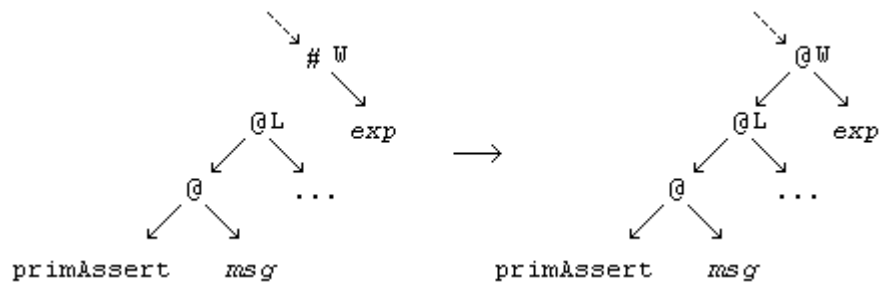
    if (!emptyCheckList()) {
        if (restrictEval())
            temp = checkList->next;
        else
            temp = checkList;
        while (temp->root != NIL) {
            mark(temp->root);
            temp = temp->next;
        }
    }

    if (!emptyClipboard()) {
        temp = clipboard;
        while (temp->root != NIL) {
            mark(temp->root);
            temp = temp->next;
        }
    }
}

```

Diese Funktionsaufrufe bewirken folgendes:

1. Alle vorzeitig beendeten Anwendungen der Funktion `primAssert` werden wiederhergestellt:



Die Cell-Werte, die die Wurzelknoten der entsprechenden Graphen G_i repräsentieren, seien mit W_i bezeichnet.

2. Der in der Datei `storage.h` definierte Makro `mark` wird nacheinander auf sämtliche Cell-Werte W_i angewendet. Das veranlaßt die Speicherwaltung, alle Heap-Elemente, die die Knoten der Graphen G_i repräsentieren, so zu markieren, daß sie nicht der `freeList` hinzugefügt werden (vgl. Abschnitt 4.2.3).

5 Zusammenfassung

Die Verifikation bedient sich mathematischer Mittel, um die Konsistenz zwischen einem Programm und seiner Spezifikation formal exakt zu beweisen. Dennoch sollte in der Software-Entwicklung auf Testverfahren als Qualitätssicherungsmaßnahme nicht verzichtet werden. Der Abschnitt 3.1 hat hierfür Gründe aufgezeigt:

1. Eine Beweisführung kann fehlerhaft sein. Es ist daher durchaus von Vorteil, sie durch systematische Tests zu ergänzen, denn auf diese Weise verringert sich die Fehlerwahrscheinlichkeit.
2. Auch ein korrektes Programm kann ein fehlerhaftes Verhalten zeigen. Als mögliche Ursachen ist in diesem Zusammenhang an Fehler in Übersetzern, Betriebssystemen und Hardwarekomponenten zu denken. Ein Programmtest deckt dann zwar nicht nur reine Programmierfehler auf, er vermittelt aber einen Eindruck davon, wie sich die entwickelte Software unter realen Umgebungsbedingungen verhält.

Ein Werkzeug, das während der Programmausführung das Eintreffen bestimmter, vorher definierter Bedingungen - sogenannter Zusicherungen - überprüft, bietet in diesem Zusammenhang einen wichtigen Vorteil gegenüber anderen Verfahren, deren Ziel es ist, das Testen zu systematisieren, zu automatisieren und/oder zu ergänzen: Es führt unter Umständen zu der Entdeckung von Fehlern, die sich in den Ausgabedaten nicht dokumentieren. Und zwar deshalb, weil die Spezifikation in den Programmtext eingebunden werden kann, wenn man die von dem Programm zu bewältigende Aufgabe mit Hilfe von Prädikaten zum Ausdruck bringt.

Wie der Abschnitt 3.2 gezeigt hat, kann man Prädikate für die Beschreibung einer Aufgabenstellung in der funktionalen Programmierung zum Teil sehr viel einfacher finden als in der imperativen Programmierung. Der Grund: Variablen werden dort in einer Weise verwendet, wie sie aus der Mathematik vertraut ist - Stichwort "Referentielle Transparenz".

Die nicht-strikte funktionale Programmiersprache Haskell, die den Untersuchungsgegenstand dieser Arbeit darstellt, erfüllt die wichtigsten Voraussetzungen für die Steuerung des Programmablaufs mit Hilfe von Zusicherungen:

- Es stehen ausreichende Mittel für die Formulierung boolescher Ausdrücke zur Verfügung; im einzelnen:
 - Ein vordefinierter Datentyp `Bool`, dessen Konstruktoren `True` und `False` die gleichlautenden booleschen Werte repräsentieren:

```
data Bool
  = False | True
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

- Verschiedene Vergleichsoperatoren: `<`, `<=`, `==`, `/=`, `>=`, `>`

In normaler Funktionsschreibweise ausgedrückt, handelt es sich hierbei um die Operatoren `<`, `≤`, `=`, `≠`, `≥`, `>`.

- Verschiedene logische Operatoren: `&&`, `|`, `not`

Diese Operatoren repräsentieren die logischen Funktionen \wedge , \vee , \neg .

- Mit der Funktion `error` steht eine Fehleroutine für den gegebenenfalls erforderlichen Programmabbruch zur Verfügung.

Hierauf aufbauend wurden für die Überprüfung von Zusicherungen zunächst folgende Funktionen definiert:

```
assert :: String -> Bool -> a -> a
assert msg cond exp
  = case cond of
      True  -> exp
      False -> error msg

assertF :: String -> (a -> Bool) -> a -> a
assertF msg func exp = assert msg (func exp) exp
```

Die weitere Untersuchung hat gezeigt, daß diese Funktionen jedoch aufgrund der Nicht-Striktheit der Programmiersprache Haskell unter Umständen die Semantik eines Programms verändern. Das heißt, die spracheigenen Mittel sind für die Überprüfung von Zusicherungen nur bedingt geeignet. Sie müssen durch Änderungen in der Implementierung der Programmiersprache Haskell ergänzt werden. Der Abschnitt 3.3.3 hat die hierfür maßgebenden Prinzipien aufgezeigt:

1. Die Überprüfung einer Zusicherung darf in einem Programmausdruck grundsätzlich keine Auswertung erzwingen, sondern beruht auf der Inaugenscheinnahme bereits ausgewerteter Ausdrücke. Das heißt, unter Umständen ist es erforderlich, sie ohne Resultat abzubrechen.
2. Wenn man der Forderung nach dem Erhalt der Semantik vollkommen gerecht werden will, müssen der Überprüfung einer Zusicherung Beschränkungen auferlegt werden, die sicherstellen, daß die Semantik eines Programms nicht alleine schon aufgrund ihrer Inangsetzung verändert wird. Denkbar ist zum Beispiel eine zeitliche Limitierung (Timeout) der Überprüfung oder eine zahlenmäßige Beschränkung der Reduktionen, die sie anstoßen darf.
3. Wenn die Überprüfung einer Zusicherung abgebrochen wird, muß die Fortsetzung der normalen Programmausführung sichergestellt sein. Das bedeutet vom Ergebnis her: Die Zusicherung gilt erst einmal als erfüllt.
4. Wenn die normale Programmausführung eine Reduktion bewirkt hat, die zuvor im Rahmen der Überprüfung einer Zusicherung als unzulässig erachtet wurde, muß diese Zusicherung einer erneuten Überprüfung unterzogen werden.

Die Beachtung dieser Prinzipien stellt sicher, daß die Semantik eines Haskell-Programms durch die Überprüfung einer Zusicherung nicht verändert wird. Sie hat allerdings zur Folge, daß ein Fehler - im Gegensatz zu einer strikten Programmiersprache - unentdeckt bleibt, wenn die beiden folgenden Voraussetzungen erfüllt sind:

3. Die Überprüfung der Zusicherung, mit deren Hilfe sich der Fehler eigentlich aufdecken ließe, muß ergebnislos abgebrochen werden, weil die Reduktion eines Ausdrucks A als unzulässig erachtet wird.
4. Der Ausdruck A erfährt im Rahmen des normalen Programmablaufs keine Reduktion.

Der Abschnitt 4 hat aufgezeigt, wie die im Rahmen der vorliegenden Arbeit gewonnenen Erkenntnisse in der Implementierung des Haskell-Interpreters Hugs, Version: Mai 1999, umgesetzt wurden.

Anhang A

Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

Der Anhang A dokumentiert sämtliche Änderungen an der Implementierung des Haskell-Interpreters Hugs, Version: Mai 1999, die im Rahmen der vorliegenden Diplomarbeit vorgenommen wurden, und zwar gegliedert nach den Quellcode-Dateien, die sie betreffen.

A.1 Änderungen in der Datei `~/Hugs98/lib/Prelude.hs`

1. Änderung:

```
seq, ($!),  
-- Debugging with Assertions:  
assertB, assertF
```

ersetzt in der Originaldatei Zeile 103:

```
seq, ($!)
```

2. Änderung:

```
-- Debugging with Assertions -----  
primitive primAssert :: String -> Bool -> a -> a  
  
assertB :: String -> Bool -> a -> a  
assertB = primAssert  
  
assertF :: String -> (a -> Bool) -> a -> a  
assertF msg func exp = assertB msg (func exp) exp  
  
-- End of Hugs standard prelude -----
```

ersetzt in der Originaldatei Zeile 1668:

```
-- End of Hugs standard prelude -----
```

A.2 Änderungen in der Datei `~/Hugs98/src/options.h.in`

1. Änderung:

```
#define CHECK_TAGS 0

/* Wenn der Name DEBUGGING_WITH_ASSERTIONS mit dem Wert 1 definiert wird,
 * stehen dem Anwender zwei zusätzliche Optionen A und d zur Verfügung.
 * Sie versetzen ihn in die Lage, einem Programm fuer die Ueberpruefung von
 * Zusicherungen mit Hilfe der beiden Funktionen assertB und assertF
 * Testpunkte hinzuzufuegen.
 *
 * Gemaess der Definition in der Datei Prelude.hs hat die Funktion assertB
 * drei Argumente:
 * 1. msg -> String
 * 2. cond -> Praedikat
 * 3. exp -> Ausdruck vom Typ a
 *
 * Unter der Option +A ueberprueft die Funktion assertB den Wert ihres
 * zweiten Arguments. Ist er False, wird die Ausfuehrung des Programms
 * abgebrochen und msg als Fehlermeldung ausgegeben. Ansonsten ist das
 * Resultat exp. Analog verhaelt sich die Funktion assertF.
 *
 * Das erweckt zunaechst den Anschein, als verhielten sich beide Funktionen
 * gemaess den nachstehenden Haskell-Definitionen:
 *
 * assertB :: String -> Bool -> a -> a
 * assertB msg cond exp
 *   = case cond of
 *     True  -> exp
 *     False -> error msg
 *
 * assertF :: String -> (a -> Bool) -> a -> a
 * assertF msg func exp = assertB msg (func exp) exp
 *
 * Es gibt jedoch zwei wesentliche Unterschiede:
 * 1. Aehnlich wie bei dem assert-Makro in der Programmiersprache C ist es
 *    durch die Wahl der Option -A moeglich, die Testpunkte in einem
 *    Programm ignorieren zu lassen. Die Funktionen assertB und assertF
 *    verhalten sich dann wie eine Projektion auf das dritte Argument.
 * 2. Eine Auswertung, die im Rahmen des normalen Programmablaufs nicht
 *    erforderlich ist, wird auch durch die Ueberpruefung einer Zusicherung
 *    nicht erzwungen. Die Funktionen assertB und assertF verhalten sich
 *    also nicht-strikt.
 *
 * Die Option d ist fuer die Handhabung der CAF's von Bedeutung. Hugs98
 * ist so konzipiert, dass der Graph einer CAF im Heap erhalten bleibt,
 * bis das definierende Skript nach einer Aenderung erneut geladen wird
 * oder durch einen load-Befehl seine Gueltigkeit verliert. Dies kann der
 * Anwender durch die Option d beeinflussen: Unter der Option -d zeigt
 * das System weiterhin das soeben beschriebene Verhalten. Unter der Option
 * +d werden dagegen vor einen neuen Auswertung saemtliche CAF's geloescht.
 * Voraussetzung dafuer ist allerdings, dass gleichzeitig die Option +A
 * gewaehlt ist.
 */
#define DEBUGGING_WITH_ASSERTIONS 1
```

ersetzt in der Originaldatei Zeile 160:

```
#define CHECK_TAGS 0
```

A.3 Änderungen in der Datei `~/Hugs98/src/connect.h`

1. Änderung:

```
/* -----  
 * Debugging with Assertions  
 * ----- */  
  
#if DEBUGGING_WITH_ASSERTIONS  
extern Bool debugging;  
extern Bool deleteCAF;  
extern Bool gcRequired;  
extern Bool firstRun;  
  
extern Void restoreAssertion    Args((StackPtr));  
extern Void printDebugMessage  Args((Void));  
#endif  
  
/*----- */
```

ersetzt in der Originaldatei Zeile 364:

```
/*----- */
```

A.4 Änderungen in der Datei `~/Hugs98/src/hugs.c`

1. Änderung:

```
static String local strCopy          Args((String));
#if DEBUGGING_WITH_ASSERTIONS
static Void   local saveNames        Args((Void));
static Void   local prepareEvaluation Args((Cell));
#endif
```

ersetzt in der Originaldatei Zeile 81:

```
static String local strCopy          Args((String));
```

2. Änderung:

```
        String scriptFile;          /* Name of current script (if any) */

#if DEBUGGING_WITH_ASSERTIONS
        String preludeFile;         /* Name der Datei Prelude.hs      */
        Bool   firstRun   = TRUE;   /* TRUE => Es wurde noch keine    */
                                           /* Auswertung vorgenommen        */
#endif
```

ersetzt in der Originaldatei Zeile 111:

```
        String scriptFile;          /* Name of current script (if any) */
```

3. Änderung:

```
    }

#if DEBUGGING_WITH_ASSERTIONS
    preludeFile = scriptName[0];
#endif
```

ersetzt in der Originaldatei Zeile 240:

```
    }
```

4. Änderung:

```
    {'i', "Chase imports while loading modules", &chaseImports},
#if DEBUGGING_WITH_ASSERTIONS
    {'d', "Delete constant applicative forms before evaluation
        (Prerequisite: Option +A)", &deleteCAF},
#endif
```

ersetzt in der Originaldatei Zeile 666:

```
    {'i', "Chase imports while loading modules", &chaseImports},
```

