Universität Passau
Fakultät für Mathematik und Informatik

Diplomarbeit

Automatische Verifikation von Gleichheitsbeweisen in Haskell

Robert Günz

14. August 1998

Aufgabenstellung:
Prof. Christian Lengauer, Ph.D.
Lehrstuhl für Programmierung

Betreuer:
Christoph A. Herrmann

Danksagung

Ich möchte allen herzlich danken, die mir direkt oder indirekt bei der Erstellung dieser Arbeit beigestanden haben. Ganz besonders danke ich Christoph Herrmann für seine fachliche Betreuung und tatkräftige Unterstützung. Ferner möchte ich mich bei Herrn Prof. Lengauer, Ph.D., für seine Anmerkungen und Ratschläge bedanken.

Eidesstattliche Erklärung

Hiermit erkläre ich, Robert Günz, an Eides Statt, daß ich diese Diplomarbeit selbständig und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Diese Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, den 14. August 1998

Robert Günz

Zusammenfassung

Bei der Verwendung von funktionalen Sprachen lassen sich Programmtransformationen bzw. die Umformungen von einzelnen Ausdrücken mit Hilfe von Gleichheitsbeweisen (Equational Reasoning) auf ihre Korrektheit überprüfen, indem bei jedem Transformationsschritt ein Teilausdruck durch einen semantisch äquivalenten Ausdruck ersetzt wird. Obwohl diese Technik grundsätzlich auch für imperative Sprachen eingesetzt werden kann, beschränkt dort das (mögliche) Vorhandensein von Seiteneffekten einen umfassenden Einsatz. Aus diesem Grund verbindet man Equational Reasoning hauptsächlich mit funktionalen und insbesondere rein funktionalen Sprachen. Die Programmiersprache Haskell ist ein bedeutender Vertreter dieser Sprachklasse, weshalb Gleichheitsbeweise in dieser Sprache behandelt werden.

Ziel dieser Arbeit ist es, ein System zu entwickeln, das für die automatische Verifikation von Transformationsschritten eingesetzt werden kann, um deren formale Korrektheit sicherzustellen. Damit dem praktischen Einsatz nichts im Wege steht, wird ein Großteil der Haskell-Syntax unterstützt, so daß nur in seltenen Fällen Anpassungen an den zu vergleichenden Haskell-Ausdrücken vorzunehmen sind. Zur komfortableren Beweisführung erlaubt das System die Auswertung von Ausdrücken gemäß der Haskell-Semantik. Die während der Verifikation verwendeten Regeln (Gleichungen) können mit einer Prämisse versehen werden, die bei der Regelanwendung vom System berücksichtigt wird. Um zusätzliche Sicherheit zu gewährleisten, wird die Typkonsistenz einer Lösung mit Hilfe des Haskell-Interpreters hugs überprüft.

Nach erfolgreicher Verifikation besteht die Möglichkeit, einen annotierten Beweis auszugeben, in dem die verwendeten Regeln und die notwendigen Voraussetzungen aufgelistet werden.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Verwendung von Gleichheitsbeweisen

Die Gleichheit zweier Ausdrücke kann man dadurch zeigen, daß man den einen Ausdruck in den anderen umformt, indem man äquivalenz-erhaltende Transformationen verwendet. In jedem Schritt wird ein (Teil-)Ausdruck durch einen äquivalenten ersetzt. Im Englischen findet man hierfür oft die Bezeichnung „substituting equals for equals". Ein Gleichheitsbeweis (*Equational Reasoning*) ist in diesem Sinne eine Folge von Ausdrücken, die durch Verwendung geeigneter Regeln ineinander umgeformt werden können. *Equational Reasoning* (ER) bietet sich vor allem in rein funktionalen Sprachen an, da man hier Programme als eine Menge von Gleichungen auffassen kann und die Semantik dieser Sprachen gewährleistet, daß man die Seiten dieser Gleichungen innerhalb eines Ausdrucks gegeneinander austauschen kann.

1.2 Funktionale Sprachen

Neben der weit verbreiteten *imperativen* Programmierung, wie sie in *C* [22] oder *C++* [23] in Erscheinung tritt, gibt es auch noch andere Programmierstile, wie etwa die *funktionale* Programmierung [2, 39]. Diese basiert auf der Theorie der Funktionen. Im wesentlichen verbirgt sich dahinter nur eine Verallgemeinerung der Konzepte, die bereits aus der Schulmathematik bekannt sind. Ein funktionales Programm besteht aus einer Menge von Funktionsdefinitionen der Form $\texttt{f x}_1 \ldots \texttt{x}_n \texttt{ = exp}$ $(n \geq 0)$, die als Gleichungen angesehen werden können. Bekannte Vertreter dieser Richtung sind *Lisp* [19, 21], *Standard ML (SML)* [27, 41] und *Haskell* [36]. Nun sind aber nicht alle funktionalen Sprachen in ihrem Aufbau gleich. Es lassen sich zwei Klassen unterscheiden:

- rein funktionale Sprachen (*pure functional languages*)
- nicht rein funktionale Sprachen

Nicht alle funktionalen Sprachen halten sich vollständig an das funktionale Konzept, sondern man findet in ihnen manchmal auch Konstrukte der imperativen Welt, wie z.B. in Standard ML [29]. Haskell hingegen zählt zu den rein funktionalen Sprachen, und nur für diesen Sprachtyp ist ER ohne Einschränkung als Beweismethode anwendbar. Hier können nämlich Funktionsdeklarationen als Gleichungen verwendet werden.

1.3 Aufgabe des Systems

Falls die Gleichheit zweier Ausdrücke durch eine Folge von äquivalenz-erhaltenden Transformationen nachgewiesen wird, enthält der sich daraus ergebende Gleichheitsbeweis in der Regel viele Zwischenschritte, da ja jede Transformation durch die Anwendung genau einer Gleichung erreicht wird. Daher kann man leicht die Übersicht verlieren, wenn man solche Beweise per Hand führt. Es können sich Fehler einschleichen, u.a. Schreibfehler, die die Semantik des Programms ändern oder gar ein inkorrektes Programm verursachen können. Aus diesem Grund ist eine unabhängige Überprüfung der Transformationsschritte sinnvoll. Das System, das in dieser Arbeit entwickelt wurde, bietet die Möglichkeit, die Korrektheit der einzelnen Schritte zu zeigen. Dabei wird die rein funktionale Sprache *Haskell* zur Notation der zu beweisenden Ausdrücke verwendet. Da es sich um ein automatisches Verfahren handelt, kann auf diese Weise ohne großen Arbeitsaufwand die Gültigkeit eines Gleichheitsbeweises in Haskell nachgewiesen werden.

Um die Anzahl der Beweisschritte einzugrenzen (etwa bei der Auswertung von Teilausdrücken durch das wiederholte Einsetzen der Funktionsvorschrift), unterstützt das System die Auswertung von allgemeinen Haskell-Ausdrücken.

Das System kann außerdem bei der Entwicklung von ER-Beweisen helfen, da es neben der bloßen Überprüfung, ob der Beweis korrekt ist oder nicht, im Fehlerfall diejenigen Ausdrücke liefert, bei denen keine passende Gleichung gefunden werden konnte. Der Benutzer muß zu Beginn lediglich eine Menge von Gleichungen vorgeben, aus denen das System während der Verifikation wählen darf.

Wird eine Beweisverifikation erfolgreich durchgeführt, können abschließend die Transformationsschritte zusammen mit den resultierenden Bindungen und den Namen der verwendeten Gleichungen ausgegeben werden. Der Anwender erhält auf diese Weise zusätzlich einen annotierten Beweis als Ergebnis.

Ausgehend von einer Lösung, die das System geliefert hat, kann der Benutzer auf Wunsch mit Hilfe des eingebauten *Backtracking*-Mechanismus interaktiv nach weiteren Lösungen suchen, die sich hinsichtlich der verwendeten Gleichungen und der Anzahl der gebundenen Variablen unterscheiden.

1.4 Übersicht

- **Kapitel 2** beschäftigt sich hauptsächlich mit der rein funktionalen Sprache Haskell. Sie dient einerseits als Eingabesprache für das System, andererseits ist das System selbst in Haskell implementiert. Der Anfang dieses Kapitels widmet sich der Definition elementarer Begriffe und beschreibt kurz den λ-Kalkül [3, 12], der die formale Grundlage für die funktionale Programmierung liefert. Im Anschluß werden grundlegende Eigenschaften Haskells aufgeführt und die wichtigsten Sprachkonstrukte vorgestellt. Dies ist vor allem für diejenigen Leser gedacht, die mit Haskell noch wenig Erfahrung haben.

- In **Kapitel 3** wird die prinzipielle Methode des *Equational Reasoning* anhand eines kleinen Beispiels illustriert. Später wird kurz auf die Kombination von *Equational Reasoning* und Induktionsbeweisen eingegangen, um aufwendigere Probleme zu lösen. Danach wird die Aufgabenstellung für das System festgelegt. Um die Vielzahl der Haskell-Konstrukte auf einen möglichst kleinen Kern elementarer Funktionen zurückzuführen, werden dafür am Ende noch einige Umformungsregeln angegeben.

- **Kapitel 4** erläutert die Behandlung von `let`-Ausdrücken und die Auswertung von Haskell-Ausdrücken, wie sie vom System vorgenommen werden. Der Rest des Kapitels beschreibt die einzelnen Phasen, die während der Prüfung eines Beweises durchlaufen werden. Dazu werden eine Reihe von Transformationsregeln angegeben. Ferner wird die Einbettung des *Backtracking*-Mechanismus in dieses Schema gezeigt, das eine gezielte Rücknahme von Variablen-Bindungen und einschränkenden Bedingungen erlaubt.

- In **Kapitel 5** werden die wichtigsten Datentypen vorgestellt, die in der Implementierung verwendet werden. Daran anschließend folgt eine Auflistung der zentralen Funktionen, die die Umsetzung der in **Kapitel 4** gezeigten Transformationsregeln übernehmen. Der Beginn des Kapitels beschäftigt sich mit dem Parser der Eingabesprache und zwar insbesondere mit der Behandlung des sog. *Layout Styles* in Haskell.

- **Kapitel 6** zeigt, wie man das System benutzt. Es beschreibt die Bedeutung und das Format der Dateien, in denen die Information der Beweise abzulegen ist, die verifiziert werden sollen, damit das System auf sie zugreifen kann. Das Vorgehen wird dabei an einem kleinen Beispiel illustriert, in dem alle Schritte – beginnend von der Problemstellung bis hin zum *Backtracking* und der Ausgabe – durchlaufen werden.

- In **Kapitel 7** wird nochmals auf die Bedeutung von *Equational Reasoning* im Zuammenhang mit funktionalen Sprachen eingegangen. Darauf folgt eine kurze Bewertung des Systems, inwieweit es den Anforderungen für die Verifikation eines *Equational Reasoning* Beweises gerecht werden kann.

Kapitel 2

Übersicht zu Haskell

Im ersten Teil dieses Kapitels werden einige grundlegende Begriffe eingeführt. Es wird kurz der λ-Kalkül [6] beschrieben, der die Basis der funktionalen Programmierung bildet. Dabei wird auf die wichtigsten Reduktionsstrategien eingegangen, mit denen Ausdrücke des λ-Kalküls normalisiert werden können. Der zweite Teil des Kapitels beschäftigt sich mit Haskell [4]. Diese funktionale Programmiersprache ist für diese Arbeit in zweifacher Hinsicht von großer Bedeutung. Einerseits dient sie als Eingabesprache für die Beweise, die durch *Equational Reasoning* auf Korrektheit überprüft werden sollen, andererseits wurde das System selbst in Haskell realisiert. Neben den grundlegenden Eigenschaften dieser Sprache werden die wichtigsten Sprachkonstrukte beschrieben. Es wird gezeigt, in welcher Weise Funktions- und Wertdeklarationen Gleichungen darstellen, die bei Gleichheitsbeweisen angewendet werden dürfen. Den Abschluß bildet ein kurzer Blick auf das Typsystem von Haskell, wobei auch auf die Behandlung von überladenen Funktionen mit Hilfe des sog. Klassenkonzeptes eingegangen wird.

2.1 Formale Grundlagen

In den folgenden Abschnitten werden kurz einige elementare Begriffe eingeführt, auf die im weiteren Verlauf immer wieder Bezug genommen wird. Als Grundlage für diesen Überblick dienen [16, 34, 37].

2.1.1 Terme

Betrachten wir eine Signatur Σ, die eine Menge von Funktionssymbolen enthält. Dabei werden nullstellige Funktionen auch als Konstanten bezeichnet. Die Menge der Terme über der Signatur Σ und der Variablenmenge V, i.Z. $T_\Sigma(V)$, läßt sich induktiv wie folgt definieren:

- wenn f aus Σ und eine Konstante ist, dann ist f in $T_\Sigma(V)$
- wenn v eine Variable aus V ist, dann ist v in $T_\Sigma(V)$
- wenn f ein n-stelliges Funktionssymbol aus Σ ist und $t_1, \ldots, t_n$ Terme aus $T_\Sigma(V)$ sind, dann ist auch $f(t_1, \ldots, t_n)$ in $T_\Sigma(V)$

2.1.2 Substitution

Eine Abbildung $\sigma : V \rightarrow T_\Sigma(V)$ bezeichnet man als *Substitution*, falls die Menge $dom(\sigma) = \{v \mid v \in V \wedge \sigma(v) \neq v\}$ endlich ist.
Es wird folgende Schreibweise verwendet:

- anstelle von $\sigma(v)$ wird $v\sigma$ geschrieben
- eine Substitution σ mit $dom(\sigma) = \{v_1, \ldots, v_n\}$ und $v_i\sigma = t_i$ für $1 \leq i \leq n$ wird auch als $[t_1/v_1, \ldots, t_n/v_n]$ dargestellt
- [] bezeichnet die identische Abbildung

Man bezeichnet eine Substitution σ als eine *Umbenennung*, wenn σ eine Permutation auf der Variablenmenge V durchführt, d.h. σ injektiv und $\sigma(V) = V$ ist.

Durch die folgende Definition kann eine Substitution $\sigma : V \rightarrow T_\Sigma(V)$ in kanonischer Weise zu $\sigma^* : T_\Sigma(V) \rightarrow T_\Sigma(V)$ erweitert werden:

- $v\sigma^* = v\sigma$ für $v \in V$
- $(f(t_1, \ldots, t_n))\sigma^* = f(t_1\sigma^*, \ldots, t_n\sigma^*)$ für $f \in \Sigma$ und $t_1, \ldots, t_n \in T_\Sigma(V)$

Im weiteren Verlauf wird zwischen σ und σ^* nicht mehr unterschieden und für beide die Schreibweise σ verwendet.
$t\sigma$ wird als *Instanz* des Termes t bezeichnet.

Die Komposition zweier Substitutionen σ und τ ist definiert durch $t(\sigma\tau) = (\tau \circ \sigma)(t) = \tau(\sigma(t))$

Eine Substitution σ heißt *allgemeiner* als τ, i.Z. $\sigma \leq \tau$, falls eine Substitution ρ existiert, für die $\sigma\rho = \tau$ gilt.

Für Substitutionen lassen sich elementare Eigenschaften ableiten, von denen hier eine kleine Auswahl aufgeführt werden. So gilt für beliebige Substitutionen $\tau, \sigma, \sigma_1, \sigma_2$ und σ_3 und beliebige Terme t nämlich folgendes [16, 1.1.3]:

- $\sigma[] = \sigma = []\sigma$ (Komposition mit [])
- $\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3$ (Komposition ist assoziativ)
- $(\forall t : t \in T_\Sigma(V)\colon t\sigma = t\tau) \Rightarrow \sigma = \tau$

2.1.3 Unifikation

Zwei Terme t_1 und t_2 heißen *unifizierbar*, falls es eine Substitution σ gibt, für welche $t_1\sigma = t_2\sigma$ gilt. Nach Anwendung der Substitution stimmen also beide Terme syntaktisch überein.

In analoger Weise ist eine Menge M von Termen *unifizierbar*, falls eine Substitution σ existiert, so daß $t_1\sigma = t_2\sigma$ für alle t_1 und t_2 aus M gilt.

Die Substitution σ wird in diesen Fällen als *Unifikator* bezeichnet.

Ein Unifikator σ von t_1 und t_2 (bzw. M) heißt *allgemeinster Unifikator*, falls für alle weiteren Unifikatoren τ von t_1 und t_2 (bzw. M) σ allgemeiner als τ ist ($\sigma \leq \tau$).

Existenz und Eindeutigkeit allgemeinster Unifikatoren

Sind die Terme t_1 und t_2 *unifizierbar*, dann gibt es einen *allgemeinsten Unifikator* von t_1 und t_2. Sind σ und τ *allgemeinste Unifikatoren* von t_1 und t_2, dann sind sie bis auf Variablenumbenennung gleich. Es existiert also eine Umbenennung ρ mit der Eigenschaft $\sigma\rho = \tau$ bzw. $\sigma = \tau\rho^{-1}$.
Dieselben Aussagen gelten für *allgemeinste Unifikatoren* von Mengen von Termen.

Ein Beweis für diese Aussagen findet sich z.B. in [16, 2.1].

Martelli-Montanari-Kalkül

Eine Möglichkeit, einen allgemeinsten Unifikator zu bestimmen, ist die Verwendung der *Martelli-Montanari*-Regeln [24, 16]. Aufbauend auf diesen Regeln läßt sich ein Unifikationsalgorithmus formulieren.

Die Regeln haben folgende Gestalt, wobei $\uplus$ die disjunkte Vereinigung von Mengen bezeichnen soll:

Dekomposition

$$\frac{E \ \uplus \ \{f(t_1,\ldots,t_n) \equiv f(r_1,\ldots,r_n)\}}{E \ \cup \ \{t_1 \equiv r_1,\ldots,t_n \equiv r_n\}} \quad , n \geq 0$$

Variablen-Elimination

$$\frac{E \ \uplus \ \{x \equiv t\}}{E[t/x] \ \cup \ \{x \equiv t\}} \quad , \text{falls } x \text{ nicht in } t \text{ vorkommt } (\textit{occur check}), \text{ aber in } E$$

Umordnung

$$\frac{E \ \uplus \ \{t \equiv x\}}{E \ \cup \ \{x \equiv t\}} \quad , \text{falls } t \text{ keine Variable ist}$$

Elimination trivialer Gleichungen

$$\frac{E \ \uplus \ \{x \equiv x\}}{E}$$

Unifikationsalgorithmus

Eine Gleichungsmenge E von Termen heißt *vollständig gelöst*, falls sie die Form

$$E = \{x_1 \equiv t_1, \ldots, x_n \equiv t_n\} \ (n \geq 0)$$

hat, bei der die x_i paarweise verschiedene Variablen sind und kein x_i in einem der Terme t_j vorkommt.

Zur Unifikation von t_1 und t_2 beginnt man mit der Gleichungsmenge $E = \{t_1 \equiv t_2\}$ und wendet darauf die *Martelli-Montanari*-Regeln solange an, bis keine mehr paßt. Ist die verbleibende Gleichungsmenge E' *vollständig gelöst*, dann waren die beiden Terme t_1 und t_2 unifizierbar, und man kann an den Gleichungen der Form $v \equiv t$ die Substitution bzw. den allgemeinsten Unifikator ablesen. Im anderen Fall waren die zwei Terme nicht unifizierbar.

Will man den allgemeinsten Unifikator für eine endliche Menge M von Termen bestimmen, die die Form $M = \{t_1, \ldots, t_n\}$ $(n \geq 3)$ hat, so beginnt man die Ableitung mit der Gleichungsmenge $E = \{t_1 \equiv t_2, t_2 \equiv t_3, \ldots, t_{n-1} \equiv t_n\}$.

2.2 Der λ-Kalkül

Die Entwicklung des λ-Kalküls geht auf Alonzo Church in den 30er Jahren zurück [6]. Er diente ursprünglich dem Studium der Eigenschaften von Sprachen der math. Logik. Heute wird er verbreitet in der theoretischen Informatik eingesetzt, er findet aber auch in der Praxis Verwendung, wie z.B. im Hardware-Entwurf nicht-konventioneller Computer-Architekturen [34].

Den λ-Kalkül kann man einerseits als funktionale Programmiersprache, andererseits als *Theorie von Funktionen* auffassen. So ist es mittels verschiedener Reduktionsstrategien möglich, Ausdrücke des Kalküls in Normalformen zu überführen und ihnen damit eine Bedeutung zu verleihen. In dieser Sichtweise steht also der operationale Aspekt im Vordergrund wie in herkömmlichen funktionalen Sprachen auch, die häufig auf dem λ-Kalkül aufbauen. Durch Axiomatisierung der Gleichheit, Definition und Applikation von Funktionen gelangt man von dieser operationellen Sicht schließlich zur oben angesprochenen *Theorie von Funktionen*, in der auf formaler Ebene Eigenschaften von Ausdrücken nachgewiesen werden können.

Die folgenden Betrachtungen beziehen sich auf den *ungetypten* λ-Kalkül, um die Ausführungen nicht unnötig zu komplizieren. Eine Beschreibung der *getypten* Variante findet sich etwa in [34].

Als erstes wird die Grammatik beschrieben, durch die das Aussehen von λ-Ausdrücken festgelegt wird.

<exp>	:=	v	(Variable)
	|	c	(Konstante)
	|	(<exp> <exp>)	(Applikation)
	|	(λx.<exp>)	(λ-Abstraktion)

Die mit dieser Grammatik beschriebene Sprache besticht durch ihre Einfachheit. Ein λ-Ausdruck ist entweder eine Variable, eine Konstante, eine Applikation von λ-Ausdrücken oder eine λ-Abstraktion. Trotz dieser Schlichtheit ist der λ-Kalkül *Turing-mächtig*. Es lassen sich damit also alle berechenbaren Funktionen darstellen. Wie sich in diesem Kalkül rekursive Funktionen definieren lassen, wird kurz in Abschnitt 2.2.4 beschrieben, nachdem zuvor einige Ableitungsregeln angegeben wurden.

Freie und gebundene Variablen

Die Beschreibung des λ-Kalküls beschränkt sich hier auf die Darstellung von verschiedenen Reduktionsstrategien und auf die Angabe der dazugehörigen Normalformen.

Zunächst werden die Begriffe *frei* und *gebunden* für Variablen eingeführt, da diese Klassifikation für die Definition der Substitution auf λ-Ausdrücken benötigt wird.

Dazu bezeichne $\equiv_s$ die syntaktische Gleichheit von Ausdrücken, Kleinbuchstaben eine Variable und Großbuchstaben jeweils einen Ausdruck.

Variable x ist *frei* in:

- T, falls $x \equiv_s T$
- $(U\,V)$, falls x ist *frei* in U und/oder V
- $(\lambda y.T)$, falls $x \not\equiv_s y$ und x *frei* in T

Variable x ist *gebunden* in:

- $(U\,V)$, falls x ist *gebunden* in U und/oder V
- $(\lambda y.T)$, falls $x \equiv_s y$ oder x *gebunden* in T

Die Substitution auf λ-Ausdrücken wird dann folgendermaßen definiert:

- $y[T/x] \equiv_s T$, falls $x \equiv_s y$ ist
- $U[T/x] \equiv_s U$,
  falls U eine Konstante ist oder eine Variable, für die $x \not\equiv_s U$ gilt
- $(U\,V)[T/x] \equiv_s (U[T/x]\,V[T/x])$
- $(\lambda y.U)[T/x] \equiv_s (\lambda y.U)$, falls $x \equiv_s y$ oder x nicht *frei* in U ist
- $(\lambda y.U)[T/x] \equiv_s (\lambda y.U[T/x])$, falls $x \not\equiv_s y$ ist und y nicht *frei* in T ist
- $(\lambda y.U)[T/x] \equiv_s (\lambda z.U[z/y][T/x])$,
  falls $x \not\equiv_s y$, y *frei* in T und x *frei* in U ist und
  z weder *frei* in U, noch *frei* in T ist

2.2.1 Vereinfachungsregeln

Wir sind jetzt in der Lage, einige (Vereinfachungs-)Regeln zur Transformation von λ-Ausdrücken anzugeben, welche sowohl von links nach rechts als auch von rechts nach links angewandt werden dürfen.