5. Änderung:

```
#endif
#if DEBUGGING_WITH_ASSERTIONS
    {'A', "Debugging with assertions: print an error message and terminate
        the program if the expression tested is false", &debugging},
#endif
```

ersetzt in der Originaldatei Zeile 672

```
#endif
```

6. Änderung:

```
    compileDefns();
#if DEBUGGING_WITH_ASSERTIONS
    if (scriptFile == preludeFile)
        saveNames();
#endif
```

ersetzt in der Originaldatei Zeile 768:

```
    compileDefns();
```

7. Änderung:

```
    run(inputCode,sp); /* Build graph for redex */
#if DEBUGGING_WITH_ASSERTIONS
    prepareEvaluation(top());
#endif
```

ersetzt in der Originaldatei Zeile 1074:

```
    run(inputCode,sp); /* Build graph for redex */
```

8. Änderung:

```
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging && errorNumber) {
        if (!restrictEval()) {
            if (!emptyCheckList()) {
                finalCheck = TRUE;
                traceError = TRUE;
                printDebugMessage();
                resultMessageRequired = TRUE;
                clearStack();
                restoreAssertion(++sp);
                eval(pop());
            }
            else
                doNothing();
        }
        else
            doNothing();

        systemCheckRequired = TRUE;
    }
    else
        doNothing();
#endif
for (;;) {
```

ersetzt in der Originaldatei Zeile 1486:

```
for (;;) {
```

9. Änderung:

```
}

/* -----
 * Debugging with Assertions
 * ----- */

#if DEBUGGING_WITH_ASSERTIONS
/* saveNames:
 * -----
 * Die Funktion saveNames speichert in den Variablen nameAssert, nameAssertB
 * und nameAssertF die Namen der Funktionen, mit deren Hilfe Zusicherungen
 * ueberprueft werden koennen.
 */
static Void local saveNames() {
    nameAssert = findName(findText("primAssert"));
    nameAssertB = findName(findText("assertB"));
    nameAssertF = findName(findText("assertF"));
}

/* prepareEvaluation:
 * -----
 * Die Funktion prepareEvaluation wird unmittelbar vor einer neuen Auswertung
 * ausgefuehrt, und zwar nachdem der dazugehoerende Graph konstruiert wurde.
 * Sie dient dem Zweck, diesen Graphen zu markieren, gegebenenfalls die
 * CAF-Definitionen zu loeschen und die aktuelle Einstellung fuer die Option
 * d zu speichern.
 *
 * Anmerkungen:
 * 1) Erste Auswertung nach dem Aufruf von Hugs98.
 * 2) Zweite Auswertung ff. nach dem Aufruf von Hugs98.
 * 2.1) Aktuelle Auswertung findet unter der Option -A statt.
 * Vorhergehende Auswertung fand unter der Option -A statt.
 * 2.2) Aktuelle Auswertung findet unter der Option -A statt.
 * Vorhergehende Auswertung fand unter der Option +A statt.
 * 2.3) Aktuelle Auswertung findet unter der Option +A statt.
 * Vorhergehende Auswertung fand unter der Option -A statt.
 * 2.4) Aktuelle Auswertung findet unter der Option +A statt.
```


Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

*      Vorhergehende Auswertung fand unter der Option +A statt.
* 2.4.1) Die CAF-Definitionen werden geloescht (Option: +A / +d).
* 2.4.2) Aktuelle Auswertung findet unter der Option -d statt.
*      Vorhergehende Auswertung fand unter der Option +d statt.
* 2.4.3) Aktuelle Auswertung findet unter der Option -d statt.
*      Vorhergehende Auswertung fand unter der Option -d statt.
*/
static Void local prepareEvaluation(e)
Cell e; {
    /* Wurzel des zu reduzierenden      */
    /* Ausdrucks                          */
    if (firstRun) {                       /* Anmerkung 1      */
        if (!debugging)
            doNothing();
        else {
            saveCAFHandling();
            mainRoot = e;
            markInputExpr(e);
            cleanUpRequired = TRUE;
        }
    }
    else { /* Anmerkung 2 */
        if (!debugging) {
            if (!cleanUpRequired) /* Anmerkung 2.1 */
                doNothing();
            else { /* Anmerkung 2.2 */
                deleteAllDebugMarks();
                deleteEveryCAF();
                cleanUpRequired = FALSE;
            }
        }
        else {
            if (!cleanUpRequired) { /* Anmerkung 2.3 */
                deleteEveryCAF();
                cleanUpRequired = TRUE;
            }
            else { /* Anmerkung 2.4 */
                if (deleteCAF) /* Anmerkung 2.4.1 */
                    deleteEveryCAF();
                else {
                    if (newCAFHandling()) /* Anmerkung 2.4.2 */
                        deleteEveryCAF();
                    else /* Anmerkung 2.4.3 */
                        unassignEveryCAF();
                }
            }
            saveCAFHandling();
            mainRoot = e;
            markInputExpr(e);
        }
    }
}
#endif

```

ersetzt in der Originaldatei Zeile 1738:

```

}

```

A.5 Änderungen in der Datei ~/Hugs98/src/builtin.c

1. Änderung:

```
#if DEBUGGING_WITH_ASSERTIONS
Void update_DWA Args((StackPtr, Cell, Cell));

/* update_DWA:
 * -----
 * Sofern der Name DEBUGGING_WITH_ASSERTIONS mit dem Wert 1 definiert
 * wird, ersetzt die Funktion update_DWA die Update-Anweisung fuer die
 * eingebauten Funktionen. Sie sorgt dafuer, dass unter der Option +A die
 * Debug-Markierungen des reduzierten Ausdrucks aktualisiert werden.
 */
Void update_DWA(root, l, r)
StackPtr root; /* Zeiger auf ein Stack-Element */
Cell l; /* Cell-Wert */
Cell r; { /* Cell-Wert */
    Cell rootElement; /* Inhalt des Stack-Elements */

    rootElement = stack(root);
    snd(rootElement) = r;
    fst(rootElement) = l;
    if (debugging)
        updateDebugMarks(rootElement);
}

#define update(l,r) update_DWA(root,l,r)
#else
#define update(l,r) ((snd(stack(root))=r),(fst(stack(root))=l))
#endif
```

ersetzt in der Originaldatei Zeile 96:

```
#define update(l,r) ((snd(stack(root))=r),(fst(stack(root))=l))
```

2. Änderung:

```
PROTO_PRIM(primPtrToInt);
PROTO_PRIM(primAssert);
```

ersetzt in der Originaldatei Zeile 594:

```
PROTO_PRIM(primPtrToInt);
```

3. Änderung:

```
static Cell local followInd Args(( Cell ));

#if DEBUGGING_WITH_ASSERTIONS
static Void local wrapUp Args((StackPtr));
static Void local terminate Args((StackPtr));
static Void local interrupt Args((StackPtr));
static Void local resetRootPointer Args((StackPtr));
#endif

/* -----
 * Debugging with Assertions
 * ----- */

/* returnThdArg:
 * -----
 * Der Makro returnThdArg definiert eine Projektion auf das letzte
 * Argument einer dreistelligen Funktion.
 */
#define returnThdArg() updateRoot(primArg(1))
```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```
#if DEBUGGING_WITH_ASSERTIONS
/* wrapUp:
 * -----
 * Die Funktion wrapUp wird aufgerufen, wenn die Ueberpruefung einer
 * Zusicherung den Wert True ergeben hat. Sie dient der Systembereinigung.
 *
 * Anmerkungen:
 * 1) Die Ueberpruefung der Zusicherung wurde urspruenglich zurueckgestellt.
 * Die hierbei ueber sie gespeicherten Informationen koennen geloescht
 * werden.
 * 2) Der zu der ueberprueften Zusicherung gehoernde Ausdruck wurde
 * erstmalig reduziert.
 */
static Void local wrapUp(root)
StackPtr root; { /* Zeiger auf ein Stack-Element */
    freeEnv();
    if (stack(root) == checkList->root) { /* Anmerkung 1 */
        deleteDebugInfo();
        if (!traceError)
            resetRootPointer(root);
    }
    else /* Anmerkung 2 */
        doNothing();
}

/* terminate:
 * -----
 * Die Funktion terminate wird aufgerufen, wenn die Ueberpruefung einer
 * Zusicherung den Wert False ergeben hat. Sie dient insbesondere dem Zweck,
 * saemtliche fuer die Ueberwachung eines Programms gespeicherten Informationen
 * zu loeschen und dessen Ausfuehrung zusammen mit der Wiedergabe der
 * entsprechenden Fehlermeldung (1. Argument der Funktion primAssert)
 * augenblicklich zu beenden.
 */
static Void local terminate(root)
StackPtr root; { /* Zeiger auf ein Stack-Element */
    freeAllEnv();
    deleteAllDebugInfos();
    if (interimCheck) {
        interimCheck = FALSE;
        if (gcRequired)
            gcRequired = FALSE;
        if (haveTempCAFMarks)
            unmarkEveryUnrestrictedCAF();
    }
    else {
        finalCheck = FALSE;
        if (traceError)
            traceError = FALSE;
        if (numPending != 0)
            numPending = 0;
        if (debugMessageRequired)
            debugMessageRequired = FALSE;
        if (resultMessageRequired)
            resultMessageRequired = FALSE;
    }

    Printf("\n");
    Printf("\n");
    Printf("*** Program error: Assertion failed\n");
    Printf("*** ");
    push(primArg(3));
    outputString(stdout);
    ignoreError = TRUE;
    errAbort();
}

/* interrupt:
 * -----
 * Die Funktion interrupt wird aufgerufen, wenn die Ueberpruefung einer
 * Zusicherung abgebrochen wurde. Sie dient unter anderem dem Zweck, die
 * Wurzel des dazugehoerenden Graphen mit einem Indirektionsknoten zu
 * ueberschreiben, so dass die normale Programmausfuehrung fortgesetzt
```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```
* werden kann. Falls erforderlich, veranlasst sie ausserdem das Speichern
* bzw. Loeschen der Informationen, die fuer eine Wiederherstellung
* der ueberprueften Zusicherung benoetigt werden.
*/
static Void local interrupt(root)
StackPtr root; { /* Zeiger auf ein Stack-Element */
    freeEnv();
    if (interimCheck) {
        if (stack(root) == checkList->root) {
            returnThdArg();
            cutDebugInfo();
            resetRootPointer(root);
        }
        else {
            saveDebugInfo(root);
            returnThdArg();
        }
    }
    else {
        if (stack(root) == checkList->root) {
            returnThdArg();
            deleteDebugInfo();
            if (!traceError)
                resetRootPointer(root);
        }
        else
            returnThdArg();
        ++numPending;
    }
}

/* resetRootPointer:
 * -----
 * Die Funktion resetRootPointer ersetzt den Inhalt des Stack-Elements,
 * das ihr Argument bezeichnet, durch den Wert der Variablen reducedExpr
 * bzw. mainRoot.
 */
static Void local resetRootPointer(root)
StackPtr root; { /* Zeiger auf ein Stack-Element */
    if (interimCheck)
        stack(root) = reducedExpr;
    else
        stack(root) = mainRoot;
}
#endif
```

ersetzt in der Originaldatei Zeile 597:

```
static Cell local followInd Args(( Cell ));
```

4. Änderung:

```
 {"unsafePtrToInt", 1, primPtrToInt}, /* breaks the semantics */
 {"primAssert", 3, primAssert},
```

ersetzt in der Originaldatei Zeile 768:

```
 {"unsafePtrToInt", 1, primPtrToInt}, /* breaks the semantics */
```

5. Änderung:

```
 }
/* -----
 * Primitive for Debugging with Assertions
 * -----*/
```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

/* primAssert:
 * -----
 * Die Funktion primAssert definiert die eingebaute Funktion, auf deren
 * Grundlage die Funktionen assertB und assertF zum Ausdruck gebracht werden:
 *
 *     primitive primAssert :: String -> Bool -> a -> a
 *
 *     assertB :: String -> Bool -> a -> a
 *     assertB = primAssert
 *
 *     assertF :: String -> (a -> Bool) -> a -> a
 *     assertF msg func expr = assertB msg (func expr) expr
 *
 * Anmerkungen:
 * 1) Die Ueberpruefung der Zusicherung wird zurueckgestellt.
 *    -> Zusicherungen werden nur nach Beendigung einer Garbage
 *        Collection bzw. nach Beendigung der normalen Auswertung
 *        ueberprueft.
 * 2) Die Variable interimCheck oder die Variable finalCheck hat den
 *    Wert TRUE.
 * 2.1) Der Makro setjmp speichert fuer einen Sprung ausserhalb einer
 *    Routine in einer Variablen vom Typ jmp_buf Zustandsinformationen.
 *    Mit Hilfe der Funktion allocateEnv wird der entsprechende Speicher
 *    zur Verfuegung gestellt.
 * 2.2) Direkter Aufruf von setjmp.
 * 2.2.1) Auswertung des zweiten Arguments der Funktion primAssert.
 * 2.2.2) Die Ueberpruefung der Zusicherung hat den Wert True ergeben.
 * 2.2.3) Die Ueberpruefung der Zusicherung hat den Wert False ergeben.
 * 2.3) Ruecksprung zu der Funktion primAssert.
 *    -> Die Ueberpruefung der Zusicherung wurde abgebrochen.
 */
primFun(primAssert) {
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        if (projectionMode()) {
            saveDebugInfo(root);
            returnThdArg();
        }
        else {
            allocateEnv();

            if (setjmp(jumpBuffer->env) == 0) {
                eval(primArg(2));
                checkBool();
                if (whnfHead == nameTrue) {
                    returnThdArg();
                    wrapUp(root);
                }
                else
                    terminate(root);
            }
            else
                interrupt(root);
        }
    }
    else
        returnThdArg();
#else
    returnThdArg();
#endif
}

```

ersetzt in der Originaldatei Zeile 1602:

```

}

```

A.6 Änderungen in der Datei ~/Hugs98/src/machine.c

1. Änderung:

```

/*          occurs          */

#if DEBUGGING_WITH_ASSERTIONS
Bool  debugging      = FALSE;    /* TRUE => Die Funktion primAssert */
/*          verhaelt sich nicht   */
/*          ausschliesslich wie   */
/*          eine Projektion auf das */
/*          dritte Argument       */
Bool  deleteCAF      = FALSE;    /* TRUE => Vor einer neuen         */
/*          Auswertung werden die  */
/*          CAF-Definitionen       */
/*          geloescht              */
Bool  gcRequired     = FALSE;    /* TRUE => Nach der Ueberpruefung  */
/*          aller in der Check List */
/*          verzeichneten         */
/*          Zusicherungen wird eine */
/*          Garbage Collection      */
/*          durchgefuehrt         */
#endif