α-Konversion (Renaming)

Falls die Variable y nicht *frei* in T ist, dann gilt:
$(\lambda x.T) \longleftrightarrow (\lambda y.T[y/x])$

β-Konversion (Substitution)

$((\lambda x.U)\,T) \longleftrightarrow U[T/x]$

δ-Reduktion

Diese Regel dient zur Auswertung vordefinierter Operationen, wie z.B.
$((+3)7) \longrightarrow 10$
$((*4)5) \longrightarrow 20$
Obwohl diese Regel grundsätzlich auch von rechts nach links angewendet werden könnte, wird diese Richtung i.allg. nicht verwendet.

η-Konversion (Redundanz-Elimination)

Falls die Variable x nicht *frei* in T ist, dann gilt:
$(\lambda x.(T\,x)) \longleftrightarrow T$

2.2.2 Normalform

Redex

Ein Teilausdruck in einem λ-Ausdruck heißt *Redex* (*red*ucible *ex*pression), wenn sich auf ihn eine β, δ, oder η-Konversion von links nach rechts anwenden läßt. Die Anwendung der Regel wird als *Reduktion* bezeichnet.

$(\lambda x.\underline{((\lambda y.y)\,T)}) \xrightarrow{\beta} (\lambda x.T)$
$\underline{((+2)3)} \xrightarrow{\delta} 5$
$(\lambda x.x)\,\underline{((\lambda y.y)\,T)} \xrightarrow{\beta} (\lambda x.x)\,T$

In den oben angegebenen Ausdrücken wurden die Redexe jeweils durch Unterstreichen gekennzeichnet und die entsprechende Reduktion unter Angabe ihres Namens durchgeführt. Es ist zu beachten, daß ein Ausdruck durchaus mehrere Redexe enthalten kann. So ist für den dritten Ausdruck von vorher auch die folgende Reduktion möglich.

$\underline{(\lambda x.x)\,((\lambda y.y)\,T)} \xrightarrow{\beta} ((\lambda y.y)\,T)$

Normalform

Ein Ausdruck ist in Normalform, wenn er keine Redexe mehr enthält.

Schwache Normalform

Ein Ausdruck ist in schwacher Normalform, wenn er keine Redexe außerhalb von λ-Abstraktionen mehr enthält.

So ist $((\lambda x.((\lambda y.y)\,U))\,V)$ bereits in schwacher Normalform, falls V keinen Redex enthält, da sich $((\lambda y.y)\,U)$ innerhalb der λ-Abstraktion mit der Variablen x befindet. Die Normalform hat allerdings die Gestalt $U\,[V/x]$, falls sich in diesem Ausdruck keine weiteren Reduktionen ergeben.

2.2.3 Reduktionsstrategien

Nun stellt sich die Frage, wie man solche Normalformen bestimmt. Man erkennt nämlich sehr schnell, daß nicht alle Ausdrücke eine Normalform besitzen und daß sich die Wahl der Redexe darauf auswirkt, ob eine Folge von Reduktionen terminiert oder nicht.

Betrachten wir dazu folgende Ausdrücke:

(1) $((\lambda x.(x\,x))\,(\lambda x.(x\,x)))$
(2) $((\lambda x.(\lambda y.y))\,((\lambda x.(x\,x))\,(\lambda x.(x\,x))))$

Der Ausdruck (1) besitzt keine Normalform, da er nur einen β-Redex hat und man durch die entsprechende Reduktion den ursprünglichen Ausdruck wieder erhält. Man gelangt also nie zu einem Ausdruck ohne Redex. Bei Ausdruck (2) kann man eine unendliche Folge von Reduktionsschritten angeben, ohne auf eine Normalform zu stoßen, wenn man immer den Redex $((\lambda x.(x\,x))\,(\lambda x.(x\,x)))$ wählt. Wird allerdings der gesamte Ausdruck (2) als Redex genommen, kommt man in einem Schritt zu $(\lambda y.y)$ und damit zu einer Normalform.

Die oben angegebenen Vereinfachungsregeln erfüllen die *Church-Rosser*-Eigenschaft. Zu ihrer Definition verwenden wir $\rightarrow^*$, was die reflexiv, transitive Hülle der links-nach-rechts Anwendung der Konversionregeln bezeichnet. Hinter $\rightarrow^*$ verbirgt sich also eine beliebige, aber endliche Anwendungssequenz dieser Regeln.

***Church-Rosser*-Eigenschaft**

Falls sich $T \to^* U$ und $T \to^* V$ reduzieren läßt, dann existiert ein Ausdruck W, so daß sich $U \to^* W$ und $V \to^* W$ reduzieren lassen.

$$\begin{tikzcd} \mathrm{U} \arrow[dr, "*"] & \mathrm{T} \arrow[l, "*"'] \arrow[r, "*"] & \mathrm{V} \arrow[dl, "*"'] \\ & \mathrm{W} & \end{tikzcd}$$

Eine unmittelbare Konsequenz der *Church-Rosser*-Eigenschaft ist, daß eine abgeleitete Normalform bis auf α-Konversion eindeutig ist und daß die Wahl der Redexe unerheblich ist, sofern man letztendlich zu einer Normalform gelangt.

$$\underline{((\lambda x.(\lambda y.(y\,x)))\,((\lambda x.c)\,y))} \stackrel{\beta}{\to} (\lambda z.(z\,\underline{((\lambda x.c)\,y)})) \stackrel{\beta}{\to} (\lambda z.(z\,c))$$
$$((\lambda x.(\lambda y.(y\,x)))\,\underline{((\lambda x.c)\,y)}) \stackrel{\beta}{\to} \underline{((\lambda x.(\lambda y.(y\,x)))\,c)} \stackrel{\beta}{\to} (\lambda y.(y\,c)) \stackrel{\alpha}{\to} (\lambda z.(z\,c))$$

Der obige Ausdruck soll illustrieren, daß sich die auf verschiedenen Reduktionswegen abgeleiteten Normalformen bis auf α-Konversion gleichen. Bei der ersten Variante muß bei der Substitution innerhalb von $(\lambda y.(y\,x))$ eine Umbenennung von y vorgenommen werden, da y *frei* in $((\lambda x.c)y)$ vorkommt. Dies ist der Grund für den unterschiedlichen Variablennamen der λ-Abstraktion in den beiden Normalformen, der durch α-Konversion beseitigt werden kann.

***Normal-Order*-Reduktion (*leftmost-outermost*)**

Eine Folge von Reduktionsschritten wird als *Normal-Order*-Reduktion bezeichnet, wenn in jedem Schritt der Redex im Ausdruck reduziert wird, der am weitesten links beginnt. Die Normal-Order-Reduktion zeichnet sich gegenüber anderen Reduktionsstrategien dadurch aus, daß sie terminiert und eine Normalform liefert, sofern der Ausdruck überhaupt eine Normalform besitzt [3].

Wenn man sich bei der Reduktion jeweils mit der schwachen Normalform begnügt, entspricht dies dem Parameterübergabemechanismus *call-by-name*.

***Applicative-Order*-Reduktion (*leftmost-innermost*)**

Wenn in jedem Reduktionsschritt der Redex reduziert wird, der am weitesten links beginnt und zusätzlich in seinem Inneren keinen weiteren Redex enthält, dann spricht man von Applicative-Order-Reduktion.

Intuitiv werden bei dieser Strategie erst die Argumente einer Funktion reduziert und dann deren Ergebnisse im Funktionsrumpf eingesetzt. Diese Strategie muß nicht terminieren, obwohl eine Normalform existiert, wie man am Ausdruck

$$((\lambda x.(\lambda y.y))\,((\lambda x.(x\,x))\,(\lambda x.(x\,x))))$$

sehen kann.

Diese Reduktion entspricht dem Parameterübergabemechanismus *call-by-value*, sofern man sich wieder auf die schwache Normalform beschränkt. Die funktionale Sprache SML basiert auf dieser Reduktionsstrategie.

Zum Abschluß dieser kurzen Einführung in den λ-Kalkül werden noch zwei weitere bekannte Normalformen und Reduktionsstrategien beschrieben. Dazu werden Ausdrücke der Form $(\lambda x.(\lambda y.T))$ mit $(\lambda x\,y.T))$ abgekürzt und statt $(\ldots(T_1\,T_2)\ldots T_m)$ $(m \geq 2)$ nur $T_1\,T_2\ldots T_m$ geschrieben.

***Head*-Normalform und *Head*-Reduktion**

Ein Ausdruck ist in Head-Normalform (HNF), wenn er die Gestalt

$$\lambda x_1 \dots x_n \, y \, T_1 \dots T_m \ (n \geq 0, m \geq 0)$$

hat, wobei x_i und y Variablen und T_j Ausdrücke sind.

Ein Ausdruck besitzt eine HNF, wenn er auf einen Ausdruck in HNF reduziert werden kann.

Im Ausdruck $\lambda x_1 \dots x_n \,.\, (\lambda y.T)\, T_1 \dots T_m \ (n \geq 0, m \geq 0)$ bezeichnet $(\lambda y.T)\, T_1$ einen *Head-Redex* und man spricht von einer *Head-Reduktion*, wenn der Head-Redex reduziert wird.

Es gilt, daß ein Ausdruck genau dann in HNF ist, wenn er keinen Head-Redex mehr enthält.

***Weak-Head*-Normalform und *lazy* Reduktion**

Ein Ausdruck ist in Weak-Head-Normalform (WHNF), wenn er entweder die Gestalt $(\lambda x.T)$ oder $x\, T_1 \dots T_n \ (n \geq 0)$ hat, wobei x eine Variable ist und die T_i Ausdrücke repräsentieren.

Der Ausdruck $(\lambda y.T)\, T_1 \dots T_n \ (n \geq 1)$ enthält den *lazy Redex* $(\lambda y.T)\, T_1$. Wenn der lazy Redex reduziert wird, handelt es sich um eine *lazy Reduktion* [37].

Ein Ausdruck hat eine WHNF, wenn er durch eine Folge von lazy Reduktionen auf einen Ausdruck in WHNF reduziert werden kann.

Die lazy Reduktion entspricht der Philosophie von lazy Sprachen, wie Haskell oder Miranda™, da auf diese Weise die Argumente von Funktionen nicht vor deren Übergabe, sondern erst bei Bedarf innerhalb des Funktionsrumpfes ausgewertet werden.

2.2.4 Rekursion

Um im ungetypten λ-Kalkül rekursive Funktionen zu definieren, verwendet man den Fixpunktgenerator Y, der folgendermaßen definiert werden kann:

$$Y \ = \ (\lambda h.((\lambda x.(h\, x\, x))\ (\lambda x.(h\, x\, x))))$$

Damit kann man eine (rekursive) Funktion $\texttt{f=exp}_\texttt{f}$ durch $Y\,(\lambda\texttt{f.exp}_\texttt{f})$ definieren und mit der Eigenschaft des Fixpunktgenerators Y ergibt sich

$$Y\,(\lambda\texttt{f.exp}_\texttt{f}) \ \leftrightarrow^* \ (\lambda\texttt{f.exp}_\texttt{f})\ (Y\,(\lambda\texttt{f.exp}_\texttt{f}))$$

was man an folgenden Reduktionen ablesen kann:

$$\begin{aligned}
Y\,(\lambda\texttt{f.exp}_\texttt{f}) &\rightarrow (\lambda h.((\lambda x.(h\, x\, x))\ (\lambda x.(h\, x\, x))))\ (\lambda\texttt{f.exp}_\texttt{f}) \\
&\stackrel{\beta}{\rightarrow} (\lambda x.((\lambda\texttt{f.exp}_\texttt{f})\, x\, x))\ (\lambda x.((\lambda\texttt{f.exp}_\texttt{f})\, x\, x)) \\
&\stackrel{\beta}{\rightarrow} (\lambda\texttt{f.exp}_\texttt{f})\ ((\lambda x.((\lambda\texttt{f.exp}_\texttt{f})\, x\, x))\ (\lambda x.((\lambda\texttt{f.exp}_\texttt{f})\, x\, x))) \\
&\leftarrow (\lambda\texttt{f.exp}_\texttt{f})\ (Y\,(\lambda\texttt{f.exp}_\texttt{f}))
\end{aligned}$$

Die Verwendung von Y ist allerdings in dieser Form nur im ungetypten λ-Kalkül möglich, da sich im getypten Fall kein endlicher Typ für Y ableiten läßt. Dieses Problem kann man im getypten λ-Kalkül dadurch umgehen, daß der Operator Y speziell behandelt wird [34, S. 169]. Wenn Funktionen mit einem Namen versehen werden können, wie in den bekannten funktionalen Programmiersprachen, kann man eine rekursive Definition ohne einen speziellen Fixpunktoperator ausdrücken. Dennoch wird in Abschnitt 2.6 nach der Einführung der Haskell-Syntax eine Möglichkeit gezeigt, in Haskell einen Fixpunktoperator zu beschreiben.

2.3 Eigenschaften von Haskell

Haskell [4] ist eine funktionale Programmiersprache, die rein (*pure*) und nicht strikt ist. Die Nicht-Striktheit wird durch einen lazy-Auswertungsmechanismus erreicht [2.2.3].

Pure:

Während der Auswertung von Haskell-Ausdrücken treten keine sog. Seiteneffekte [35, 10] auf. Das Ergebnis eines Funktionsaufrufes ist allein durch seine Argumente festgelegt. Die einzigen Bindungsmechanismen für Variablen sind:

- Funktions- und Wertdeklaration (statisch)
- Bindung von formalen Parametern an Ausdrücke (dynamisch)

Dadurch bleibt der Wert von übergebenen Variablen und Ausdrücken nach einem Funktionsaufruf unverändert. Es gibt in funktionalen Sprachen kein Gegenstück zu Referenzparametern und Zeigern aus der imperativen Welt. Da Haskell dem funktionalen Konzept auf allen Abstraktionsebenen treu bleibt, dürfen die Deklarationen ohne weiteres als Gleichungen gelesen werden.[1]

Nicht strikt:

Eine Funktion `f` heißt strikt im i-ten Argument, wenn gilt [5]:

$$x_i = \bot \quad \Rightarrow \qquad \texttt{f}\ x_1 \ldots x_i \ldots x_n = \bot$$

Eine Funktion heißt strikt, falls sie strikt in allen Argumenten ist. Viele funktionale Sprachen kennen nur strikte Funktionen, z.B. SML [27]. Nur manche vorgegebenen Funktionen und Konstrukte werden speziell behandelt. So ist das `if-then-else` Konstrukt nur strikt im ersten Argument. In Haskell dagegen sind Funktionen grundsätzlich nicht strikt. Betrachten wir dazu die nächsten beiden Deklarationen:[2]

```
bottom :: a
bottom = bottom

first :: a -> b -> a
first x y = x
```

Der Ausdruck `bottom` repräsentiert den undefinierten Wert $\bot$, wobei $\bot$ der kleinste Fixpunkt der Gleichung `bottom = bottom` ist (denotationelle Semantik). In der operationellen Sichtweise wird die Beschreibung von `bottom` wiederum auf `bottom` zurückgeführt. Dadurch wird bei der Auswertung fortlaufend `bottom` durch `bottom` ersetzt, ohne zu einem endgültigen Wert zu gelangen. `first` bezeichnet eine konstante Abbildung im ersten Argument, d.h. der Wert des zweiten ist völlig irrelevant und tritt demnach auch nicht im Funktionsrumpf auf. Die Auswertung von `first 1 bottom` liefert in Haskell nun den Wert `1`, d.h. `first` ist nicht strikt.

[1] auf einen Aspekt, den es dabei zu beachten gibt, wird in Abschnitt 2.4.3 eingegangen

[2] eine Beschreibung der Haskell-Syntax folgt in den nächsten Abschnitten

Lazy:

Wie der Begriff *lazy* (dt. träge) erahnen läßt, wird versucht, mit möglichst geringem Aufwand das Resultat zu berechnen. Ausdrucksteile werden erst dann ausgewertet, wenn ihr Wert wirklich benötigt wird. Es wird also keine spekulative Auswertung betrieben. Dadurch ist man in der Lage, unendliche Datenstrukturen zu handhaben. Dies entspricht der lazy Reduktionsstrategie aus Abschnitt 2.2.3.

Sehen wir uns dazu folgendes an:

```
ones :: [Int]
ones = 1 : ones

length :: [a] -> Int
length   []   = 0
length (x:xs) = 1 + length xs

head :: [a] -> a
head (x:_) = x
```

Neben den bekannten Funktionen `length` und `head`, die die Länge bzw. den Kopf einer Liste berechnen, haben wir mit `ones` noch eine rekursive Definition. Mit `ones` wird eine unendliche Liste beschrieben, die nur Einsen enthält. Denn bei der strikten Auswertung tritt an die Stelle von `ones` immer `1:ones` und die fortwährende Ersetzung terminiert nicht. In Haskell liefert `head ones` aber trotzdem `1`, da `ones` nur ein einziges Mal ersetzt werden muß, damit mit dem Pattern `(x:_)` in der Definition von `head` eine Übereinstimmung erreicht und 1 an den formalen Parameter `x` gebunden wird. Aufgrund der Laziness wird der Rest der unendlichen Liste nicht betrachtet, weil er nicht benötigt wird. Bei der Berechnung von `length ones` hilft die Laziness allerdings nichts, da versucht wird, die Länge der unendlichen Liste zu bestimmen. Wir erhalten hier $\perp$. `first 2 (length ones)` ist davon nicht betroffen und ergibt `2`. Bei strikten Sprachen sind die Werte aller hier angegebenen Ausdrücke mit $\perp$ gleichzusetzen.

2.4 Haskell-Konstrukte

2.4.1 Funktions- und Wertdeklarationen

Ein funktionales Programm besteht im wesentlichen aus einer Menge von Funktions- und Wertdeklarationen. Mit diesen Deklarationen werden Bindungen zwischen Namen und Ausdrücken hergestellt. Die Namen können dann in anderen Ausdrücken verwendet werden. Dabei spielt es keine Rolle, an welcher Stelle der Name definiert wurde, ob vor oder nach seiner Anwendung in anderen Ausdrücken [30, 4.5]. Was die Auswertung betrifft, so kann man sich vorstellen, daß ein Name durch den an ihn gebundenen Ausdruck ersetzt wird. Durch sukzessive Wiederholung dieses Vorgehens gelangt man schließlich zum Ergebnis, sofern keine Endlosrekursion vorliegt. Da Haskell rein funktional ist und keine Seiteneffekte auftreten, läßt sich mit diesem Mechanismus der Wert eines Haskellausdrucks ermitteln. Die Substitution eines Namens durch den gebundenen Ausdruck, findet u.a. beim sog. *Equational Reasoning* Anwendung, das im folgenden Kapitel beschrieben wird.

Zunächst betrachten wir zwei einfache Funktionsdeklarationen:

```
add x y = x+y
inc :: Int -> Int
inc x = add 1 x
```

Bei `add` handelt es sich um eine Funktion mit zwei Argumenten, welche einfach addiert werden. `(+)` ist eine vordefinierte Funktion, die überladen ist. Das bedeutet, hinter ein und demselben Funktionssymbol, hier `(+)`, verbergen sich verschiedene Implementationen, je nachdem welchen Typ die Argumente haben. Bei der Deklaration von `inc` wurde zusätzlich eine Typsignatur angegeben. `inc` verwendet die vorher definierte Funktion `add`, um ihr Argument um Eins zu erhöhen. Eine Auswertung von `inc 5` per Hand könnte etwa folgendermaßen aussehen:

```
inc 5 -> add 1 5 -> 1+5 -> 6
```

Unterschied zwischen Funktions- und Wertdeklaration

Es sollte an dieser Stelle darauf hingewiesen werden, daß in Haskell Funktions- und Wertdeklarationen i.allg. nicht äquivalent sind, wenn sie auf oberster Ebene durchgeführt werden (*toplevel*-Deklaration). Definiert man `add` mit Hilfe einer λ-Abstraktion, die im Abschnitt 2.4.4 genauer beschrieben wird, als Wertzuweisung, etwa `add = \x y -> x+y`, so führt dies zu einem Fehler. Schuld daran ist die überladene Funktion `(+)`. Bei dieser Deklaration von `add` wird die sog. *monomorphism restriction* [30, 4.5.4] verletzt, welche in Haskell aus Effizienzgründen eingeführt wurde. Damit wird vermieden, daß ein Ausdruck, der durch eine Wertzuweisung gebunden ist, mehrfach ausgewertet wird, je nachdem in welchem Kontext man ihn verwendet. Innerhalb von `let`-Ausdrücken oder wenn eine Typsignatur angegeben wird, wie etwa `add :: Int -> Int -> Int`, wirkt sich diese Einschränkung nicht aus.

2.4.2 Benutzerdefinierte Datentypen

Haskell bietet eine Reihe von vordefinierten Datentypen. Dazu zählen numerische Typen, wie `Int` oder `Float`. Ferner kann der Benutzer auf strukturbildende Datentypen, wie Listen (z.B. `[1,2,5]`) oder Tupel (z.B. `(1,"",[])`), zurückgreifen, um die wichtigsten zu nennen. Trotz alledem ist man mit den vordefinierten Datentypen allein in der Programmgestaltung sehr eingeschränkt. Deshalb bekommt man die Möglichkeit, eigene Datentypen zu definieren. Damit lassen sich einfache Aufzählungstypen genauso realisieren wie komplexere rekursive Datentypen. In Haskell verwendet man hierfür das `data`-Konstrukt, um solche sog. *algebraische Datentypen* einzuführen.

```
data Wochentage = Mo | Di | Mi | Do | Fr | Sa | So
data KalenderWoche = Woche Int
```

Bei dem Datentyp `Wochentage` handelt es sich um einen einfachen Aufzählungstyp. Elemente eines mit `data` definierten Datentyps werden mit sog. Konstruktoren erzeugt. `Mo` ist ein solcher Konstruktor und repräsentiert einen Ausdruck vom Typ `Wochentage`. Konstruktoren kann man als spezielle Funktionen auffassen, deren Resultat ein Element vom zugehörigen Datentyp ist. Wenn wir uns den Konstruktor `Woche` ansehen, erkennen wir, daß er ein Argument vom Typ `Int` benötigt. Somit handelt es sich bei `Woche 1` um ein Objekt vom Typ `KalenderWoche`. Die Konstruktoren von `Wochentage` hingegen sind alle nullstellig, brauchen also kein weiteres Argument und stellen für sich allein ein Element vom Typ `Wochentage` dar.

In der Sichtweise von Funktionen ergeben sich für die obigen Konstruktoren folgenden Typsignaturen:

```
Mo,Di,Mi,Do,Fr,Sa,So :: Wochentage
Woche                :: Int -> KalenderWoche
```

Zur Illustration eines rekursiven Datentyps definieren wir uns die Struktur eines Baumes. Um die Struktur möglichst flexibel einsetzen zu können, soll der Typ der Elemente, die in den Knoten des Baumes gespeichert werden, nicht bei der Definition festgelegt werden, sondern variabel sein. Das bringt uns in etwa zu folgendem:

```
data Baum a = LeererBaum | Zweig a (Baum a) (Baum a)
```

Auf den Typbezeichner `Baum` folgt eine sog. Typvariable, in unserem Beispiel mit dem Namen `a`. Dabei dürfen im allgemeinen Fall durchaus mehr als nur eine einzige folgen. An die Stelle einer Typvariable kann jeder beliebige Typ treten. Man spricht in diesem Zusammenhang auch von polymorphen Datentypen, sobald eine Typvariable in der Definition auftritt.