```

ersetzt in der Originaldatei Zeile 27:

```

/*          occurs          */

```

2. Änderung:

```

static Void  local evalString  Args((Cell));

#if DEBUGGING_WITH_ASSERTIONS
static Void  local saveNames    Args((Void));
static Void  local normalCodeExecution  Args((StackPtr, Name));
static Void  local restrictedCodeExecution  Args((StackPtr, Name));
static Bool  local canReduce    Args((Cell));
static Void  local checkAssertion  Args((StackPtr));
static Void  local printResultMessage  Args((Void));
#endif

```

ersetzt in der Originaldatei Zeile 176:

```

static Void  local evalString  Args((Cell));

```

3. Änderung:

```

#endif

/* -----
 * Debugging with Assertions
 * ----- */

#if DEBUGGING_WITH_ASSERTIONS
/* saveNames:
 * -----
 * Die Funktion saveNames speichert in der Variablen nameIOCase den Namen
 * der Funktion, die mit der Beendigung ihres Aufrufs signalisiert, dass
 * die Auswertung abgeschlossen ist.
 */
static Void local saveNames() {
    namePerformIO = cellAt(name(nameIORun).code + 7);
    nameIOCase    = cellAt(name(namePerformIO).code + 9);
}

/* normalCodeExecution:
 * -----

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```
* Die Funktion normalCodeExecution sorgt dafuer, dass ein Ausdruck reduziert
* wird. Das geschieht entweder durch den Aufruf einer eingebauten Funktion
* oder durch die Anweisung, den dazugehoerenden G-Code auszufuehren.
*/
static Void local normalCodeExecution(root, n)
StackPtr root; /* Zeiger auf ein Stack-Element */
Name n; { /* Funktionssymbol */
    if (name(n).primDef)
        (*name(n).primDef)(root);
    else
        run(name(n).code, root);
}

/* restrictedCodeExecution:
* -----
* Die Funktion restrictedCodeExecution steuert die Auswertung, falls der
* Anwender die Option +A gewaehlt hat und eine Zusicherung ueberprueft
* wird: Laesst die Markierung an der Wurzel des zu reduzierenden Ausdrucks
* eine Auswertung zu, setzt die Funktion restrictedCodeExecution die
* Ueberpruefung der Zusicherung durch den Aufruf der Funktion
* normalCodeExecution fort. Ansonsten bricht sie diese mittels Ruecksprung
* zu der Funktion primAssert ab.
*/
static Void local restrictedCodeExecution(root, n)
StackPtr root; /* Zeiger auf ein Stack-Element */
Name n; { /* Funktionssymbol */
    Cell rootElement;

    rootElement = stack(root); /* Wurzel des Redex */
    if (canReduce(rootElement) || isDfun(n))
        normalCodeExecution(root, n);
    else
        longjmp(jumpBuffer->env, 1);
}

/* canReduce:
* -----
* Bei der Ueberpruefung einer Zusicherung muss vor jeder Auswertung
* festgestellt werden, ob der betreffende Ausdruck reduziert werden darf.
* Das geschieht mit Hilfe der Funktion canReduce, indem die Markierung
* an dessen Wurzel betrachtet wird.
*/
static Bool local canReduce(n)
Cell n; { /* Cell-Wert */
    switch (debugMark(n)) {
        case NOT_MARKED : return TRUE;
                          break;
        case ASSERTION : return TRUE;
                          break;
        case NON_ASSERTION: return FALSE;
                          break;
        case CAF : if (isAssertion(n))
                      return TRUE;
                  else
                      return FALSE;
                  break;
    }
}

/* checkAssertion:
* -----
* Zusicherungen werden im Anschluss an eine Garbage Collection bzw.
* nach Beendigung der normalen Auswertung ueberprueft. Die Funktion
* checkAssertion dient in diesem Zusammenhang insbesondere dem Zweck,
* CAF-Definitionen, die nicht dem Bereich NON_ASSERTION zugeordnet
* wurden, temporaer zu markieren, sofern dies noch nicht geschehen ist,
* und die in der Check List an erster Stelle verzeichnete Zusicherung
* wiederherzustellen.
*/
static Void local checkAssertion(root)
StackPtr root; { /* Zeiger auf ein Stack-Element */
    if (interimCheck) {
        if (!emptyCheckList()) {
            reducedExpr = stack(root);
        }
    }
}
```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

        if (!gcRequired)
            gcRequired = TRUE;
        if (!haveTempCAFMarks)
            markEveryUnrestrictedCAF();
        restoreAssertion(root);
    }
    else {
        if (!emptyClipboard())
            restoreCheckList();
        if (gcRequired) {
            garbageCollect();
            gcRequired = FALSE;
        }
        if (haveTempCAFMarks)
            unmarkEveryUnrestrictedCAF();
        interimCheck = FALSE;
    }
}
else {
    if (!emptyCheckList()) {
        if (debugMessageRequired) {
            printDebugMessage();
            debugMessageRequired = FALSE;
        }
        if (!resultMessageRequired)
            resultMessageRequired = TRUE;
        restoreAssertion(root);
    }
    else {
        finalCheck = FALSE;
        if (traceError)
            traceError = FALSE;
        if (debugMessageRequired)
            debugMessageRequired = FALSE;
        if (resultMessageRequired) {
            printResultMessage();
            resultMessageRequired = FALSE;
        }
    }
}
}
}

/* restoreAssertion:
 * -----
 * Die Funktion restoreAssertion macht die Projektion auf das dritte
 * Argument der Funktion primAssert rueckgaengig, und zwar fuer die in
 * der Check List an erster Stelle verzeichnete Zusicherung.
 */
Void restoreAssertion(root)
StackPtr root; {                               /* Zeiger auf ein Stack-Element */
    stack(root) = checkList->root;
    fst(stack(root)) = checkList->leftSubgraph;
}

/* printDebugMessage:
 * -----
 * Wenn nach Beendigung der normalen Auswertung noch Zusicherungen
 * ueberprueft werden muessen, teilt die Funktion printDebugMessage
 * dies dem Anwender mit.
 */
Void printDebugMessage() {
    Printf("\n");
    Printf("\n");
    Printf("*****\n");
    Printf("*** Debug message ***\n");
    Printf("*****\n");
    Printf("\n");
    Printf("The evaluation process is completed.\n");
    if (numAssert == 1)
        Printf("There is %d assertion left to check.\n", numAssert);
    else
        Printf("There are %d assertions left to check.\n", numAssert);
    Printf("\n");
    Printf("Please wait ...\n");
}

```



```

}

/* printResultMessage:
 * -----
 * Wenn nach Beendigung der normalen Auswertung noch Zusicherungen
 * ueberprueft wurden, zeigt die Funktion printResultMessage an, ob
 * saemtliche Ueberpruefungen zu Ende gefuehrt werden konnten.
 */
static Void local printResultMessage() {
    Printf("\n");
    Printf("Result of the final check:\n");
    if (numPending == 0)
        Printf("All assertions are fulfilled\n");
    else {
        if (numPending == 1)
            Printf("%d assertion is not verifiable.\n", numPending);
        else
            Printf("%d assertions are not verifiable.\n", numPending);
        numPending = 0;
    }
}
#endif

```

ersetzt in der Originaldatei Zeile 1232:

```
#endif
```

4. Änderung:

```

#if DEBUGGING_WITH_ASSERTIONS
    Name saveName = n;
#endif
#if DEBUG_CODE
    if (debugCode) {
        Printf("%*sEntering name(%d): %s\n", base, "",
            n - NAMEMIN, textToStr(name(n).text));
    }
#endif
#else
#if DEBUG_CODE
    Name saveName = n;
    if (debugCode) {
        Printf("%*sEntering name(%d): %s\n", base, "",
            n - NAMEMIN, textToStr(name(n).text));
    }
#endif
#endif
#endif

```

ersetzt in der Originaldatei Zeile 1286-1292:

```

#if DEBUG_CODE
    Name saveName = n;
    if (debugCode) {
        Printf("%*sEntering name(%d): %s\n", base, "",
            n - NAMEMIN, textToStr(name(n).text));
    }
#endif

```

5. Änderung:

```
#if DEBUGGING_WITH_ASSERTIONS
    if (restrictEval())
        restrictedCodeExecution(root, n);
    else
        normalCodeExecution(root, n);
#else
    if (name(n).primDef) /* reduce */
        (*name(n).primDef)(root);
    else
        run(name(n).code, root);
#endif
```

ersetzt in der Originaldatei Zeile 1305-1308:

```
if (name(n).primDef) /* reduce */
    (*name(n).primDef)(root);
else
    run(name(n).code, root);
```

6. Änderung:

```
sp = root; /* continue... */

#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        if (saveName == nameIOCase) {
            finalCheck = TRUE;
            debugMessageRequired = TRUE;
        }

        if (!restrictEval() &&
            !projectionMode())
            checkAssertion(root);
    }
#endif
```

ersetzt in der Originaldatei Zeile 1312:

```
sp = root; /* continue... */
```

7. Änderung:

```
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        markCAF(saveName);
    }

    if (sp > base) {
        fun(top()) = n;
        if (debugging &&
            name(saveName).assignment == \
            ASSERTION)
            updateDebugMarks(top());
    }
#else
    if (sp > base)
        fun(top()) = n;
#endif
```

ersetzt in der Originaldatei Zeile 1330-1331:

```
if (sp > base)
    fun(top()) = n;
```

8. Änderung:

```
                snd(t) = pop();
#if DEBUGGING_WITH_ASSERTIONS
                if (debugging)
                    updateDebugMarks(t);
#endif
```

ersetzt in der Originaldatei Zeile 1488:

```
                snd(t) = pop();
```

9. Änderung:

```
                snd(t) = top();
#if DEBUGGING_WITH_ASSERTIONS
                if (debugging)
                    updateDebugMarks(t);
#endif
```

ersetzt in der Originaldatei Zeile 1495:

```
                snd(t) = top();
```

10. Änderung:

```
                snd(t) = pop();
#if DEBUGGING_WITH_ASSERTIONS
                if (debugging)
                    updateDebugMarks(t);
#endif
```

ersetzt in der Originaldatei Zeile 1502:

```
                snd(t) = pop();
```

11. Änderung:

```
                snd(t) = top();
#if DEBUGGING_WITH_ASSERTIONS
                if (debugging)
                    updateDebugMarks(t);
#endif
```

ersetzt in der Originaldatei Zeile 1509:

```
                snd(t) = top();
```

12. Änderung:

```
                evalError = catcherr;
#if DEBUGGING_WITH_ASSERTIONS
                if (firstRun) {
                    saveNames();
                    firstRun = FALSE;
                }
#endif
```

ersetzt in der Originaldatei Zeile 1591:

```
                evalError = catcherr;
```

13. Änderung:

```
    evalError = &catcherr;  
#if DEBUGGING_WITH_ASSERTIONS  
    if (firstRun) {  
        saveNames();  
        firstRun = FALSE;  
    }  
#endif
```

ersetzt in der Originaldatei Zeile 1600:

```
    evalError = &catcherr;
```

A.7 Änderungen in der Datei `~/Hugs98/src/storage.h`

1. Änderung:

```
#if DEBUGGING_WITH_ASSERTIONS
#include <setjmp.h>
#endif

typedef Int          Text;                /* text string          */
```

ersetzt in der Originaldatei Zeile 23:

```
typedef Int          Text;                /* text string          */
```

2. Änderung:

```
        Cell defn;
#if DEBUGGING_WITH_ASSERTIONS
        Int  assignment;                /* NIL | ASSERTION | NON_ASSERTION */
        Int  debugMarks;                /* NOT_DETERMINED | DETERMINED     */
#endif
```

ersetzt in der Originaldatei Zeile 545:

```
        Cell defn;
```

3. Änderung:

```
#endif

/* -----
 * Debugging with Assertions
 * ----- */

#if DEBUGGING_WITH_ASSERTIONS

/* ----- *
 * Variablen
 * ----- */

extern Name nameAssert;
extern Name nameAssertB;
extern Name nameAssertF;
extern Name namePerformIO;
extern Name nameIOCase;

extern Bool interimCheck;
extern Bool finalCheck;

extern Cell mainRoot;
extern Cell reducedExpr;

extern Bool systemCheckRequired;
extern Bool ignoreError;
extern Bool traceError;

extern Bool cleanUpRequired;

extern Int  numAssert;
extern Int  numPending;
extern Bool debugMessageRequired;
extern Bool resultMessageRequired;

extern Bool prevCAFHandling;

extern Bool haveTempCAFMarks;
```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

/* ----- *
 * Konstanten *
 * ----- */

#define NOT_MARKED 0
#define ASSERTION 1
#define NON_ASSERTION 2
#define CAF 3

#define NOT_DETERMINED 0
#define DETERMINED 1

/* ----- *
 * Funktionsprototypen *
 * ----- */

extern Void allocateEnv Args((Void));
extern Void freeEnv Args((Void));
extern Void freeAllEnv Args((Void));

extern Void saveDebugInfo Args((StackPtr));
extern Void deleteDebugInfo Args((Void));
extern Void deleteAllDebugInfos Args((Void));
extern Void cutDebugInfo Args((Void));
extern Void pasteDebugInfo Args((Void));
extern Void restoreCheckList Args((Void));

extern Void markExpression Args((Cell));
extern Bool isAssertion Args((Cell));
extern Void updateDebugMarks Args((Cell));
extern Void deleteAllDebugMarks Args((Void));

extern Void markCAF Args((Name));
extern Void markEveryUnrestrictedCAF Args((Void));
extern Void unmarkEveryUnrestrictedCAF Args((Void));

extern Void deleteEveryCAF Args((Void));
extern Void unassignEveryCAF Args((Void));

/* ----- *
 * Handhabung der Zustandsinformationen, die fuer einen Ruecksprung *
 * zu der Funktion primAssert in Variablen vom Typ jmp_buf *
 * gespeichert werden. *
 * ----- */

struct strBuffer {
    jmp_buf env;
    struct strBuffer *next;
};

typedef struct strBuffer BufferRecord;

extern BufferRecord *jumpBuffer;

/* ----- *
 * Handhabung der Informationen, die fuer das Wiederherstellen einer *
 * Zusicherung erforderlich sind. *
 * ----- */

struct strAssertion {
    Cell root;
    Cell leftSubgraph;
    struct strAssertion *next;
};

typedef struct strAssertion InfoRecord;

extern InfoRecord *checkList;
extern InfoRecord *clipboard;

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```
/* ----- *
 * Handhabung der Markierungen, mit denen beim Debuggen ueberprueft *
 * wird, ob ein Ausdruck reduziert werden darf. *
 * ----- */

extern Heap heapFourth;
extern Heap heapTopFourth;

/* ----- *
 * Makrodefinitionen *
 * ----- */

#define restrictEval() (jumpBuffer->next != NIL)
#define debugMark(n) heapTopFourth[n]
#define markInputExpr(n) markExpression(n)
#define saveRoot(n) (checkList->root = n)
#define saveLeftSubgraph(n) (checkList->leftSubgraph = fst(n))
#define emptyCheckList() (checkList->root == NIL)
#define emptyClipboard() (clipboard->root == NIL)
#define saveCAFHandling() (prevCAFHandling = deleteCAF)
#define newCAFHandling() (deleteCAF != prevCAFHandling)
#define isCAF(n) (whatIs(n) == NAME && name(n).arity == 0)
#define projectionMode() (!interimCheck && !finalCheck)
#define updateTempCAFMarks(n) markUnrestrictedDefinition(name(n).defn)
#endif
```