Welchen Typ haben nun die Konstruktoren des Datentyps `Baum a`? In analoger Weise zu den vorherigen Datentypen erhält man:

```
LeererBaum :: Baum a
Zweig      :: a -> (Baum a) -> (Baum a) -> (Baum a)
```

Damit handelt es sich bei `Zweig 'b' LeererBaum LeererBaum` um ein Element vom Typ `Baum Char`, da wegen des Knotenelements `'b'` für die Typvariable `a` der Typ `Char` zu ersetzen ist.

Um mehr Übersicht in zusammengesetzte Typen zu bringen, lassen sich mit `type` Typsynonyme einführen. Die Anweisung

```
type Zeichenkette = Baum Char
```

bindet `Zeichenkette` an `Baum Char`. Es wird auf diese Weise aber *kein* neuer Datentyp erzeugt. Es ist vielmehr so, daß man sich jedes Auftreten von `Zeichenkette` im Programm implizit durch `Baum Char` ersetzt vorstellen muß.

2.4.3 Pattern-Matching

Nachdem man im letzten Abschnitt gesehen hat, wie eigene Datentypen definiert werden, bleibt noch die Frage zu klären, wie man auf den neuen Datenelementen arbeitet. Wie kann man etwa feststellen, ob ein Ausdruck vom Typ `Baum a` ein `LeererBaum` ist? Dazu verwendet man *Pattern-Matching* (PM). Ein Pattern (dt. Muster) ist ein spezieller Ausdruck, der mit einem anderen Ausdruck in Übereinstimmung gebracht werden soll. Dies geschieht formal, indem das Pattern mit dem Ausdruck unifiziert [2.1.3] wird. Dabei kann die Unifikation scheitern oder erfolgreich sein. Im Erfolgsfall werden die im Pattern freien Variablen gebunden. Es ist zu beachten, daß freie Variablen in einem Pattern nicht mehrfach auftreten dürfen, im Gegensatz etwa zur Programmiersprache *PROLOG* [7]. Sehen wir uns dazu folgendes an:

Pattern	Ausdruck	paßt	Bindungen
`v`	`7`	ja	`v=7`
`LeererBaum`	`Zweig 1 x y`	nein	
`[x,2,y]`	`[1,2,3]`	ja	`x=1, y=3`

In Haskell kommen Pattern innerhalb einer `case`-Anweisung zur Anwendung. Ein Pattern ist im wesentlichen ein Ausdruck über freien Variablen und Konstruktoren. Es ist zu beachten, daß ein Pattern keine Funktion darstellen darf, d.h. Konstruktoren müssen jeweils mit allen ihren Argumenten angegeben werden und am Kopf des Patterns (ganz links) darf keine Variable stehen, es sei denn, diese Variable allein bildet das Pattern.

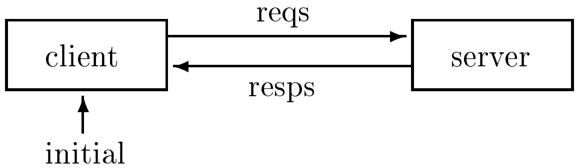
Einige kleine Beispiele sollen dies nun illustrieren:

Pattern	gültig	
`Baum x`	nein	Konstruktor `Baum` hat 3 Argumente
`f x`	nein	anstelle der Variablen `f` muß ein Konstruktor stehen
`x`	ja	paßt immer
`[y,y]`	nein	Variable `y` wird zweimal verwendet
`(LeererBaum,x)`	ja	

Haskell stellt noch spezielle Pattern zur Verfügung, die einerseits eine übersichtlichere Programmierung gestatten und andererseits – je nach Güte des Compilers bzw. Interpreters – zu einem effizienteren Programm beitragen.

`_`	Wildcard
`var@pat`	As-Pattern
`~pattern`	Lazy-Pattern

Unter einer Wildcard _ kann man sich eine freie Variable vorstellen, die nirgendwo im Programm verwendet wird. Eine Wildcard kann an einen beliebigen Ausdruck gebunden werden. Ein As-Pattern bindet Variable `var` an den Ausdruck, der mit `pat` abgedeckt wird. Diese beiden Pattern-Arten sind *syntactic sugar* und erhöhen die Ausdrucksfähigkeit nicht, d.h. sie lassen sich bereits mit anderen Sprachkonstrukten umschreiben. Sie spielen auch für die effizientere Codeerzeugung eine Rolle.[3] Lazy-Pattern sind gut für die rekursive Definition von unendlichen Datenstrukturen zu gebrauchen. Haskell ist zwar lazy, aber beim Pattern-Matching wird die Übereinstimmung eines Patterns mit einem Ausdruck vollständig bestimmt. Man kann hier keine Berechnung zurückstellen, solange nicht geklärt ist, ob das Pattern paßt oder nicht. Ein Lazy-Pattern erlaubt die Übereinstimmung eines Pattern mit einem Ausdruck vorwegzunehmen, ohne daß dies während des PM explizit durch Berechnung bestätigt wird. Erst später, wenn man auf eine Variable zugreift, wird nachträglich das herkömmliche PM durchgeführt. Wenn das Pattern dann nicht paßt, ergibt sich $\perp$. Nehmen wir z.B. ein einfaches client-server-Modell [17, 4.4], um die Anwendung von Lazy-Pattern zu demonstrieren. Ein `server` bearbeitet Anfragen von einem `client`, der seine weiteren Anfragen von den entsprechenden Antworten des `server` abhängig macht. Bei der Modellierung nimmt der `client` eine Startanfrage `initial` als Argument. Graphisch laßt sich dies folgendermaßen veranschaulichen:



[3] Der Compiler erhält auf diese Weise Information, die er andernfalls erst durch Programmanalyse bestimmen könnte.

Die Funktionen `process` und `next`, die in der anschließenden Implementierung zu finden sind, bearbeiten die erhaltenen Anfragen und Antworten, sollen uns aber hier nicht weiter beschäftigen.

```
reqs  = client initial resps
resps = server reqs

client x y = case (x,y) of
              (init,(rp:rps)) -> init : client (next rp) rps
server x   = case x of
              (rq:rqs)        -> process rq : server rqs
```

Betrachtet man die ersten drei Anfragen mit `take 3 reqs`, bekommt man kein Ergebnis, denn die Auswertung von `reqs` stützt sich auf `client initial resps`, was PM zwischen `(rp:rps)` und `resps` erfordert. `resps` kann auf `server reqs` reduziert werden. Die Auswertung dieses Ausdrucks erfordert PM zwischen `(rq:rqs)` und `reqs`. Die dafür notwendige Evaluierung von `reqs` bringt uns wieder zum Anfang des Problems zurück. Es kann also nie auf die initiale Anfrage des `clients` zugegriffen werden. Verwendet man nun ein Lazy-Pattern und ändert die Deklaration von `client` zu

```
client x y = case (x,y) of
              (init,~(rp:rps)) -> init:client (next rp) rps,
```

tritt das oben geschilderte Problem nicht auf und der angegebene Ausdruck läßt sich ohne weiteres auswerten, da das *Matching* von `(rp:rps)` und `reqs` erst auf der rechten Seite von `client` nachgeholt wird, wenn auf `rp` oder `rps` zugegriffen wird.

Man kann Pattern in zwei Klassen einteilen. Bei *refutable* Pattern kann die Unifikation mit dem betrachteten Ausdruck grundsätzlich scheitern. Solche Pattern erkennt man leicht daran, daß Konstruktoren in ihnen auftreten. Bei *irrefutable* Pattern kommt es hingegen immer zu einer Übereinstimmung. Vertreter dieser Klasse sind Wildcard `_`, Lazy-Pattern und Variablen.

Wie verhält sich nun das Pattern-Matching innerhalb eines `case`-Ausdrucks, wenn mehrere Alternativen zur Auswahl stehen? Hier werden die Pattern von oben nach unten ausprobiert und diejenige Alternative gewählt, bei der das PM als erstes erfolgreich war. Damit wirkt sich die Reihenfolge der Pattern auf das Programmverhalten aus, sofern die einzelnen Pattern nicht paarweise disjunkt sind. Das bedeutet, daß sie Bereiche eines Datenobjektes abdecken, die sich nicht überlappen. Sollte keines der Pattern passen, kommt es schließlich zu einem Laufzeitfehler. Sehen wir uns dazu folgendes an:

```
null ls                        leererBaum tr
 = case ls of                   = case tr of
     [] -> True                     LeererBaum -> True
     _  -> False                    Baum _ _ _ -> False
```

Die Vertauschung der Pattern in der Funktion `null` ist nicht anzuraten, da man dann die konstante Funktion `False` erhielte. Bei `leererBaum` spielt die Reihenfolge keine Rolle.

Pattern kann man auch innerhalb von Funktionsdeklarationen benutzen. Auch hier werden die einzelnen Alternativen von oben nach unten ausprobiert und der Rumpf analog zur `case`-Anweisung bei der Auswertung zugeteilt. Bei mehrstelligen Funktionen sind die Pattern einer Alternative zusätzlich von links nach rechts zu lesen.

```
null [] = True          take 0    _    = []
null _  = False         take _    []   = []
                        take i (x:xs) = if i<=0
                                        then []
                                        else x : take (i-1) xs
```

Die Funktion `null` wird oben mit Pattern innerhalb der Funktionsdeklaration notiert. Semantisch besteht kein Unterschied zu der vorherigen Deklaration von `null`. Die Funktion `take` nimmt die ersten `i` Elemente einer Liste. Sie ist mehrstellig, und daher müssen die Pattern innerhalb einer Alternative von links nach rechts abgearbeitet werden. `take 0 bottom` evaluiert zu `[]`, da `0` zu `0` und `_` zu `bottom` paßt. `take bottom []` liefert jetzt aber nicht `[]`, sondern $\perp$. Das liegt daran, daß die Unifikation von `0` mit `bottom` bereits $\perp$ liefert. Die zweite Alternative kommt damit nicht mehr zum Zuge. Vertauscht man die beiden Argumente von `take` bei der Deklaration, so erhält man bei beiden entsprechend modifizierten Anfragen jeweils `[]` als Resultat [17, 4.2]. Dies zeigt nochmals deutlich, daß die Reihenfolge von sich überlappenden Pattern entscheidenden Einfluß auf das Programmverhalten nimmt.

Gleichungen

Wenn eine Funktion mit Hilfe der Pattern-Schreibweise definiert wird, dann darf man die einzelnen Alternativen nur dann als Gleichungen auffassen, wenn die angegebenen Pattern sich nicht überlappen. Aus diesem Grund ist es falsch, aus der obigen Definition von `null`, die Gleichungen

$$\begin{gathered}\texttt{null []} \equiv \texttt{True} \\ (\forall \texttt{x} : \texttt{x} \text{ ist Liste} : \texttt{null x} \equiv \texttt{False})\end{gathered}$$

abzuleiten. Hier muß man entweder für disjunkte Pattern sorgen oder die einzelnen Alternativen einer Deklaration zusammenfassen, wie es später im Abschnitt 3.5.3 gezeigt wird. So ist es möglich, die beiden Gleichungen

$$\begin{gathered}\texttt{null []} \equiv \texttt{True} \\ (\forall \texttt{x,y:} \ \texttt{(x:y)} \text{ ist Liste} : \texttt{null (x:y)} \equiv \texttt{False})\end{gathered}$$

zu verwenden oder man beschränkt sich auf nur eine Gleichung, die beide (alle) Fälle beinhaltet.

$$(\forall \texttt{l:} \ \texttt{l} \text{ ist Liste} : \texttt{null l} \equiv \texttt{case l of} \ \{\ \texttt{[] -> True; \_ -> False}\ \})$$

Wenn davon gesprochen wird, daß eine Funktionsdeklaration als Gleichung aufzufassen ist, dann gilt das nur in der kompakten, letzten Darstellung. Andernfalls muß erst die Disjunktheit der Pattern nachgewiesen werden, um die Einzelgleichungen verwenden zu dürfen.

2.4.4 λ-Abstraktion

Funktionale Sprachen basieren auf dem λ-Kalkül [2.2], und die λ-Abstraktion bildet darin ein zentrales Sprachkonstrukt. Mit ihr lassen sich Funktionen definieren, ohne sie mit einem Namen zu versehen, im Gegensatz zu Funktionsdeklarationen. In Haskell wird eine λ-Abstraktion der Form $(\lambda\texttt{x.e})$ als `(\x->e)` notiert.

Betrachten wir dazu folgendes kleine Beispiel:

```
wende_an :: (a->b) -> a -> b
wende_an f x = f x

inc :: Int -> Int
inc x = x + 1

bsp1, bsp2 :: Int
bsp1 = wende_an inc 5
bsp2 = wende_an (\x->x+1) 5
```

`wende_an` nimmt eine Funktion und ihr Argument als Eingabe und führt explizit eine Funktionsapplikation aus. Wir haben es hier mit einer sog. *Funktion höherer Ordnung* (FHO) zu tun, da eines der Argumente von `wende_an` eine Funktion ist. Allgemein erkennt man FHOs leicht an ihrer Typsignatur, da diese mehr als einen Pfeil `->` enthält. In `bsp1` wird `wende_an` die Funktion `inc` übergeben, welche durch eine Funktionsdeklaration definiert wurde. Im Gegensatz dazu erhält `wende_an` in `bsp2` die äquivalente Funktion in Form einer λ-Abstraktion.

In Haskell können geschachtelte λ-Abstraktionen der Form $(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{e}))$ anstelle von `(\x->\y->e)` mit `(\x y->e)` abgekürzt werden. Dies erhöht vor allem bei großen Schachtelungstiefen die Lesbarkeit.

Um nochmals die zentrale Stellung von λ-Abstraktionen innerhalb von funktionalen Sprachen zu verdeutlichen, sei an dieser Stelle darauf hingewiesen, daß sich Funktionsdeklarationen damit auf einfache Wertzuweisungen zurückführen lassen. Die Deklaration `inc x = x+1` kann genauso gut wie `inc = \x -> x+1` geschrieben werden, solange die Typsignatur `inc :: Int -> Int` mitangegeben wird, weil in Haskell die Eins-zu-Eins-Beziehung von Funktions- und Wertdeklarationen nicht immer gegeben ist, im Gegensatz zu den meisten anderen funktionalen Sprachen. Darauf wurde kurz in Abschnitt 2.4.1 eingegangen.

Abgesehen vom Klassenkonzept und den damit verbundenen Sprachkonstrukten, worauf kurz in [2.5] eingegangen wird, lassen sich die weiteren Haskell-Sprachkonstrukte auf die bis jetzt genannten zurückführen. Es handelt sich demnach nur um eine syntaktische Erweiterung der Sprache, die eine bequemere bzw. strukturiertere Programmierung erlauben, aber die Ausdrucksmächtigkeit nicht vergrößern. Solche Konstrukte werden auch als syntactic sugar bezeichnet. Im folgenden werden noch kurz die wichtigsten vorgestellt.

2.4.5 `let`-Ausdruck

Damit lassen sich lokale Bindungen erzeugen, die nur innerhalb des `let`-Ausdrucks von Bedeutung sind, aber außerhalb nicht zum Tragen kommen. Diese Bindungen bilden eine neue Umgebung für den Ausdruck, über den sich das `let` erstreckt. Der `let`-Ausdruck kann z.B. dazu verwendet werden, einen Wert, der mehrfach benutzt wird, nur einmal berechnen zu lassen, indem man ihn einer Variablen zuweist. Diese Variable wird dann an den entsprechenden Stellen verwendet.

Als Beispiel dient die Funktion `quadsol`, welche die reellen Nullstellen der quadratischen Gleichung ax^2+bx+c=0 berechnet.

```
quadsol :: (Ord a, Floating a) => a -> a -> a -> Maybe (a,a)
quadsol a b c
 = let  x = b^2 - 4*a*c
        root = sqrt x
        sol sign = (-b+sign*root) / (2*a)
   in if x>=0 then Just (sol 1, sol (-1)) else Nothing
```

An der Kontext-Information in der Typsignatur von `quadsol`, die vor „`=>`“ angegeben wird, ist zu erkennen, daß es sich um eine überladene Funktion handelt. `quadsol` kann auf Argumente angewendet werden, auf denen eine Ordnung definiert ist (`Ord a`) und die die Eigenschaft von reellen Zahlen haben (`Floating a`). Das Resultat ist entweder `Nothing`, falls keine Lösung existiert, oder `Just`(*Lsg1*,*Lsg2*), wobei `Nothing` und `Just` die beiden Konstruktoren des vordefinierten algebraischen Datentyps `Maybe` sind. Die Bindungen von `x`, `root` und `sol` sind nur lokal verfügbar und treten nicht in Konflikt mit anderen Definitionen.

Eine weitere Konsequenz der neuen Umgebung ist das Überschreiben bereits bestehender Bindungen, ohne daß dies zu einem Fehler führt. Denn eine solche Redefinition auf derselben Deklarationsebene ist in rein funktionalen Sprachen wie Haskell verboten:

```
inc :: Int -> Int
inc x = x+1
bsp3  = inc (let inc x = x+2 in inc 5)
```

In `bsp3` wird `inc` lokal neu definiert, was zur Folge hat, daß bei `inc 5` zunächst eine Erhöhung um 2 stattfindet. Auf dieses Resultat wird dann die außerhalb definierte Funktion `inc` angewendet, da deren Verwendung nicht im `let`-Ausdruck gekapselt ist. Als Ergebnis ergibt sich somit `8`.

Einfache `let`-Ausdrücke ohne rekursive Definitionen können auf λ-Abstraktionen reduziert werden. `bsp3` kann demnach auch so aussehen:

```
bsp3 = inc ((\inc -> inc 5) (\x->x+2))
```

Anstelle der lokalen Deklaration `inc` wird hier das formale Argument `inc` in der λ-Abstraktion verwendet, an das bei der Auswertung `(\x->x+2)` gebunden wird. Bei rekursiven Deklarationen scheitert diese Transformation allerdings, da eine Applikation mit einem λ-Ausdruck nur einmal reduziert werden kann.

2.4.6 Listen-Komprehension

Listen spielen in vielen funktionalen Sprachen eine wichtige Rolle. Deswegen werden oft Funktionen zur Manipulation dieser Strukturen bereitgestellt. Mit der Listen-Komprehension (eng. list comprehension) kann man auf elegante Art und Weise neue Listen aus bereits bestehenden aufbauen. Diese Notation zeichnet sich vor allem durch ihre Übersichtlichkeit aus.

```
bsp4 :: [Int]
bsp4 = [ x+1  | x <- [1,2,3,4], even x]
bsp51, bsp52 :: [(Int,Int)]
bsp51 = [(x,y) | x <- [1,2], y<-[3,4]]
bsp52 = [(x,y) | x <- [1,2], y<-[1,2,3], x==y]
```

`bsp4` entspricht der Liste `[3,5]`. Durch den sog. Generator `x <- [1,2,3,4]` wird x nämlich nacheinander mit 1 bis 4 belegt. Davon werden diejenigen x, die `even x` erfüllen, also gerade sind, um eins erhöht. `even x` wird auch als Filter bezeichnet. Damit bilden nur 3 und 5 die Elemente der neuen Liste. In List-Comprehensions dürfen beliebig viele Generatoren und Filter in beliebiger Reihenfolge auftreten. So liefert `bsp51` die Liste `[(1,3),(1,4),(2,3),(2,4)]` als Resultat, da der zweite Generator alle Elemente zu jedem Element des ersten Generators erzeugt. In analoger Weise ergibt sich für `bsp52` die Liste `[(1,1),(2,2)]`.

In Abschnitt 3.5.1 wird gezeigt, wie die Listen-Komprehension auf elementare Haskell-Konstrukte zurückgeführt werden kann.

2.4.7 Arithmetische Sequenzen

Haskell ermöglicht auch eine einfache Angabe von Zahlbereichen. Mit der arithmetischen Sequenz kann der Benutzer eine Folge von Elementen notieren. Die Bereiche können dabei endlich oder, dank der Laziness, sogar unendlich viele Elemente enthalten.

```
bsp6, bsp7, bsp8 :: [Int]
bsp6 = [1..4]
bsp7 = [1,3..10]
bsp8 = [1..]
```

Hinter `bsp6` verbirgt sich die Liste `[1,2,3,4]`. Die Zeichensequenz „`..`“ bewirkt nämlich, daß alle Zahlen zwischen `1` und `4` in die Liste aufgenommen werden. Im Falle von `[1..0]` ergibt sich die leere Liste `[]`, da die Obergrenze kleiner als die Untergrenze ist. `bsp7` entspricht der Liste `[1,3,5,7,9]`, wobei sich die Differenz zweier aufeinanderfolgender Listenelemente aus `3-1` zu `2` ergibt. `bsp8` repräsentiert schließlich eine unendliche Liste, die mit eins beginnt und alle natürlichen Zahlen durchläuft. Das Fehlen einer Obergrenze hinter „`..`“ ist dafür verantwortlich, daß alle Elemente des Datentyps aufgezählt werden. Da Haskell lazy ist [2.3], können auch unendliche Strukturen sinnvoll eingesetzt werden.

Die obigen Beispiele mögen zwar den Eindruck erweckt haben, daß arithmetische Sequenzen auf ganzzahlige Folgen beschränkt sind. Allgemein lassen sich solche Sequenzen allerdings mit allen Datentypen notieren, die als Aufzählungstyp gekennzeichnet sind. Diese Eigenschaft wird im Klassenkonzept Haskells festgelegt, worauf kurz im nächsten Abschnitt eingegangen wird. Das abschließende Beispiel soll deshalb nur den allgemeineren Fall von arithmetischen Sequenzen demonstrieren.

```
data WochenTage
 = Mo | Di | Mi | Do | Fr | Sa | So
 deriving (Eq, Ord, Enum)

arbeitsTage = [Mo .. Fr]
```

`arbeitsTage` enthält die Element `Mo, Di, Mi, Do` und `Fr` in der resultierenden Liste.

2.5 Typsystem

Haskell besitzt ein statisches Typsystem, das typsichere (*type-safe*) Programme gewährleistet. Statisch heißt, daß der Typ von Ausdrücken bereits zur Kompilationszeit festgestellt wird und eine (dynamische) Prüfung zur Laufzeit entfällt. Die Typherleitung erfolgt automatisch und liefert den allgemeinsten Typ, der für den betrachteten Ausdruck gültig ist, sofern kein Typfehler vorliegt. Man kann Deklarationen und (Teil-)Ausdrücke mit einer Typsignatur versehen, die bei der Herleitung berücksichtigt wird. Eine solche Typangabe ist zwar optional, erhöht aber vor allem bei Deklarationen das Verständnis, weil deren Funktionalität direkt an Ort und Stelle abgelesen werden kann. Wenn nun der Benutzer eine Typvorgabe macht, wird diese mit dem abgeleiteten Typ verglichen. Die Typvorgabe ist gültig, wenn sie eine Instanz des abgeleiteten Typs darstellt. Ist dies der Fall, wird dem Ausdruck der u.U. speziellere Typ des Benutzers zugrundegelegt.