ersetzt in der Originaldatei Zeile 816:

```
#endif
```

A.8 Änderungen in der Datei ~/Hugs98/src/storage.c

1. Änderung:

```
#endif

#if DEBUGGING_WITH_ASSERTIONS
static Void local initializeJumpBuffer      Args((Void));
static Void local errorJumpBuffer          Args((Void));
static Void local initializeCheckList      Args((Void));
static Void local errorCheckList          Args((Void));
static Void local initializeClipboard      Args((Void));
static Void local errorClipboard           Args((Void));

static Void local restrictCAF              Args((Name));
static Bool local isClosure                Args((Cell));
static Void local markUnrestrictedDefinition Args((Cell));
static Void local unmarkUnrestrictedDefinition Args((Cell));

static Void local restorePendingAssertions Args((Void));
static Void local markPendingAssertions   Args((Void));
static Void local undoRestoreProcess      Args((Void));

static Void local systemCheck             Args((Void));
#endif
```

ersetzt in der Originaldatei Zeile 49:

```
#endif
```

2. Änderung:

```
    name(nameHw).defn      = NIL;
#if DEBUGGING_WITH_ASSERTIONS
    name(nameHw).assignment = NIL;
    name(nameHw).debugMarks = NOT_DETERMINED;
#endif
```

ersetzt in der Originaldatei Zeile 444:

```
    name(nameHw).defn      = NIL;
```

3. Änderung:

```
#endif /* !GIMME_STACK_DUMPS */

/* -----
 * Debugging with Assertions
 * ----- */

#if DEBUGGING_WITH_ASSERTIONS

/* ----- *
 * Variablen *
 * ----- */

Name nameAssert;          /* Name der Funktion primAssert */
Name nameAssertB;        /* Name der Funktion assertB */
Name nameAssertF;        /* Name der Funktion assertF */
Name namePerformIO;      /* Name der lokal definierten
                          /* Funktion performIO */
Name nameIOCase;         /* Name der Funktion, die fuer den
                          /* case-Ausdruck in der Definition
                          /* der Funktion performIO gebildet
                          /* wird */
```


Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

Bool interimCheck = FALSE;           /* TRUE => Nach Beendigung der      */
                                      /* Garbage Collection                */
                                      /* werden die in der Check List     */
                                      /* verzeichneten Zusicherungen     */
                                      /* ueberprueft                      */
Bool finalCheck   = FALSE;           /* TRUE => Nach Beendigung der      */
                                      /* normalen Auswertung              */
                                      /* werden die in der Check List     */
                                      /* verzeichneten Zusicherungen     */
                                      /* ueberprueft                      */

Cell mainRoot;                       /* Wurzel des Graphen, der an den   */
                                      /* Auswertungsmechanismus          */
                                      /* uebergeben wird                 */
Cell reducedExpr;                    /* Wurzel eines reduzierten         */
                                      /* Teilausdrucks                   */

Bool systemCheckRequired = FALSE;    /* TRUE => Die Programmausfuehrung  */
                                      /* wurde aufgrund eines Fehlers      */
                                      /* abgebrochen                      */
Bool ignoreError    = FALSE;        /* TRUE => Die Programmausfuehrung  */
                                      /* wurde abgebrochen, weil die     */
                                      /* Ueberpruefung einer Zusicherung */
                                      /* als Ergebnis den Wert False     */
                                      /* geliefert hat                     */
Bool traceError     = FALSE;        /* TRUE => Nach einem Programm-    */
                                      /* abbruch, der nicht auf einen    */
                                      /* Aufruf der Funktion terminate   */
                                      /* zurueckzufuehren ist, werden   */
                                      /* die in der Check List verzeichn- */
                                      /* eten Zusicherungen ueberprueft  */

Bool cleanUpRequired = FALSE;        /* TRUE => Die vorhergehende       */
                                      /* Auswertung fand unter Option +A */
                                      /* statt                             */

Int  numAssert = 0;                 /* Anzahl der in der Check List    */
                                      /* verzeichneten Zusicherungen     */
Int  numPending = 0;               /* Anzahl der Zusicherungen, deren */
                                      /* Ueberpruefung nicht zu Ende     */
                                      /* gefuehrt werden konnte          */
Bool debugMessageRequired = FALSE;  /* TRUE => Nach Beendigung der     */
                                      /* normalen Auswertung wird angezei- */
                                      /* gt, wieviele Zusicherungen noch  */
                                      /* einer Ueberpruefung unterzogen   */
                                      /* werden muessen                   */
Bool resultMessageRequired = FALSE; /* TRUE => Nach Beendigung der     */
                                      /* normalen Auswertung wird angezei- */
                                      /* gt, ob saemtliche ueberprueften */
                                      /* Zusicherungen erfuehlt sind     */

Bool prevCAFHandling;              /* Setting fuer die Option d in der */
                                      /* vorhergehenden Auswertung        */

Bool haveTempCAFMarks = FALSE;     /* TRUE => Die Graphen derjenigen  */
                                      /* CAF's, die nicht dem Bereich     */
                                      /* ASSERTION und nicht dem Bereich  */
                                      /* NON_ASSERTION zuzuordnen sind,   */
                                      /* wurden temporaer markiert       */

/* ----- *
 * Initialisierung *

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

* ----- */
struct strBuffer *jumpBuffer;          /* Verkettete Liste, in der die      */
                                        /* fuer die Ueberpruefung der        */
                                        /* Zusicherungen erforderlichen     */
                                        /* Variablen vom Typ jmp_buf        */
                                        /* zusammengefasst werden           */

InfoRecord *checkList;                /* Verkettete Liste, in der die      */
                                        /* Zusicherungen verzeichnet        */
                                        /* werden, deren Ueberpruefung      */
                                        /* zurueckgestellt bzw. abgebrochen */
                                        /* wurde                             */

InfoRecord *clipboard;                /* Verkettete Liste, in die        */
                                        /* temporaer Elemente der Check    */
                                        /* List verschoben werden koennen   */

Heap heapFourth;                      /* Zusaetzlicher Heap, in dem die   */
                                        /* fuer die Ueberpruefung der      */
                                        /* Zusicherungen erforderlichen     */
                                        /* Markierungen gespeichert werden  */

Heap heapTopFourth;

/* initializeJumpBuffer:
* -----
* Die Funktion initializeJumpBuffer initialisiert die Liste jumpBuffer.
*/
static Void local initializeJumpBuffer() {
    jumpBuffer = (BufferRecord *) malloc(sizeof(BufferRecord));

    if (jumpBuffer == NULL)
        errorJumpBuffer();
    else
        jumpBuffer->next = 0;
}

static Void local errorJumpBuffer() {
    ERRMSG(0) "Cannot allocate storage space " ETHEN
    ERRTXT "for \"Debugging with Assertions\".\n" ETHEN
    ERRTXT "Assignment: jumpBuffer"
    EEND;
}

/* initializeCheckList:
* -----
* Die Funktion initializeCheckList initialisiert die Liste checkList.
*/
static Void local initializeCheckList() {
    checkList = (InfoRecord *) malloc(sizeof(InfoRecord));

    if (checkList == NULL)
        errorCheckList();
    else {
        checkList->root          = 0;
        checkList->leftSubgraph = 0;
        checkList->next         = 0;
    }
}

static Void local errorCheckList() {
    ERRMSG(0) "Cannot allocate storage space " ETHEN
    ERRTXT "for \"Debugging with Assertions\".\n" ETHEN
    ERRTXT "Assignment: Check List"
    EEND;
}

/* initializeClipboard:
* -----
* Die Funktion initializeClipboard initialisiert die Liste clipboard.
*/
static Void local initializeClipboard() {
    clipboard = (InfoRecord *) malloc(sizeof(InfoRecord));
}

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

    if (clipboard == NULL)
        errorClipboard();
    else {
        clipboard->root          = 0;
        clipboard->leftSubgraph = 0;
        clipboard->next         = 0;
    }
}

static Void local errorClipboard() {
    ERRMSG(0) "Cannot allocate storage space " ETHEN
    ERRTEXT  "for \"Debugging with Assertions\".\n" ETHEN
    ERRTEXT  "Assignment: Clipboard"
    EEND;
}

/* -----
 * Handhabung der Zustandsinformationen, die fuer einen Ruecksprung
 * zu der Funktion primAssert in Variablen vom Typ jmp_buf
 * gespeichert werden.
 * ----- */

/* allocateEnv:
 * -----
 * Die Funktion allocateEnv stellt den Speicherplatz zur Verfuegung, der
 * benoetigt wird, um Zustandsinformationen in einer Variablen vom Typ
 * jmp_buf speichern zu koennen.
 */
Void allocateEnv() {
    BufferRecord *temp;

    temp = jumpBuffer;
    jumpBuffer = (BufferRecord *) malloc(sizeof(BufferRecord));

    if (jumpBuffer == NULL)
        errorJumpBuffer();
    else
        jumpBuffer->next = temp;
}

/* freeEnv:
 * -----
 * Die Funktion freeEnv entfernt aus der Liste jumpBuffer das erste Element
 * und gibt den durch dieses Element belegten Speicher wieder frei.
 */
Void freeEnv() {
    BufferRecord *temp;

    temp          = jumpBuffer;
    jumpBuffer    = jumpBuffer->next;
    free(temp);
}

/* freeAllEnv:
 * -----
 * Die Funktion freeAllEnv entfernt aus der Liste jumpBuffer alle Elemente
 * und gibt den durch diese Elemente belegten Speicher wieder frei.
 */
Void freeAllEnv() {
    while (jumpBuffer->next != NIL)
        freeEnv();
}

/* -----
 * Handhabung der Informationen, die fuer das Wiederherstellen einer
 * Zusicherung erforderlich sind.
 * ----- */

/* saveDebugInfo:
 * -----
 * Wenn die Ueberpruefung einer Zusicherung zurueckgestellt bzw. abgebrochen

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```
* wird, speichert die Funktion saveDebugInfo alle Informationen, die fuer
* die spaetere Wiederherstellung der Zusicherung erforderlich sind:
* 1. Die Heap-Adresse des Wurzelknotens
* 2. Die Heap-Adresse des linken Nachfolgers des Wurzelknotens
*/
Void saveDebugInfo(root)
StackPtr root; { /* Zeiger auf ein Stack-Element */
    InfoRecord *newRecord;
    Cell rootElement;

    newRecord = (InfoRecord *) malloc(sizeof(InfoRecord));

    if (newRecord == NULL)
        errorCheckList();
    else {
        newRecord->next = checkList;
        checkList = newRecord;
        rootElement = stack(root);
        saveRoot(rootElement);
        saveLeftSubgraph(rootElement);
        if (interimCheck)
            cutDebugInfo();
        ++numAssert;
    }
}

/* deleteDebugInfo:
* -----
* Die Funktion deleteDebugInfo entfernt aus der Liste checkList das erste
* Element und gibt den durch dieses Element belegten Speicher wieder frei.
* Das heisst, die Informationen ueber die Zusicherung, die es bezeichnet,
* werden geloescht.
*/
Void deleteDebugInfo() {
    InfoRecord *temp;

    temp = checkList;
    checkList = checkList->next;
    free(temp);
    --numAssert;
}

/* deleteAllDebugInfos:
* -----
* Die Funktion deleteAllDebugInfos entfernt aus der Liste checkList
* alle Elemente und gibt den durch diese Elemente belegten Speicher
* wieder frei. Das heisst, die Informationen ueber die Zusicherungen,
* die sie bezeichnen, werden geloescht.
*/
Void deleteAllDebugInfos() {
    restoreCheckList();

    while (!emptyCheckList())
        deleteDebugInfo();
}

/* cutDebugInfo:
* -----
* Die Funktion cutDebugInfo entfernt aus der Liste checkList das erste
* Element und fuegt es der Liste clipboard hinzu.
*/
Void cutDebugInfo() {
    InfoRecord *temp;

    temp = checkList;
    checkList = checkList->next;
    temp->next = clipboard;
    clipboard = temp;
}