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

intLENGTH = length

intLength :: [Int] -> Int
intLength = length

newLength :: [a] -> Int
newLength = intLength
```

`intLength` und `length` unterscheiden sich darin, daß `intLength` nur auf `Int`-Listen angewendet werden darf, während `length` auf Listen beliebigen Typs operieren kann. Dies wird durch die einschränkende Typsignatur von `intLength` erreicht. Sie ist eine Instanz des Typs von `length`, da man anstelle der Typvariablen `a` den Typ `Int` einsetzen kann. `intLENGTH` bekommt keine solche Einschränkung, was dazu führt, daß dieser Funktion der allgemeinste Typ zugewiesen wird, nämlich `[a] -> Int`. Diese Deklaration von `newLength` muß scheitern, da die Typsignatur `[a] -> Int` zu allgemein gewählt wird. Sie kann nämlich nicht durch Spezialisierung von `[Int] -> Int` erreicht werden. Obwohl sich `newLength` ebenfalls auf `length` zurückführen läßt, ist diese Deklaration wegen der Typspezialisierung bei `intLength` nicht möglich.

Klassenkonzept

Das statische Typsystem Haskells basiert auf dem traditionellen Hindley-Milner Typ-System [26]. Dieses System wurde allerdings um ein Klassenkonzept [11] erweitert, das in strukturierter und kontrollierter Weise das Überladen von Funktionen ermöglicht. Polymorphe Typen, d.h. Typen mit einer Typvariablen, sind grundsätzlich allquantifiziert aufzufassen. Der Typ von

```
length :: [a] -> Int
```

ist im Grunde genommen als

```
length :: (∀ a : a ist Typ : [a] -> Int)
```

zu lesen. Durch das Klassenkonzept wird nun eine Einschränkung der Typen erreicht, welche anstelle einer Typvariablen eingesetzt werden dürfen. Es gibt nämlich Funktionen, die nur auf einer Auswahl der Datentypen sinnvoll definiert werden können. So hat das arithmetische `(+)` zwar für numerische Datentypen eine Bedeutung, nicht aber für Zeichenketten. Ein wesentlicher Unterschied zu *polymorphen* Funktionen ist, daß je nach Typ eine spezielle Implementation für diese Art von Funktion verwendet wird. Es handelt sich hier nämlich um sog. *überladene* Funktionen. Manchmal wird auch der Begriff *adhoc polymorphism* verwendet. Was ist zu tun, wenn der Programmierer keine überladenen Funktionen definieren kann, wie z.B. in SML? Dann müssen derartige Funktionen einfach für die in Frage kommenden Datentypen unter verschiedenem Namen definiert werden. Betrachten wir dazu den Gleichheitstest, der überprüft, ob zwei Datenelemente übereinstimmen. Man erhält dann vielleicht für die Datentypen `Int` und `Bool` die Funktionen `eqInt :: Int -> Int -> Bool` und `eqChar :: Char -> Char -> Bool`. Der Nachteil des Ganzen ist, daß beide Funktionen in keiner Beziehung zueinander stehen, obwohl sie doch eines gemeinsam haben: sie berechnen die Gleichheit zweier Datenelemente. Funktionen, die ihrerseits auf derart überladene Funktionen zugreifen, muß man zusätzliche Argumente spendieren, um damit sinnvoll umgehen zu können. Sehen wir uns dazu die Funktion `splitat` an, die eine Liste an einem gegebenen Element aufspalten soll.

```
splitat :: (a->a->Bool) -> a -> [a] -> ([a],[a])
splitat eq x []
 = ([],[])
splitat eq x (y:ys)
 = if eq x y
   then ([],y:ys)
   else let (l,r) = splitat eq x ys
        in  (y:l,r)
```

Das erste Argument von `splitat` ist nur dazu da, die Gleichheitsfunktion zu übergeben. Wenn nun noch andere überladene Funktionen, z.B. zur Beschreibung der Ordnung, verwendet werden oder wenn mehrere Gleichheitsfunktionen an verschiedenen Stellen benötigt werden, müßten nach diesem Schema dementsprechend mehr Argumente angegeben werden.

Haskell ermöglicht es, solche überladenen Funktionen zusammenzufassen. Dies geschieht, indem die entsprechenden Typen eine Klasse bilden und die Funktion zu einer Klassenfunktion wird. An einem Beispiel soll dies nun skizziert werden, genauere Angaben finden sich in [17] und [30].

```
data Switch = On | Off

eqSwitch On  On  = True
eqSwitch Off Off = True
eqSwitch _   _   = False

class MyEq a where
      (===)  :: a -> a -> Bool

instance MyEq Switch where
   s1 === s2  =  eqSwitch s1 s2
```

Mit `class MyEq a` wird der Name der neuen Klasse `MyEq` festgelegt. Die Typ-Variable `a` steht für diejenigen Datentypen, die in dieser Klasse zusammengefaßt werden. Hinter `where` folgt die Angabe der Klassenfunktionen, für die eine Instanz zu definieren ist, um die Zugehörigkeit eines Datentyps zur Klasse zu erhalten. In unserem Fall handelt es sich nur um die Funktion `(===)` mit Typ-Signatur `a -> a -> Bool`. Um den Datentyp `Switch` in die Klasse `MyEq` aufzunehmen, bedarf es der Anweisung `instance MyEq Switch`. Anstelle der Typ-Variablen `a` tritt nun `Switch`. Die Implementation von

`(===)` für den Datentyp `Switch` folgt in der `where`-Klausel, d.h. `(===)` hat in diesem Fall den Typ `Switch -> Switch -> Bool`. Damit ist jetzt ein Vergleich von `On === Off` möglich, aber die Angabe von `False === False` im Programm liefert einen Compiler-Fehler, da der Typ `Bool` nicht zur Klasse `MyEq` gehört. Dieser Vergleich kann erst dann durchgeführt werden, wenn man `Bool` in die Klasse `MyEq` aufnimmt.

Definieren wir jetzt die Funktion `splitat` so, daß sie auf unsere Gleichheitsfunktion zugreift:

```
splitat :: MyEq a => a -> [a] -> ([a],[a])
splitat x []     = ([],[])
splitat x (y:ys) = if x === y
                   then ([],y:ys)
                   else let (l,r) = splitat x ys
                        in  (y:l,r)
```

Nun übernimmt Haskell die Verwaltung der überladenen Funktionen. Der Benutzer braucht keine zusätzlichen Argumente mitschleppen, wenn er solche Funktionen benutzt. Dies wird intern vom Haskell-Compiler bewerkstelligt.

2.6 Beschreibung eines Fixpunktoperators in Haskell

Der in Abschnitt 2.2.4 gezeigte Fixpunktoperator Y kann in Haskell nicht verwendet werden, da sich dafür kein endlicher Typ ableiten läßt. Dadurch, daß in Haskell (wie in SML) für Funktionen Namen vergeben werden können, ist Haskell auf keinen speziellen Fixpunktoperator angewiesen. Dennoch soll hier als Ergänzung zu 2.2.4 eine mögliche Definition angegeben werden. Mit

```
y :: (a -> a) -> a
y h = h (y h)
```

kann ein geeigneter Fixpunktoperator definiert werden. `h` ist vom Typ `(a -> a)` und enthält die Beschreibung der (möglicherweise) rekursiven Funktion. Möchte man z.B. mit Hilfe von `y` die Fakultätsfunktion angeben, kann man dabei so vorgehen:

```
fac :: Int -> Int
fac = y (\fac_ n ->
          if n <= 0 then 1
                    else n * fac_ (n-1))
```

Die Verwendung eines Fixpunktoperators ist hier aber – wie bereits oben erwähnt – nicht notwendig. Eine rekursive Beschreibung der Fakultät würde in Haskell beispielsweise so aussehen:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
fac _ = error "negative input"
```

Kapitel 3

Equational Reasoning

Um Programmtransformationen in funktionalen Programmen auf formale Korrektheit hin zu überprüfen, gibt es die Beweismethode des sog. *Equational Reasoning* [18, 28, 4]. Dabei betrachtet man in einem Ausdruck einen Teilausdruck und ersetzt ihn durch einen semantisch äquivalenten. Auf diese Weise kann man einen Ausdruck gezielt manipulieren, und die Gleichheit ist bei jedem Transformationsschritt sichergestellt. Wann ein solches Vorgehen sinnvoll ist, soll in einer einführenden Motivation geklärt werden. Darauf folgt ein kleines Beispiel, an dem die Durchführung eines Equational Reasoning Beweises demonstriert wird. Für komplexere Beweise wird das Zusammenspiel von Equational Reasoning und Induktionsbeweisen erläutert. Schließlich werden die Anforderungen an ein System formuliert, das für die automatische Verifikation von solchen Beweisen eingesetzt werden soll. Am Schluß dieses Kapitels werden spezielle Haskell-Konstrukte mit Hilfe von Equational Reasoning auf elementare Sprachkonstrukte zurückgeführt, was im weiteren Verlauf eine einheitliche Behandlung erlaubt und weniger Spezialfälle verursacht.

3.1 Motivation

Gehen wir davon aus, daß wir einen gegebenen Algorithmus zunächst auf einfache Art und Weise implementieren, um seine korrekte Umsetzung in dieser Form leichter zeigen zu können. Wie kommt man anschließend zu einer effizienteren Variante, egal ob hinsichtlich Laufzeit oder Speicherplatzbedarf? Zwei Möglichkeiten kommen in Betracht:

- Neu-Implementation
- schrittweise Transformation der bestehenden Implementierung

Die erste Alternative ist nicht zu empfehlen, da man die Arbeit, die man in die erste Implementierung gesteckt hat, im großen und ganzen nicht nutzen kann. Außerdem muß für jede Neu-Implementierung streng genommen ein eigener Korrektheitsbeweis durchgeführt werden. Wenn man sich allerdings für die zweite Variante entscheidet und nur solche Transformationsschritte zuläßt, die äquivalenz-erhaltend sind, reicht es aus, die Korrektheit *genau* einmal zu zeigen. Man weist also für das Ausgangsprogramm nach, daß es die angegebenen Anforderungen erfüllt, und führt danach äquivalenz-erhaltende Transformationen durch. Das hat zur Folge, daß jede solche Umformung wieder zu einer korrekten Umsetzung des Algorithmus führt. Wenn nun diese Umformungen auf eine Effizienzsteigerung hin ausgerichtet sind, erhält man auf diese Weise nach Abschluß aller Modifikationen eine semantisch äquivalente, effizientere Umsetzung des ursprünglichen Algorithmus. Natürlich ist die richtige Wahl geeigneter Transformationen nicht offensichtlich und erfordert u.U. einen nicht unerheblichen Aufwand.

Was hat das Ganze nun mit *Equational Reasoning* zu tun?

Betrachtet man die ursprüngliche Implementierung des Algorithmus und die in mehreren Schritten abgeleitete, so wird ihre semantische Gleichheit dadurch sichergestellt, daß äquivalenz-erhaltende Transformationen eingesetzt werden. Die Verwendung einer solchen Transformation kann man gewissermaßen auf eine implizite Regelanwendung zurückführen, wie folgendes Beispiel zeigt:

$$\texttt{exp (head (x:xs))} \equiv \texttt{exp (x)}$$

Der Übergang von `exp (head (x:xs))` nach `exp (x)` ist durch die Gleichheit bzw. die Regel

$$\texttt{head (v:vs)} \equiv \texttt{v}$$

gerechtfertigt. Wenn man nun eine Folge derartiger Transformationen betrachtet, erhält man auf diese Weise einen Beweis, in dem sich jeweils zwei aufeinanderfolgende Umformungen durch genau eine Regelanwendung unterscheiden. Somit haben wir es mit einem ER-Beweis zu tun.

Die gezielte Modifikation eines Programmes, um eine bestimmte Eigenschaft zu erreichen, ist keine Spielerei, sondern kommt z.B. in funktionalen Compilern während der Optimierung und Code-Erzeugung zum Einsatz [32]. Allerdings sind die Möglichkeiten automatischer Verfahren beschränkt. Es kann aber für den Programmierer durchaus sinnvoll sein, überschaubare Programme bzw. Programmfragmente auf diese Weise per Hand zu bearbeiten, um gewünschte Eigenschaften zu erreichen.