/* pasteDebugInfo:
* -----
* Die Funktion pasteDebugInfo entfernt aus der Liste clipboard das erste
* Element und fuegt es der Liste checkList hinzu.
```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

*/
Void pasteDebugInfo() {
    InfoRecord *temp;

    temp      = clipboard;
    clipboard = clipboard->next;
    temp->next = checkList;
    checkList = temp;
}

/* restoreCheckList:
 * -----
 * Die Funktion restoreCheckList entfernt aus der Liste clipboard alle
 * Elemente und fuegt sie der Liste checkList hinzu.
 */
Void restoreCheckList() {
    while (!emptyClipboard())
        pasteDebugInfo();
}

/* -----
 * Handhabung der Markierungen, mit denen beim Debuggen ueberprueft
 * wird, ob ein Ausdruck reduziert werden darf.
 *
 * 1. Funktionen, die nicht ausschliesslich der Markierung einer CAF
 * dienen.
 * -----
 */

/* markExpression:
 * -----
 * Die Funktion markExpression markiert einen beliebigen Graphen.
 * Und zwar mit dem Ziel, dass der Auswertungsmechanismus anhand der
 * Markierungen erkennt, ob ein Ausdruck im Rahmen der Ueberpruefung einer
 * Zusicherung reduziert werden darf oder nicht.
 */
Void markExpression(n)
Cell n; { /* Cell-Wert */
    Cell left;
    Cell right;

    if (!isGenPair(n)) {
        if (isCAF(n))
            restrictCAF(n);
        else
            return;
    }
    else {
        if (debugMark(n) == NOT_MARKED || debugMark(n) == CAF) {
            left = fst(n);
            right = snd(n);
            if (isAssertion(n)) {
                debugMark(n) = ASSERTION;
                markExpression(right);
            }
            else if (isGenPair(left)) {
                debugMark(n) = NON_ASSERTION;
                markExpression(left);
                markExpression(right);
            }
            else if (isNull(left) || left == INDIRECT) {
                debugMark(n) = NON_ASSERTION;
                markExpression(right);
            }
            else {
                debugMark(n) = NON_ASSERTION;
                if (left >= BCSTAG) {
                    if (isCAF(left))
                        restrictCAF(left);
                    markExpression(right);
                }
            }
            else
                return;
        }
    }
}

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

        }
        else
            return;
    }
}

/* isAssertion:
 * -----
 * Die Funktion isAssertion stellt fest, ob es sich bei einem gegebenen
 * Ausdruck um eine zu ueberpruefende Zusicherung handelt.
 */
Bool isAssertion(n)
Cell n; { /* Cell-Wert */
    Cell left;
    Cell leftLeft;
    Name func;

    if (isGenPair(n)) {
        left = fst(n);
        if (isGenPair(left)) {
            leftLeft = fst(left);
            if (isGenPair(leftLeft)) {
                if (whatIs(fst(leftLeft)) == NAME) {
                    func = fst(leftLeft);
                    if (func == nameAssert ||
                        func == nameAssertB ||
                        func == nameAssertF)
                        return TRUE;
                    else
                        return FALSE;
                }
                else
                    return FALSE;
            }
            else
                return FALSE;
        }
        else
            return FALSE;
    }
    else
        return FALSE;
}

/* updateDebugMarks:
 * -----
 * Die Funktion updateDebugMarks aktualisiert die fuer das Debuggen
 * erforderlichen Markierungen, wenn die Wurzel einer Funktionsanwendung
 * durch eine Update-Anweisung ueberschrieben wurde. Und zwar beginnend
 * an ebendiesem Knoten.
 */
Void updateDebugMarks(n)
Cell n; { /* Cell-Wert */
    switch (debugMark(n)) {
        case NOT_MARKED : doNothing();
                        break;
        case ASSERTION  : doNothing();
                        break;
        case NON_ASSERTION: if (isAssertion(n)) {
                            debugMark(n) = ASSERTION;
                            markExpression(snd(n));
                        }
                        else {
                            markExpression(fst(n));
                            markExpression(snd(n));
                        }
                        break;
        case CAF        : doNothing();
                        break;
    }
}

/* deleteAllDebugMarks:
 * -----

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

* Die Funktion deleteAllDebugMarks versieht alle Heap-Zellen mit der
* Markierung NOT_MARKED.
*/
Void deleteAllDebugMarks() {
    Int i;

    for (i = 1; i <= heapSize; ++i) {
        if (debugMark(-i) != NOT_MARKED)
            debugMark(-i) = NOT_MARKED;
    }
}

/* -----
* Handhabung der Markierungen, mit denen beim Debuggen ueberprueft
* wird, ob ein Ausdruck reduziert werden darf.
*
* 2. Funktionen, die ausschliesslich der Markierung einer CAF
* dienen.
* ----- */

/* restrictCAF:
* -----
* Die Funktion restrictCAF ordnet eine CAF dem Bereich NON_ASSERTION zu,
* falls der Markierungsalgorithmus deren Funktionssymbol ueber einen
* Knoten erreicht hat, der mit der Markierung NON_ASSERTION versehen wurde.
*/
static Void local restrictCAF(n)
Name n; { /* Funktionssymbol */
    if (name(n).assignment == NIL && !isDfun(n)) {
        if (name(n).defn == NIL)
            name(n).assignment = NON_ASSERTION;
        else {
            if (isClosure(name(n).defn))
                name(n).assignment = ASSERTION;
            else {
                name(n).assignment = NON_ASSERTION;
                markExpression(name(n).defn);
            }
        }
    }
    else
        doNothing();
}

/* markCAF:
* -----
* Die Funktion markCAF markiert eine beliebige CAF. Und zwar mit dem
* Ziel, dass der Auswertungsmechanismus anhand der Markierungen erkennt,
* ob ein Ausdruck im Rahmen der Ueberpruefung einer Zusicherung reduziert
* werden darf oder nicht.
*/
Void markCAF(n)
Name n; { /* Funktionssymbol */
    if (isClosure(name(n).defn)) {
        if (name(n).assignment != ASSERTION)
            name(n).assignment = ASSERTION;
        else
            doNothing();
    }
    else
        switch (name(n).assignment) {
            case NON_ASSERTION:
                if (name(n).debugMarks != DETERMINED) {
                    markExpression(name(n).defn);
                    name(n).debugMarks = DETERMINED;
                }
                else
                    doNothing();
            case NIL:
                if (interimCheck && !isDfun(n))
                    updateTempCAFMarks(n);
                else
                    doNothing();
        }
}

```

```

    }
}

/* isClosure:
 * -----
 * Die Funktion isClosure stellt fest, ob es sich bei einem gegebenen
 * Ausdruck um eine partielle Anwendung der Funktion assertB, assertF
 * bzw. primAssert handelt, und zwar um eine partielle Anwendung der
 * folgenden Form:
 *
 *      @
 *     / \
 *    @  arg2
 *   / \
 *  func arg1
 */
static Bool local isClosure(n)
Cell n; { /* Cell-Wert */
    Cell left;
    Name func;

    if (isGenPair(n)) {
        left = fst(n);
        if (isGenPair(left)) {
            if (whatIs(fst(left)) == NAME) {
                func = fst(left);
                if (func == nameAssert ||
                    func == nameAssertB ||
                    func == nameAssertF)
                    return TRUE;
                else
                    return FALSE;
            }
            else
                return FALSE;
        }
        else
            return FALSE;
    }
    else
        return FALSE;
}

/* markEveryUnrestrictedCAF:
 * -----
 * Vor der Ueberpruefung einer Zusicherung werden mit Hilfe der Funktion
 * markEveryUnrestrictedCAF die Graphen der noch nicht dem Bereich
 * NON_ASSERTION zugeordneten CAF's markiert, und zwar ausschliesslich
 * mit der Markierung CAF (Ausnahme: Die Graphen der dem Bereich ASSERTION
 * zugeordneten CAF's).
 */
Void markEveryUnrestrictedCAF() {
    Int i;

    for (i = NAMEMIN; i < nameHw; ++i) {
        if (name(i).arity == 0 &&
            name(i).assignment == NIL &&
            name(i).defn != NIL &&
            !isDfun(i))
            markUnrestrictedDefinition(name(i).defn);
    }

    haveTempCAFMarks = TRUE;
}

static Void local markUnrestrictedDefinition(n)
Cell n; { /* Cell-Wert */
    Cell left;
    Cell right;

    if (!isGenPair(n))
        return;
    else {
        if (debugMark(n) == NOT_MARKED) {

```


Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

        left = fst(n);
        right = snd(n);
        debugMark(n) = CAF;
        if (isAssertion(n))
            markUnrestrictedDefinition(right);
        else if (isGenPair(left)) {
            markUnrestrictedDefinition(left);
            markUnrestrictedDefinition(right);
        }
        else if (isNull(left) || left == INDIRECT)
            markUnrestrictedDefinition(right);
        else {
            if (left >= BCSTAG)
                markUnrestrictedDefinition(right);
            else
                return;
        }
    }
    else
        return;
}

}

}

/* unmarkEveryUnrestrictedCAF:
 * -----
 * Vor dem Zurueckschalten in den Projektionsmodus werden mit Hilfe der
 * Funktion unmarkEveryUnrestrictedCAF die Graphen der noch nicht dem
 * Bereich NON_ASSERTION zugeordneten CAF's mit der Markierung NOT_MARKED
 * versehen (Ausnahme: Die Graphen der dem Bereich ASSERTION zugeordneten
 * CAF's).
 */
Void unmarkEveryUnrestrictedCAF() {
    Int i;

    for (i = NAMEMIN; i < nameHw; ++i) {
        if (name(i).arity == 0 &&
            name(i).assignment == NIL &&
            name(i).defn != NIL &&
            !isDfun(i))
            unmarkUnrestrictedDefinition(name(i).defn);
    }

    haveTempCAFMarks = FALSE;
}

static Void local unmarkUnrestrictedDefinition(n)
Cell n; { /* Cell-Wert */
    Cell left;
    Cell right;

    if (!isGenPair(n))
        return;
    else {
        if (debugMark(n) == CAF) {
            left = fst(n);
            right = snd(n);
            debugMark(n) = NOT_MARKED;
            if (isAssertion(n))
                unmarkUnrestrictedDefinition(right);
            else if (isGenPair(left)) {
                unmarkUnrestrictedDefinition(left);
                unmarkUnrestrictedDefinition(right);
            }
            else if (isNull(left) || left == INDIRECT)
                unmarkUnrestrictedDefinition(right);
            else {
                if (left >= BCSTAG)
                    unmarkUnrestrictedDefinition(right);
                else
                    return;
            }
        }
    }
    else
        return;
}

```

```

    }
}

/* ----- *
 * Handhabung der CAF's im Rahmen der Funktion prepareEvaluation *
 * ----- */

/* deleteEveryCAF:
 * -----
 * Die Funktion deleteEveryCAF loescht vor einer Auswertung saemtliche
 * CAF-Definitionen. Das geschieht in vier Faellen:
 * 1. Nach einem Wechsel von der Option +A zur Option -A.
 * 2. Nach einem Wechsel von der Option -A zur Option +A.
 * 3. Wenn sowohl die Option +A als auch die Option +d gewaehlt wurde.
 * 4. Unter der Option +A nach einem Wechsel von der Option +d zur Option -d.
 */
Void deleteEveryCAF() {
    Int i;

    for (i = NAMEMIN; i < nameHw; ++i) {
        if (name(i).arity == 0) {
            if (name(i).assignment != NIL)
                name(i).assignment = NIL;
            if (name(i).debugMarks == DETERMINED)
                name(i).debugMarks = NOT_DETERMINED;
            if (name(i).defn != NIL)
                name(i).defn = NIL;
        }
    }
}

/* unassignEveryCAF:
 * -----
 * Die Funktion unassignEveryCAF setzt die Zuordnung aller CAF's auf NIL
 * zurueck (Option: +A / -d).
 */
Void unassignEveryCAF() {
    Int i;

    for (i = NAMEMIN; i < nameHw; ++i) {
        if (name(i).arity == 0) {
            if (name(i).assignment != NIL)
                name(i).assignment = NIL;
            if (name(i).debugMarks == DETERMINED)
                name(i).debugMarks = NOT_DETERMINED;
        }
    }
}

/* ----- *
 * Garbage Collection *
 * ----- */

/* restorePendingAssertions:
 * -----
 * Die Funktion restorePendingAssertions stellt alle in der Check List
 * und im Clipboard verzeichneten Zusicherungen wieder her.
 */
static Void local restorePendingAssertions() {
    InfoRecord *temp;

    if (!emptyCheckList()) {
        if (restrictEval())
            temp = checkList->next;
        else
            temp = checkList;
        while (temp->root != NIL) {
            fst(temp->root) = temp->leftSubgraph;
            temp = temp->next;
        }
    }
}

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```

    if (!emptyClipboard()) {
        temp = clipboard;
        while (temp->root != NIL) {
            fst(temp->root) = temp->leftSubgraph;
            temp = temp->next;
        }
    }
}

/* markPendingAssertions:
 * -----
 * Die Funktion markPendingAssertions stellt sicher, dass die Heap-Knoten,
 * die zu den in der Check List und im Clipboard verzeichneten Zusicherungen
 * gehoeren, nicht in die Freispeicherliste aufgenommen werden.
 */
static Void local markPendingAssertions() {
    InfoRecord *temp;

    if (!emptyCheckList()) {
        if (restrictEval())
            temp = checkList->next;
        else
            temp = checkList;
        while (temp->root != NIL) {
            mark(temp->root);
            temp = temp->next;
        }
    }

    if (!emptyClipboard()) {
        temp = clipboard;
        while (temp->root != NIL) {
            mark(temp->root);
            temp = temp->next;
        }
    }
}

/* undoRestoreProcess:
 * -----
 * Die Funktion undoRestoreProcess macht die Wiederherstellung der in der
 * Check List und im Clipboard verzeichneten Zusicherungen durch die Funktion
 * restorePendingAssertions wieder rueckgaengig.
 */
static Void local undoRestoreProcess() {
    InfoRecord *temp;

    if (!emptyCheckList()) {
        if (restrictEval())
            temp = checkList->next;
        else
            temp = checkList;
        while (temp->root != NIL) {
            fst(temp->root) = INDIRECT;
            temp = temp->next;
        }
    }

    if (!emptyClipboard()) {
        temp = clipboard;
        while (temp->root != NIL) {
            fst(temp->root) = INDIRECT;
            temp = temp->next;
        }
    }
}

/* ----- *
 * Anpassung des RESET-Kommandos *
 * ----- */

/* systemCheck:
 * -----

```

Anhang A – Die Änderungen an der Implementierung des Haskell-Interpreters Hugs im Überblick

```
* Die Funktion systemCheck bewirkt fuer verschiedene Systemkomponenten, die
* bei der Ueberpruefung von Zusicherungen bedeutsam sind, ein RESET. Und
* zwar genau dann, wenn die Programmausfuehrung abgebrochen wurde, der
* Grund hierfuer aber nicht ein Aufruf der Funktion terminate ist.
*/
static Void local systemCheck() {
    Int i;

    if (ignoreError)
        ignoreError = FALSE;
    else {
        if (jumpBuffer->next != NIL)
            freeAllEnv();

        if (numAssert != 0)
            deleteAllDebugInfos();

        if (interimCheck) {
            interimCheck = FALSE;
            if (gcRequired)
                gcRequired = FALSE;
            if (haveTempCAFMarks)
                unmarkEveryUnrestrictedCAF();
        }

        if (finalCheck) {
            finalCheck = FALSE;
            if (traceError)
                traceError = FALSE;
            if (numPending != 0)
                numPending = 0;
            if (debugMessageRequired)
                debugMessageRequired = FALSE;
            if (resultMessageRequired)
                resultMessageRequired = FALSE;
        }
    }

    systemCheckRequired = FALSE;
}
#endif
```

ersetzt in der Originaldatei Zeile 965:

```
#endif /* !GIMME_STACK_DUMPS */
```

4. Änderung:

```
        c        = freeList;
#if DEBUGGING_WITH_ASSERTIONS
        if (debugging && !firstRun && !finalCheck) {
            interimCheck = TRUE;
        }
#endif
```

ersetzt in der Originaldatei Zeile 1365:

```
        c        = freeList;
```

5. Änderung:

```
#endif
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        restorePendingAssertions();
    }
#endif
```

ersetzt in der Originaldatei Zeile 1566:

```
#endif
```

6. Änderung:

```
        freeList = -i;
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        if (debugMark(-i) != NOT_MARKED)
            debugMark(-i) = NOT_MARKED;
    }
#endif
```

ersetzt in der Originaldatei Zeile 1694:

```
        freeList = -i;
```

7. Änderung:

```
#endif

#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        undoRestoreProcess();
    }
#endif
```

ersetzt in der Originaldatei Zeile 1755:

```
#endif
```

8. Änderung:

```
        heapTopSnd = heapSnd + heapSize;
#if DEBUGGING_WITH_ASSERTIONS
        heapTopFourth = heapFourth + heapSize;
#endif
```

ersetzt in der Originaldatei Zeile 2696:

```
        heapTopSnd = heapSnd + heapSize;
```

9. Änderung:

```
#endif
#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        if (systemCheckRequired)
            systemCheck();
        if (!firstRun)
            deleteAllDebugMarks();
    }
#endif
```

ersetzt in der Originaldatei Zeile 2721:

```
#endif
```

10. Änderung:

```
    }

#if DEBUGGING_WITH_ASSERTIONS
    if (debugging) {
        start();
        if (interimCheck && restrictEval())
            mark(reducedExpr);
        if (finalCheck && !traceError && restrictEval())
            mark(mainRoot);
        markPendingAssertions();
        end("Check List and Clipboard", 2);
    }
#endif
```

ersetzt in der Originaldatei Zeile 2813:

```
    }
```

11. Änderung:

```
    scriptHw = 0;

#if DEBUGGING_WITH_ASSERTIONS
    initializeJumpBuffer();
    initializeCheckList();
    initializeClipboard();

    heapFourth = heapAlloc(heapSize);

    if (heapFourth==(Heap)0) {
        ERRMSG(0) "Cannot allocate heap storage (%d cells)",
            heapSize
        EEND;
    }

    heapTopFourth = heapFourth + heapSize;
#endif
```

ersetzt in der Originaldatei Zeile 2929:

```
    scriptHw = 0;
```

Anhang B

Beispielauswertungen

Der Anhang B dokumentiert Beispielauswertungen, die mit dem modifizierten Haskell-Interpreter Hugs, Version: Mai 1999, vorgenommen wurden.

1. Beispiel:

Die Korrektheit der Funktion `qSort` soll mit Hilfe einer Zusicherung überprüft werden. Diese Funktion implementiert den Quicksort-Algorithmus:

```
qSort :: Ord a => [a] -> [a]
qSort x = assertB msg (isSorted x res) res
  where
    msg = "Funktion qSort fehlerhaft definiert"
    res = qSort' x
    qSort' :: Ord a => [a] -> [a]
    qSort' [] = []
    qSort' (x:xs) = qSort' [y | y<-xs, y<=x] ++
                     [x] ++
                     qSort' [y | y<-xs, y>x]

-- !! Die Definition der Funktion qSort' ist fehlerhaft !!
-- Ihre letzte Zeile muesste lauten:
--           qSort' [y | y<-xs, y>x]

isSorted :: Ord a => [a] -> [a] -> Bool
isSorted x y = isOrdered y && isPermutation x y

isOrdered :: Ord a => [a] -> Bool
isOrdered x = foldr1 (&&) (convert x)
  where
    convert :: Ord a => [a] -> [Bool]
    convert [] = [True]
    convert [x] = [True]
    convert y@(x:xs) = zipWith (<=) y xs

isPermutation :: Ord a => [a] -> [a] -> Bool
isPermutation [] [] = True
isPermutation (x:xs) z@(y:ys) = isPermutation xs (delete x z)
isPermutation _ _ = False
```

Der Fehler in der Definition der Funktion `qSort'` bewirkt, daß die Teilliste der Elemente, die größer als das Pivotelement sind, nicht sortiert wird.

Beispielauswertungen unter der Option `+A/-d`:

```
# _____
# ||   ||   ||   ||   ||   ||
# ||___||   ||___||   ||___||   ||___||
# ||---||       ||___||
# ||   ||
# ||   || Version: May 1999
#
# ...
#
# Example.lhs
# Type :? for help
# Example> qSort [1, 2, 3, 4]
# [1,2,3,4]
```

Hugs 98: Based on the Haskell 98 standard
 Copyright (c) 1994-1999
 World Wide Web: <http://haskell.org/hugs>
 Report bugs to: hugs-bugs@haskell.org

```

#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> qSort [3, 1, 2, 4]
# [1,2,3,4]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> qSort [2, 1, 4, 3]
# [1,2,4,3]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Funktion qSort fehlerhaft definiert
#
# Example>

```

2. Beispiel:

Die Korrektheit der Funktion `qSort'` soll mit Hilfe einer Zusicherung überprüft werden. Diese Funktion implementiert genau wie die Funktion `qSort` den Quicksort-Algorithmus, unterscheidet sich von letzterer aber dadurch, daß die Funktion `assertB` nur partiell auf zwei Argumente angewendet wird:

```

qSort' :: Ord a => [a] -> [a]
qSort' x = assert res
  where
    assert = assertB msg (isSorted x res)
    msg = "Funktion qSort' fehlerhaft definiert"
    res = qSort'' x
    qSort'' :: Ord a => [a] -> [a]
    qSort'' [] = []
    qSort'' (x:xs) = qSort'' [y | y<-xs, y<=x] ++
                      [x] ++
                      [y | y<-xs, y>x]

-- !! Die Definition der Funktion qSort'' ist fehlerhaft !!

```



```
# Example> prim 5
# [2,3,5,7,9]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Das Ergebnis der Funktion prim beinhaltet auch Nicht-Primzahlen
#
# Example>
```

4. Beispiel:

Die Korrektheit der Funktion `prim'` soll mit Hilfe einer Zusicherung überprüft werden. Die Funktion `prim'` erwartet genau wie die Funktion `prim` als Argument eine natürliche Zahl `n` und liefert als Ergebnis eine Liste, die aus den ersten `n` Primzahlen besteht. Die Funktion `prim'` unterscheidet sich von der Funktion `prim` dadurch, daß die Funktion `assertF` nur partiell auf zwei Argumente angewendet wird.

```
prim' n = take n (assert (prim'' [2..]))
  where
    assert = assertF msg allPrim
    msg = "Das Ergebnis der Funktion prim' beinhaltet " ++
          "auch Nicht-Primzahlen"
    prim'' :: [Int] -> [Int]
    prim'' (x:xs)
      = x : [ y | y <- xs , y `rem` x /= 0 ]

-- !! Die Definition der Funktion prim'' ist fehlerhaft !!
-- Ihre letzte Zeile muesste lauten:
--      = x : prim'' [ y | y <- xs , y `rem` x /= 0 ]
```

Beispielauswertungen unter der Option `+A/-d`:

```
# _____
# ||  ||  ||  ||  ||  ||  ||  ||
# ||__||  ||__||  ||__||  ||__||
# ||---||      ||
# ||  ||
# ||  || Version: May 1999
#
# ...
#
# Example.lhs
# Type :? for help
# Example> prim' 4
# [2,3,5,7]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
```

```

# 1 assertion is not verifiable.
#
# Example> prim' 5
# [2,3,5,7,9]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Das Ergebnis der Funktion prim' beinhaltet auch Nicht-Primzahlen
#
# Example>

```

5. Beispiel:

Die nachstehend definierte Funktion `gauss` erwartet als Argument eine Liste, die aus positiven, natürlichen Zahlen besteht und berechnet als Ergebnis für jede Zahl `n` aus dieser Liste den Wert $1 + 2 + \dots + n$. Mit Hilfe einer Zusicherung soll sichergestellt werden, daß alle Elemente der Argument-Liste größer als Null sind, um eine Nicht-Terminierung der Berechnung zu verhindern.

```

gauss :: [Int] -> [Int]
gauss xs = map gSum (assertF msg allPositive xs)
  where
    msg = "Funktion gauss: Summenbildung " ++
          "fuer eine Zahl n <= 0"
    gSum :: Int -> Int
    gSum 1 = 1
    gSum n = n + gSum (n - 1)

allPositive :: (Num a, Ord a) => [a] -> Bool
allPositive [] = False
allPositive l@(x:xs) = all (0 <) l

```

Beispielauswertungen unter der Option `+A/-d`:

```

# _____
# || || || || || || || || ||
# ||_|| ||_|| ||_|| ||_|| ||_||
# ||---|| ||_|| ||_|| ||_||
# || || || || || || || || ||
# || || || Version: May 1999
#
# ...
#
# Example.lhs
# Type :? for help
# Example> gauss [1, 2, 3, 4]
# [1,3,6,10]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...

```

```

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org

```

```

#
# Result of the final check:
# All assertions are fulfilled
#
# Example> gauss [1, 2, 3, 0]
# [1,3,6,
# ERROR: Control stack overflow
#
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Funktion gauss: Summenbildung fuer eine Zahl n <= 0
# Example> gauss (take 3 [1, 2, 3, 0])
# [1,3,6]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> take 3 (gauss [1, 2, 3, 0])
# [1,3,6]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# 1 assertion is not verifiable.
#
# Example>

```

6. Beispiel:

Die nachstehend definierte Funktion `gauss'` erwartet genau wie die Funktion `gauss` als Argument eine Liste, die aus positiven, natürlichen Zahlen besteht und berechnet als Ergebnis für jede Zahl n aus dieser Liste den Wert $1 + 2 + \dots + n$. Mit Hilfe einer Zusicherung soll sichergestellt werden, daß alle Elemente der Argument-Liste größer als Null sind, um eine Nicht-Terminierung der Berechnung zu verhindern. Die Funktion `gauss'` unterscheidet sich von der Funktion `gauss` dadurch, daß die Funktion `assertF` nur partiell auf zwei Argumente angewendet wird.

```

gauss' :: [Int] -> [Int]
gauss' xs = map gSum (assert xs)
           where
             assert = assertF msg allPositive

```

Anhang B - Beispielauswertungen

```

msg = "Funktion gauss: Summenbildung " ++
      "fuer eine Zahl n <= 0"
gSum :: Int -> Int
gSum 1 = 1
gSum n = n + gSum (n - 1)

```

Beispielauswertungen unter der Option +A/-d:

```

# _____
# ||_||_||_||_||_||_||_||_||_||_
# ||____||_||_||_||_||_||_||_||_
# ||---||_||_||_||_||_||_||_||_
# ||_|_||_||_||_||_||_||_||_||_
# ||_|_||_||_||_||_||_||_||_||_
# ||_|_||_||_||_||_||_||_||_||_
# ||_|_||_||_||_||_||_||_||_||_
# ||_|_||_||_||_||_||_||_||_||_
# _____
#
# ...
#
# Example.lhs
# Type :? for help
# Example> gauss' [1, 2, 3, 4]
# [1,3,6,10]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> gauss' [1, 2, 3, 0]
# [1,3,6,
# ERROR: Control stack overflow
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** Funktion gauss: Summenbildung fuer eine Zahl n <= 0
# Example> gauss' (take 3 [1, 2, 3, 0])
# [1,3,6]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> take 3 (gauss' [1, 2, 3, 0])
# [1,3,6]
#
# *****
# *** Debug message ***
# *****

```



```

# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> fak (-4)
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Funktion fak ist nur fuer natuerliche Zahlen definiert
#
# Example>

```

8. Beispiel:

Die nachstehend definierte Funktion `fak'` berechnet für eine natürliche Zahl `n` die Fakultät, und zwar rekursiv. Mit Hilfe einer Zusicherung soll sichergestellt werden, daß die Zahl `n` nicht negativ ist, um eine Nicht-Terminierung der Berechnung zu verhindern.

```

fak' :: Int -> Int
fak' n = assertB msg (n >= 0) (fak'' n)
  where
    msg = "Funktion fak' ist nur fuer natuerliche " ++
          "Zahlen definiert"
    fak'' :: Int -> Int
    fak'' 0 = 1
    fak'' n = n * fak'' (n - 1)

```

Beispielauswertungen unter der Option `+A/-d`:

```

# -----
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||
# ||___||   ||___||   ||___||   ||___||   ||___||   ||___||   ||___||   ||___||   ||___||
# ||---||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||
# -----
#
# ...
#
# Example.lhs
# Type :? for help
# Example> fak' 4
# 24
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:

```

```

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org

```

```
# All assertions are fulfilled
#
# Example> fak' 0
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> fak' (-4)
#
# ERROR: Control stack overflow
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Funktion fak' ist nur fuer natuerliche Zahlen definiert
# Example>
```

9. Beispiel:

Die Korrektheit der Funktion `deleteControl` soll mit Hilfe einer Zusicherung überprüft werden. Diese Funktion entfernt aus einer Zeichenkette alle Steuerzeichen:

```
deleteControl :: String -> String
deleteControl [] = []
deleteControl (x:xs)
  | myIsControl x = deleteControl xs
  | otherwise    = x : deleteControl xs

myIsControl :: Char -> Bool
myIsControl c = c < ' '

-- !! Die Definition der Funktion myIsControl ist fehlerhaft !!
-- Ihre letzte Zeile muesste lauten:
-- myIsControl c = c < ' ' || c == '\DEL'

checkString :: String -> Bool
checkString [] = True
checkString (x:xs) = (ord x >= 32 && ord x /= 127) &&
  (checkString xs)

myPrint :: String -> IO ()
myPrint s = do putStr (assertF msg checkString (deleteControl s))
  where
    msg = "deleteControl entfernt nicht alle Steuerzeichen"
```

Der Fehler in der Definition der Funktion `myIsControl` bewirkt, daß das Steuerzeichen `'\DEL'` nicht gelöscht wird.

Beispielauswertungen unter der Option `+A/-d`:

```

#                               _____
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||_
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||_
# ||---||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||_
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||_
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||_
# ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||   ||_
#                               Version: May 1999
#                               _____
#
# ...
#
# Example.lhs
# Type :? for help
# Example> myPrint "Eine Zeichenkette ohne Steuerzeichen"
# Eine Zeichenkette ohne Steuerzeichen
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> myPrint "Eine \DEL Zeichenkette mit \DEL Steuerzeichen"
# Eine Zeichenkette mit Steuerzeichen
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** deleteControl entfernt nicht alle Steuerzeichen
#
# Example> myPrint (init "Eine Zeichenkette mit Steuerzeichen\DEL")
# Eine Zeichenkette mit Steuerzeichen
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example>

```

10. Beispiel:

Die Korrektheit der Funktion `capitalize` soll mit Hilfe einer Zusicherung überprüft werden. Diese Funktion wandelt alle in einer Zeichenkette enthaltenen Kleinbuchstaben in Großbuchstaben um:

```
capitalize :: String -> String
capitalize x = map myToUpper x

myToUpper :: Char -> Char
myToUpper x = chr (ord x + offset)

-- !! Die Definition der Funktion myToUpper ist fehlerhaft !!
-- Sie muesste lauten:
-- myToUpper x | isLower x = chr (ord x + offset)
--             | otherwise = x
-- Das entspraecher der Definition in der Datei Prelude.hs

offset :: Int
offset = ord 'A' - ord 'a'

checkConversion :: String -> String -> Bool
checkConversion [] [] = True
checkConversion (x:xs) (y:ys)
    | isLower x = (ord y == ord x + offset) && checkConversion xs ys
    | otherwise = (x == y)                  && checkConversion xs ys

myPrint' :: String -> IO ()
myPrint' s = do putStr (assertF msg
                       (checkConversion s)
                       (capitalize s))
               where
                 msg = "capitalize wandelt Zeichen fehlerhaft um"
```

Der Fehler in der Definition der Funktion `myToUpper` bewirkt, daß Zeichen, bei denen es sich nicht um Kleinbuchstaben handelt, ebenfalls umgewandelt werden.

Beispielauswertungen unter der Option `+A/-d:`

```
# _____
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ...
#
# Example.lhs
# Type :? for help
# Example> myPrint' "hugs"
# HUGS
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
```

```
#
# Example> myPrint' "hugsYX"
# HUGS98
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** capitalize wandelt Zeichen fehlerhaft um
#
# Example> myPrint' (take 4 "hugs98")
# HUGS
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example>
```

11. Beispiel:

Mit Hilfe der vordefinierten Funktion `inRange` soll eine Zusicherung formuliert werden, die sicherstellt, daß das Argument einer Funktion innerhalb eines zulässigen Wertebereichs liegt:

```
myAbs :: Int -> Int
myAbs x = assertB msg (inRange (min, max) x) (abs x)
  where
    msg = "Argument der Funktion myAbs ausserhalb " ++
          "des zulaessigen Wertebereichs"
    min = (-10)
    max = 10
```

Beispielauswertungen unter der Option `+A/-d`:

```
# _____
# ||_____| ||_____| ||_____| ||_____| ||_____|
# ||_____| ||_____| ||_____| ||_____| ||_____|
# ||_____| ||_____| ||_____| ||_____| ||_____|
# ||_____| ||_____| ||_____| ||_____| ||_____|
# ||_____| ||_____| ||_____| ||_____| ||_____|
# ||_____| ||_____| ||_____| ||_____| ||_____|
# ||_____| ||_____| ||_____| ||_____| ||_____|
# _____
#
# ...
#
# Example.lhs
# Type :? for help
# Example> myAbs (-11)
# 11
#
# *****
# *** Debug message ***
# *****
```

```
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
```

Anhang B - Beispielauswertungen

```
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Argument der Funktion myAbs ausserhalb des zulaessigen Wertebereichs
#
# Example> myAbs (-10)
# 10
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> myAbs 10
# 10
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> myAbs 11
# 11
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Argument der Funktion myAbs ausserhalb des zulaessigen Wertebereichs
#
# Example>
```

12. Beispiel:

Mit Hilfe der vordefinierten Funktion `inRange` soll eine Zusicherung formuliert werden, die sicherstellt, daß das Ergebnis einer Funktion innerhalb eines zulässigen Wertebereichs liegt:

```
mySum :: [Int] -> Int
mySum l = assertF msg (inRange (min, max)) (sum l)
  where
    msg = "Ergebnis der Funktion mySum ausserhalb " ++
```

```
"des zulaessigen Wertebereichs"
min = (-10)
max = 10
```

Beispielauswertungen unter der Option +A/-d:

```
# _____
# || _____ || || _____ || || _____ || || _____
# || _____ || || _____ || || _____ || || _____
# || - - - || || _____ || || _____ ||
# || _____ || || _____ || || _____ ||
# || _____ || || _____ || || _____ ||
# || _____ || || _____ || || _____ ||
# || _____ || || _____ || || _____ ||
# || _____ || || _____ || || _____ ||
# _____
# Hugs 98: Based on the Haskell 98 standard
# Copyright (c) 1994-1999
# World Wide Web: http://haskell.org/hugs
# Report bugs to: hugs-bugs@haskell.org
# _____
#
# ...
#
# Example.lhs
# Type :? for help
# Example> mySum [-1, -2, -3, -4, -5]
# -15
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Ergebnis der Funktion mySum ausserhalb des zulaessigen Wertebereichs
#
# Example> mySum [-1, -2, -3, -4]
# -10
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> mySum [1, 2, 3, 4]
# 10
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> mySum [1, 2, 3, 4, 5]
# 15
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
```

```

#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Ergebnis der Funktion mySum ausserhalb des zulaessigen Wertebereichs
#
# Example>

```

13. Beispiel:

Die Korrektheit der Funktion `mapBinTree` soll mit Hilfe einer Zusicherung überprüft werden. Analog zu der vordefinierten Funktion `map` wendet diese Funktion eine Funktion `f` auf alle Knoten eines Binärbaums an:

```

data BinTree a = Nil | Node a (BinTree a) (BinTree a)
                 deriving (Eq, Ord, Show)

build x = foldr' myInsert (reverse x) Nil
         where
         foldr' f [] z      = z
         foldr' f (x:xs) z = f x (foldr' f xs z)

myInsert x Nil = Node x Nil Nil
myInsert x (Node k l r)
 | x < k      = Node k (myInsert x l) r
 | otherwise = Node k l (myInsert x r)

mapBinTree f x
 = assertB msg (map f (preorder x) == preorder res) res
   where
   msg = "mapBinTree erstreckt sich nicht auf alle Knoten " ++
         "des Binaerbaums"
   res = mapBinTree' f x
   mapBinTree' f Nil
     = Nil
   mapBinTree' f (Node k l r)
     = Node (f k) (mapBinTree' f l) r

-- !! Die Definition der Funktion mapBinTree' ist fehlerhaft !!
-- Ihre letzte Zeile muesste lauten:
--   = Node (f k) (mapBinTree' f l) (mapBinTree' f r)

preorder Nil        = []
preorder (Node k l r) = [k] ++ preorder l ++ preorder r

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

```

Der Fehler in der Definition der Funktion `mapBinTree'` bewirkt, daß rekursiv immer nur der linke Nachfolger eines Knotens besucht wird.

Beispielauswertungen unter der Option `+A/-d`:

```

#
# |||| || |||| || || ||||
# ||____ || ||____ || ||____ ||____ ||
# ||--- || ||____ ||
# || ||
# || || Version: May 1999
#
# _____
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
#

```

Anhang B - Beispielauswertungen

```
# ...
#
# Example.lhs
# Type :? for help
# Example> mapBinTree square Nil
# Nil
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> mapBinTree square (build [2])
# Node 4 Nil Nil
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> mapBinTree square (build [2,1])
# Node 4 (Node 1 Nil Nil) Nil
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> mapBinTree square (build [1,2])
# Node 1 Nil (Node 2 Nil Nil)
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** mapBinTree erstreckt sich nicht auf alle Knoten des Binaerbaums
#
# Example>
```


14. Beispiel:

Dieses Beispiel veranschaulicht das Verhalten des modifizierten Haskell-Interpreters Hugs bei `let`-Ausdrücken und `where`-Ausdrücken:

Zu testende `let`-Ausdrücke:

```
let x = 1 in 2 + assertB "Fehler" (x==0) (x*x)
```

- Programmabbruch zu erwarten, da `x` im normalen Programmablauf ausgewertet werden muss.

```
let x = 1 in (\a -> 2 + assertB "Fehler" (x==0) (a*a)) 1
```

- Programmabbruch nicht zu erwarten, da `x` im normalen Programmablauf nicht ausgewertet werden muss.

```
let x = True in True || assertB "Fehler" (x==False) False
```

- Programmabbruch aufgrund der Definition der ODER-Verknüpfung nicht zu erwarten:

```
False || x = x
True  || x = True
```

```
let x = True in assertB "Fehler" (x==False) (True || False)
```

- Programmabbruch zu erwarten, da die Zusicherung im Gegensatz zu dem vorstehenden Ausdruck nicht mehr Bestandteil des zweiten Arguments der ODER-Verknüpfung ist, sondern diese umschließt.

```
let x = 1; y = 2 in fst (assertB "Fehler" (x==0 && y/=0) (x,y))
```

- Programmabbruch aufgrund der Definition der UND-Verknüpfung zu erwarten:

```
False && x = False
True  && x = x
```

Zu testende `where`-Ausdrücke:

```
2 + assertB "Fehler" (x==0) (x*x) where x = 1
```

```
(\a -> 2 + assertB "Fehler" (x==0) (a*a)) 1 where x = 1
```

```
True || assertB "Fehler" (x==False) False where x = True
```

```
assertB "Fehler" (x==False) (True || False) where x = True
```

```
fst (assertB "Fehler" (x==0 && y/=0) (x,y)) where x = 1; y = 2
```

Beispielauswertungen unter der Option `+A/-d`:

```
# _____
# ||      ||  ||      ||  ||      ||  ||_____
# ||_____||  ||_____||  ||_____||  _____||
# ||---||      _____||
# ||      ||
# ||      || Version: May 1999
#
# ...
#
# Example.lhs
# Type :? for help
# Example> let x = 1 in 2 + assertB "Fehler" (x==0) (x*x)
```

```
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
```

Anhang B - Beispielauswertungen

```
# 3
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Fehler
#
# Example> let x = 1 in (\a -> 2 + assertB "Fehler" (x==0) (a*a)) 1
# 3
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# 1 assertion is not verifiable.
#
# Example> let x = True in True || assertB "Fehler" (x==False) False
# True
# Example> let x = True in assertB "Fehler" (x==False) (True || False)
# True
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Fehler
#
# Example> let x = 1; y = 2 in fst (assertB "Fehler" (x==0 && y/=0) (x,y))
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Fehler
#
# Example> 2 + assertB "Fehler" (x==0) (x*x) where x = 1
# 3
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
```

```

# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Fehler
#
# Example> (\a -> 2 + assertB "Fehler" (x==0) (a*a)) 1 where x = 1
# 3
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# 1 assertion is not verifiable.
#
# Example> True || assertB "Fehler" (x==False) False where x = True
# True
# Example> assertB "Fehler" (x==False) (True || False) where x = True
# True
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Fehler
#
# Example> fst (assertB "Fehler" (x==0 && y/=0) (x,y)) where x = 1; y = 2
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Fehler
#
# Example>

```

15. Beispiel:

Die Korrektheit der Funktion `filterEvenNat` soll mit Hilfe einer Zusicherung überprüft werden. Diese Funktion filtert aus einer Liste vom Typ `[Int]` alle Elemente heraus, die positiv und geradzahlig sind:

```

filterEvenNat :: [Int] -> [Int]
filterEvenNat l = assertF msg
                  (\a -> null (filter ((>=) 0) a ++
                                filter (not . isEven) a))
                  (filterEvenNat' l)

```

```

where
  filterEvenNat' :: [Int] -> [Int]
  filterEvenNat' [] = []
  filterEvenNat' (x:xs)
    -- | x < 0      = filterEvenNat' xs
    | x == 0      = filterEvenNat' xs
    | x `rem` 2 /= 0 = filterEvenNat' xs
    | otherwise   = x : filterEvenNat' xs
  msg = "filterEvenNat fehlerhaft definiert"

```

-- !! Die Definition der Funktion filterEvenNat' ist fehlerhaft !!
 -- Das Auskommentieren ihrer vierten Zeile bewirkt, dass negative
 -- Zahlen, die ohne Rest durch zwei teilbar sind, ebenfalls
 -- herausgefiltert werden.

```

isEven :: Int -> Bool
isEven x = x `rem` 2 == 0

```

Beispielauswertungen unter der Option +A/-d:

```

# _____|_____|_____|_____|_____
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# ||_||_||_||_||_||_||_||_||_||_||_||_||_||_
# _____|_____|_____|_____|_____
#
# ...
#
# Example.lhs
# Type :? for help
# Example> filterEvenNat [1, 2, 3, 4]
# [2,4]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> filterEvenNat [1, 2, 3, -4]
# [2,-4]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** filterEvenNat fehlerhaft definiert
#
# Example> filterEvenNat (take 3 [1, 2, 3, -4])
# [2]
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.

```

```

# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> head (filterEvenNat [1, 2, 3, -4])
# 2
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# 1 assertion is not verifiable.
#
# Example>

```

16. Beispiel:

Dieses Beispiel soll zeigen, daß die Informationen über wiederherzustellende Zusicherungen ordnungsgemäß gespeichert werden.

```

multiple x y z = assertF "Zusicherung 1" (1 ==) x +
                 assertF "Zusicherung 2" (2 ==) y +
                 assertF "Zusicherung 3" (3 ==) z

```

Beispielauswertungen unter der Option +A/-d:

```

#
#  |__|  |__|  |__|  |__|  |__|
#  ||_||  ||_||  ||_||  ||_||  ||_||
#  ---|  ---|  ---|  ---|  ---|
#  ||  ||  ||  ||  ||  ||  ||
#  ||  ||  ||  ||  ||  ||  ||  Version: May 1999
#
# ...
#
# Example.lhs
# Type :? for help
# Example> multiple 1 2 3
# 6
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 3 assertions left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> multiple 1 2 0
# 3
#
# *****
# *** Debug message ***
# *****
#

```

```

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org

```

Anhang B - Beispielauswertungen

```
# The evaluation process is completed.
# There are 3 assertions left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** Zusicherung 3
#
# Example> multiple 1 0 3
# 4
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 3 assertions left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** Zusicherung 2
#
# Example> multiple 0 2 3
# 5
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 3 assertions left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** Zusicherung 1
#
# Example>
```

17. Beispiel:

Dieses Beispiel soll zeigen, daß eine Zusicherung auch nach einer Garbage Collection ordnungsgemäß wiederhergestellt werden kann.

```
multiple' x y z = assertB "Zusicherung 1" (x == 1) (
    assertB "Zusicherung 2" (y == 2) (
        assertB "Zusicherung 3" (z == 3) (
            sum (assertF "Zusicherung 4" allPositive [1..50000]) +
                x + y + z)))
```

Beispielauswertungen unter der Option +A/-d:

```
# _____
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||__||  ||__||  ||__||  ||__||  ||__||
# ||---||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
# _____
# ...
#
# Example.lhs
# Type :? for help
# Example> :s +s
# Example> sum [1..50000]
```

```
_____
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-1999
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
_____
```

Anhang B - Beispielauswertungen

```
# 1250025000
# (1050051 reductions, 1645227 cells, 7 garbage collections)
# Example> :s -s
# Example> multiple' 1 2 3
# 1250025006
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 4 assertions left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
#
# Example> multiple' 1 2 0
# 1250025003
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 4 assertions left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** Zusicherung 3
#
# Example> multiple' 1 0 3
# 1250025004
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 4 assertions left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** Zusicherung 2
#
# Example> multiple' 0 2 3
# 1250025005
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 4 assertions left to check.
#
# Please wait ...
#
# *** Program error: Assertion failed
# *** Zusicherung 1
#
# Example>
```

18. Beispiel:

Dieses Beispiel soll veranschaulichen, wie der modifizierte Haskell-Interpreter Hugs reagiert, wenn die eingebauten Fehlerrountinen aufgrund der Division durch die Zahl Null zu einem Programmabbruch führen. Insbesondere soll es auch zeigen, daß nach einem Programmabbruch sämtliche in der Check List verzeichneten Zusicherungen einer erneuten Überprüfung unterzogen werden.

Zu testende Ausdrücke:

- ```
1 / fromInt (assertB "Fehler" True 0)
```
- Zusicherung erfüllt, keine ergänzende Fehlermeldung zu erwarten.
- ```
1 / fromInt (assertB "Fehler" False 0)
```
- Zusicherung nicht erfüllt, ergänzende Fehlermeldung zu erwarten.
- ```
1 / assertB "Zsg. 1" True (fromInt (assertB "Zsg. 2" True 0))
```
- Zusicherungen erfüllt, keine ergänzende Fehlermeldung zu erwarten.
- ```
1 / assertB "Zsg. 1" False (fromInt (assertB "Zsg. 2" True 0))
```
- Zsg. 1 nicht erfüllt, ergänzende Fehlermeldung zu erwarten.
- ```
1 / assertB "Zsg. 1" True (fromInt (assertB "Zsg. 2" False 0))
```
- Zsg. 2 nicht erfüllt, ergänzende Fehlermeldung zu erwarten.
- ```
1 / assertB "Fehler" (1 == (2 / fromInt 0)) (fromInt 0)
```

Beispielauswertungen unter der Option +A/-d:

```
# _____
# || || || || || || || || || || ||
# ||_____| ||_____| ||_____| ||_____|
# ||----| || | || || || | ||
# || || || || || || || || || ||
# || || || || || || || || || ||
# || || || || || || || || || ||
# || || || || || || || || || ||
# _____
# ...
#
# Example.lhs
# Type :? for help
# Example> 1 / fromInt (assertB "Fehler" True 0)
#
# Program error: {primDivDouble 1.0 0.0}
#
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
# Example> 1 / fromInt (assertB "Fehler" False 0)
#
# Program error: {primDivDouble 1.0 0.0}
#
#
# *****
```


Anhang B - Beispielauswertungen

```
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Fehler
# Example> 1 / assertB "Zsg. 1" True (fromInt (assertB "Zsg. 2" True 0))
#
# Program error: {primDivDouble 1.0 0.0}
#
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 2 assertions left to check.
#
# Please wait ...
#
# Result of the final check:
# All assertions are fulfilled
# Example> 1 / assertB "Zsg. 1" False (fromInt (assertB "Zsg. 2" True 0))
#
# Program error: {primDivDouble 1.0 0.0}
#
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 2 assertions left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zsg. 1
# Example> 1 / assertB "Zsg. 1" True (fromInt (assertB "Zsg. 2" False 0))
#
# Program error: {primDivDouble 1.0 0.0}
#
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 2 assertions left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zsg. 2
# Example> 1 / assertB "Fehler" (1 == (2 / fromInt 0)) (fromInt 0)
#
# Program error: {primDivDouble 1.0 0.0}
#
#
# *****
# *** Debug message ***
# *****
#
```

```
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Program error: {primDivDouble 2.0 0.0}
# Example>
```

19.Beispiel:

Dieses Beispiel soll das Verhalten des modifizierten Haskell-Interpreters Hugs im Zusammenhang mit CAF-Definitionen veranschaulichen.

```
pair :: (Int, Int)
pair = assertF msg (not . isEqSnd) (1, 0)
      where
        msg = "Zweites Element der CAF pair ist 0"

isEqSnd :: (Int, Int) -> Bool
isEqSnd (_, 0) = True
isEqSnd (_, _) = False

incr :: Int -> Int
incr x = fst pair + x

numberOne :: Int
numberOne = fst pair
```

Zu testende Ausdrücke:

- 1) `1 + fst (assertF "Fehler" (not . isEqSnd) (1, 2-2))`
 - Zu erwartendes Ergebnis: Kein Programmabbruch durch die Zusicherung, da das zweite Element des Paares im normalen Programmablauf nicht ausgewertet wird.
- 2) `1 + fst pair`
 - Zu erwartendes Ergebnis: Kein Programmabbruch durch die Zusicherung, da sich dieser Ausdruck von dem ersten zu testenden Ausdruck alleine dadurch unterscheidet, daß das Argument von `fst` durch eine CAF ersetzt wurde.
- 3) `1 + assertB "Fehler" (snd pair == 0) (incr 1)`
 - Zu erwartendes Ergebnis: Kein Programmabbruch durch die Zusicherung, weil die CAF `pair` nach Bildung von `foo` als Programmausdruck qualifiziert wird.
- 4) `1 + assertB "Fehler" (snd pair == 0) numberOne`
 - vgl. Ausdruck 3
- 5) `1 + assertB "Fehler" (snd pair == 0) 4`
 - Zu erwartendes Ergebnis: Programmabbruch durch die in der CAF `pair` enthaltene Zusicherung.
- 6) `1 + assertB "Fehler" (snd pair == 0) numberOne + snd pair`
 - Zu erwartendes Ergebnis: Programmabbruch durch die in der CAF `pair` enthaltene Zusicherung.


```
# Example> 1 + assertB "Fehler" (snd pair == 0) 4
# 5
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zweites Element der CAF pair ist 0
#
# Example> 1 + assertB "Fehler" (snd pair == 0) numberOne + snd pair
# 2
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There are 2 assertions left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zweites Element der CAF pair ist 0
#
# Example>
```

20. Beispiel:

Dieses Beispiel soll anhand der Auswertung der Ausdrücke `fst pair` und `snd pair` das Zusammenspiel der Optionen A und d veranschaulichen.

Beispielauswertungen unter der Option +A/-d:

```
#
#  _  _  _  _  _  _
# ||  ||  ||  ||  ||  ||
# | |  | |  | |  | |  | |
# | |  | |  | |  | |  | |
# | |  | |  | |  | |  | |
# | |  | |  | |  | |  | |
# ||  ||  ||  ||  ||  ||
#
# Hugs 98: Based on the Haskell 98 standard
# Copyright (c) 1994-1999
# World Wide Web: http://haskell.org/hugs
# Report bugs to: hugs-bugs@haskell.org
#
#
# ...
#
# Example.lhs
# Type :? for help
# Example> fst pair
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# 1 assertion is not verifiable.
#
# Example> snd pair    !! kein Programmabbruch !!
# 0
```

Anhang B - Beispielauswertungen

Example>

Dieselben Beispielauswertungen in umgekehrter Reihenfolge (Option: +A/-d):

```
# _____
# ||  ||  ||  ||  ||  ||
# ||__||  ||__||  ||__||  ||__||
# ||--||      ||__||
# ||  ||      Version: May 1999
# _____
#
# ...
#
# Example.lhs
# Type :? for help
# Example> snd pair
# 0
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zweites Element der CAF pair ist 0
#
# Example> fst pair
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zweites Element der CAF pair ist 0
#
# Example>
```

Dieselben Beispielauswertungen unter der Option +d:

```
# _____
# ||  ||  ||  ||  ||  ||
# ||__||  ||__||  ||__||  ||__||
# ||--||      ||__||
# ||  ||      Version: May 1999
# _____
#
# ...
#
# Example.lhs
# Type :? for help
# Example> :s +d
# Example> fst pair
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
```


Anhang B - Beispielauswertungen

```
#
# *** Program error: Assertion failed
# *** Zweites Element der CAF pair ist 0
#
# Example> fst pair
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zweites Element der CAF pair ist 0
#
# Example> :s -A      !! Optionswechsel !!
# Example> :s +A
# Example> fst pair  !! Trotz Optionswechsel wird die CAF pair nicht !!
# 1                  !! geloescht. Grund: Unter der Option -A wurde  !!
#                    !! keine Auswertung vorgenommen.             !!
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
# *** Zweites Element der CAF pair ist 0
#
# Example> :s -A
# Example> 2+2
# 4
# Example> :s +A
# Example> fst pair
# 1
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
# Result of the final check:
# 1 assertion is not verifiable.
#
# Example> snd pair
# 0
# Example> :s +d
# Example> snd pair
# 0
#
# *****
# *** Debug message ***
# *****
#
# The evaluation process is completed.
# There is 1 assertion left to check.
#
# Please wait ...
#
#
# *** Program error: Assertion failed
```



```
# There is 1 assertion left to check.  
#  
# Please wait ...  
#  
# Result of the final check:  
# 1 assertion is not verifiable.  
#  
# Example>
```

Trotz Zusicherung wird in der zweiten Auswertung der Fehler nicht erkannt. Der Grund hierfür ist die UND-Verknüpfung:

```
False && x    = False  
True  && x    = x
```

Für die Ergebnisermittlung muß das zweite Argument der Funktion `xor` nicht ausgewertet werden, also wird die Überprüfung der Zusicherung nicht zu Ende geführt.

Literaturverzeichnis

- [Abelson 1991] Harold Abelson, Gerald Jay Sussman, Julie Sussman: *Struktur und Interpretation von Computerprogrammen*. Springer-Verlag, Berlin Heidelberg New York, 1991.
- [Augustsson 1984] Lennart Augustsson: *A compiler for Lazy ML*, in: Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin, S. 218-227, August 1984.
- [Backhouse 1989] Roland C. Backhouse: *Programmkonstruktion und Verifikation*. Carl Hanser Verlag, München Wien / Prentice Hall International, 1989.
- [Davie 1992] Antony J. T. Davie: *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, Cambridge, 1992.
- [Endres 1977] Albert Endres: *Analyse und Verifikation von Programmen*. Oldenbourg Verlag, München Wien, 1977.
- [Futschek 1989] Gerald Futschek: *Programmentwicklung und Verifikation*. Springer-Verlag, Wien New York, 1989.
- [Hauptmann 1992] Klaus Hauptmann: *Abnahmetest großer Systeme*, in: *Testen, Analysieren und Verifizieren von Software*, hrsg. von Peter Liggesmeyer, Harry M. Sneed, Andreas Spillner. Springer-Verlag, Berlin Heidelberg New York, 1992, S. 102 - 110.
- [Hinze 1992] Ralf Hinze: *Einführung in die funktionale Programmierung mit Miranda*. Teubner, Stuttgart, 1992.
- [Hohlfeld 1992] Bernhard Hohlfeld, Werner Struckmann: *Einführung in die Programmverifikation*. BI-Wissenschaftsverlag, Mannheim Leipzig Wien Zürich, 1992.
- [Hugs-Prelude] Modul `Prelude.hs` des Haskell-Interpreters Hugs, Version: Mai 1999.
- [Johnsson 1984] T. Johnsson: *Efficient compilation of lazy evaluation*, in: Proceedings of the ACM Conference on Compiler Construction, Montreal, S. 58-69, Juni 1984.
- [Jones 1994] Mark P. Jones: *The implementation of the Gofer functional programming system*. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994 - Der Bericht ist erhältlich über:
<http://www.cse.ogi.edu/~mpj/pubs.html>

- [Kernighan 1990] Brian W. Kernighan, Dennis M. Ritchie: *Programmieren in C*. Carl Hanser Verlag, München Wien / Prentice Hall International, 2. Aufl., 1990.
- [Liggesmeyer 1992] Peter Liggesmeyer: *Testen, Analysieren und Verifizieren von Software - eine klassifizierende Übersicht der Verfahren*, in: *Testen, Analysieren und Verifizieren von Software*, hrsg. von Peter Liggesmeyer, Harry M. Sneed, Andreas Spillner. Springer-Verlag, Berlin Heidelberg New York, 1992, S. 1 - 25.
- [Meyer 1992] Bertrand Meyer: *EIFFEL: THE LANGUAGE*. Prentice Hall International, 1992.
- [Müllerburg 1992] Monika Müllerburg: *Zur Formalisierung von Testkonzepten*, in: *Testen, Analysieren und Verifizieren von Software*, hrsg. von Peter Liggesmeyer, Harry M. Sneed, Andreas Spillner. Springer-Verlag, Berlin Heidelberg New York, 1992, S. 35 - 44.
- [Myers 1995] Glenford J. Myers: *Methodisches Testen von Programmen*. Oldenbourg Verlag, München Wien, 5. Aufl., 1995.
- [Peyton Jones 1987] Simon L. Peyton Jones: *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.
- [Peyton Jones 1992] Simon L. Peyton Jones, David R. Lester: *Implementing Functional Languages*. Prentice Hall International, 1992.
- [Peyton Jones 1999] Simon L. Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler: *The Haskell 98 Report - Haskell 98: A Non-strict, Purely Functional Language*. Februar 1999.
- [Prelude] In [Peyton Jones 1999] spezifizierte Programmbibliothek.
- [Spillner 1992] Andreas Spillner: *Testmethoden und Testdatengewinnung für den Integrationstest modularer Softwaresysteme*, in: *Testen, Analysieren und Verifizieren von Software*, hrsg. von Peter Liggesmeyer, Harry M. Sneed, Andreas Spillner. Springer-Verlag, Berlin Heidelberg New York, 1992, S. 91 - 101.
- [Thompson 1999] Simon Thompson: *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2. Aufl., 1999.