3.2 Beispiel

Betrachten wir einmal den Ausdruck `head (map f (drop i xs))`, wobei `xs` eine beliebige Liste, `i` eine natürliche Zahl und `f` eine Funktion ist, die auf Listenelementen von `xs` arbeitet. Die Funktionen `drop`, `map`, `head` und `(!!)` sind in Haskell vordefiniert:

```
drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = if n>0 then drop (n-1) xs
                       else error "drop"

map :: (a->b) -> [a] -> [b]
map f []     = []                    -- map.1
map f (x:xs) = f x : map f xs        -- map.2

head :: [a] -> a
head (x:_) = x                       -- head
head _     = error "head"
```

```
(!!) :: [b] -> Int -> b
(x:_)  !! 0  = x
(_:xs) !! n  = if n>0 then xs !! (n-1)
                         else error "(!!)"
```

Wie kann man nun oben genannten Ausdruck vereinfachen? Irgendwie sollte man über den Teilausdruck `drop i xs` auf die verbleibenden Listenelemente schließen können. Nur hilft einem die Variable `xs` hier nicht weiter. Allerdings kann man eine Liste `xs` auch anders darstellen, und zwar durch:

(A1) `xs` ≡ `[xs!!j | j <- [0..length xs-1]]`

Man repräsentiert eine Liste `xs` einfach durch Aufzählung ihrer Elemente. Dazu wird der Index-Operator `(!!)` verwendet, der das `j`-te Element einer Liste liefert. Der Index-Bereich beginnt dabei bei 0, d.h. `xs!!0` gibt das erste Element der Liste `xs` zurück. Die oben genannte Repräsentation gilt genaugenommen nur für endliche Listen, sie ist aber in dieser Form sehr einfach zu verstehen. Eine Verallgemeinerung, die auch unendliche Listen einschließt, kann etwa so aussehen:

(A2) `xs` ≡ `[xs!!j | (j,_) <- zip [0..] xs]`

Dabei ist `zip` die vordefinierte Funktion:

```
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _ _ = []
```

Der Einfachheit halber belassen wir es bei der ersten Gleichung. Was kann man nun über den Ausdruck `drop i [xs!!j | j <- [0..length xs-1]]` aussagen? Eine genauere Analyse führt zu folgender Äquivalenz:

(A3)
`drop i [xs!!j | j <- [0..length xs-1]]`
≡
`[xs!!j | j <- [i..length xs-1]]`

An dieser Stelle sollen (A1) und (A3) als Axiome aufgefaßt werden, deren Richtigkeit bereits gezeigt wurde. Andernfalls müßten diese Gleichungen noch nachgewiesen werden.

Mit dieser Information ausgestattet, kann man nun die Gleichheit

`head (map f (drop i xs))` ≡ `f (xs!!i)`

zeigen. Gehen wir beim Beweis zunächst von `i<length xs` aus. Dann kann man folgende Transformationen durchführen:

```
head (map f (drop i xs))
      = {- (A1) -}
head (map f (drop i [xs!!j | j <- [0..length xs-1]]))
      = {- (A3) -}
head (map f [xs!!j | j <- [i..length xs-1]])
      = {- Abspaltung des ersten Listenelementes -}
head (map f ((xs!!i):[xs!!j | j <- [i+1..length xs-1])
      = {- map.2 -}
head (f (xs!!i) : map f [xs!!j | j <- [i+1..length xs-1]]
      = {- head -}
f (xs!!i)
```

Damit hat man die oben angegebene Gleichheit nachgewiesen, allerdings nur für `i<length xs`. Was passiert aber, wenn `i`$\geq$`length xs` ist? Wie man der Definition von `drop` leicht entnehmen kann, ergibt die Auswertung von `drop i xs` die leere Liste `[]`. Das führt uns schließlich über die Definition von `map` zum Ausdruck `head []`. Dessen Berechnung ist nicht definiert und führt zu einem Fehler. Diese Ausnahme kann man auch als $\perp$ auffassen, da ein `error`-Ausdruck für einen undefinierten Wert steht. Wie verhält sich nun `f (xs!!i)`? Auch hier erhält man einen undefinierten Wert, da der Index `i` nicht in der Liste `xs` enthalten ist. Somit zeigen beide Ausdrücke für den Fall `i>=length xs` dasselbe Verhalten.

Insgesamt hat man mit obigem ER-Beweis und den daran anschließenden Überlegungen zu `i`$\geq$`length xs` folgendes gezeigt:

$$\begin{array}{ll} \forall\ \texttt{i, f, xs : i} \in \mathbb{N},\ \texttt{xs::[}\alpha\texttt{], f::}\alpha \rightarrow \beta & \\ \quad \texttt{i} < \text{length xs} \Rightarrow & \texttt{head (map f (drop i xs))} \equiv \texttt{f (xs!!i)} \\ \quad \texttt{i} \geq \text{length xs} \Rightarrow & \texttt{head (map f (drop i xs))} \equiv \texttt{f (xs!!i)} \equiv \perp \end{array}$$

Bei unendlichen Listen `xs` muß genaugenommen das Axiom (A2) anstatt (A1) verwendet werden. Das schließt noch eine kleine Anpassung von Axiom (A3) mit ein, dann kann aber der obige Beweis fast wörtlich übernommen werden.

Die Verwendung von unendlichen Listen `xs` hat aber nur in lazy Sprachen wie Haskell einen Sinn, da für strikte Sprachen die Auswertung von `xs` Nichttermination zur Folge hat. Es ergibt sich also unabhängig vom Index `i` immer $\perp$.

3.3 Induktionsbeweise

Nicht alle Beweise verlaufen so „geradlinig“ wie im vorangegangenen Beispiel. Es kann u.U. erforderlich sein, Induktionsbeweise zu führen. Bei diesen müssen einerseits Basisfälle nachgewiesen werden und andererseits – darauf aufbauend – Induktionsfälle, die sich auf eine Induktionshypothese abstützen. Das Prinzip der strukturellen Induktion läßt sich so zusammenfassen [5]:

> Sei M die kleinste Menge, die folgende Anforderungen erfüllt:
>
> - $a_i \in M$ für alle $1 \leq i \leq k$
> - $m \in M \;\Rightarrow\; (C_j\, m) \in M$ für alle $1 \leq j \leq l$
>
> Um die Gültigkeit der Eigenschaft $P(m)$ für alle $m \in M$ nachzuweisen, kann der Beweis in zwei Teile aufgespalten werden.
>
> - Induktionsanfang
>   Zeige die Gültigkeit von $P(a_i)$ für alle $1 \leq i \leq k$, d.h. weise die Korrektheit für die Basisfälle nach.
> - Induktionsschritt
>   Unter der Annahme, daß $P(m)$ für ein beliebiges $m \in M$ gilt (Induktionshypothese), zeige die Gültigkeit von $P(C_j\, m)$ für alle $1 \leq j \leq l$. Damit werden die Induktionsfälle nachgewiesen.
>
> Wenn diese beiden Teile erfolgreich behandelt wurden, ist die Korrektheit von $P(m)$ für alle $m \in M$ erbracht.

Die Anzahl der Basis- und Induktionsfälle hängt sowohl vom Problem als auch von den beteiligten Datentypen ab. Bei Listen hat man oft einen Basisfall für die leere Liste und einen Induktionsfall, der Listen mit mindestens einem Element abdeckt. Sehen wir uns dazu folgendes Problem an.

`length xs` ≡ `length (map f xs)`, `length xs` endlich

Um den Nachweise der Gleichheit dieser beiden Ausdrücke zu erbringen, zeigt man zunächst, daß diese Aussage für die leere Liste gültig ist. Dies bildet den Basisfall.

Basisfall: `xs=[]`

(linke Seite)	`length [] = 0`
(rechte Seite)	`length (map f []) = {- map.1 -} length [] = 0`

Nachdem die Gleichheit für diesen Fall gezeigt wurde, kann man daran gehen, das Problem für `xs=y:ys` unter der Voraussetzung zu betrachten, daß die Gleichheit von `length ys` ≡ `length (map f ys)` gegeben ist (Induktionshypothese). Damit läßt sich der Induktionsfall so beschreiben.

Induktionsfall: `xs=(y:ys)` und `(Hyp) length ys` ≡ `length (map f ys)`

(linke Seite)	`length (y:ys) =`
	`1+length ys = {-(Hyp)-}`
	`1+length (map f ys)`
(rechte Seite)	`length (map f (y:ys)) = {- map.2 -}`
	`length (f y:map f ys) =`
	`1+length (map f ys)`

Mit dem Nachweis des Basis- und Induktionsfalles wurde hier die Gültigkeit der obigen Gleichung nachgewiesen. Für den Fall `xs=[x]` entspricht die Induktionshypothese gerade dem oben bewiesenen Basisfall. Mit dieser Methode kann man die Gültigkeit der Gleichung für beliebige *endliche* Listen zeigen, aber `xs` steht niemals für eine unendliche Liste. Eine Möglichkeit, Beweise über unendliche Datenstrukturen zu führen, ist die Verwendung von *Koinduktion* [25, 8, 9].

Die Induktion wurde hier über die Länge der Liste `xs` geführt. Der Basisfall deckt die Länge `Null` ab und der Induktionfall die Länge `n+1` ($n \in \mathbb{N}$), wobei die Behauptung für Listen der Länge kleiner gleich `n` vorausgesetzt wird.

Bei Induktionsbeweisen muß dafür Sorge getragen werden, daß alle Basis- und Induktionsfälle behandelt werden. Erst dann ist der Beweis vollständig.

3.4 Problemstellung

Solange ein Beweis mit nur wenigen Zwischenschritten geführt werden kann und nur einfache Regeln zur Anwendung kommen, die keine zusätzlichen Bedingungen einführen, bleiben ER-Beweise überschaubar und können durchaus per Hand verifiziert werden. Andernfalls ist es von Vorteil, diese Aufgabe einem automatischen System zu übertragen, um mögliche Fehlerquellen weitestgehend auszuschließen.

3.4.1 Anforderungen

Um die Verifikation von ER-Beweisen automatisch durchführen zu können, benötigt ein derartiges Verfahren neben dem Beweis bzw. den Transformationsschritten eine Datenbasis mit Regeln. Wenn sich die Gleichheit von Ausdrücken nicht direkt zeigen

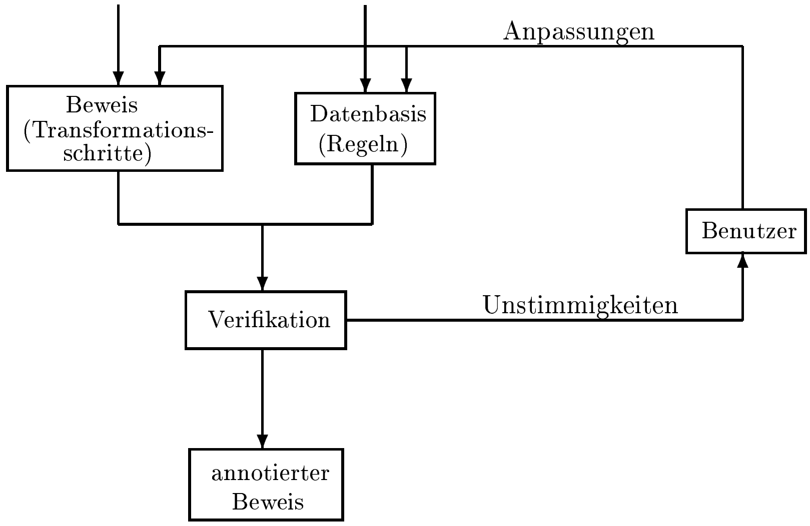


Abbildung 3.1: Anforderungen

läßt, kann in dieser Datenbasis nach einer passenden Regel gesucht werden. Die Regeln sollen dabei nicht gerichtet sein, d.h. sie dürfen von links nach rechts und von rechts nach links angewendet werden. Das Problem von Zyklen in der Herleitung ergibt sich nicht, da pro Beweisschritt höchstens eine Regelanwendung beteiligt sein darf, worauf kurz im nächsten Abschnitt eingegangen wird. Ferner muß ein Mechanismus zur Verfügung stehen, der die Eingabe von Induktionsbeweisen erlaubt. In den Fällen, in denen die Gültigkeit von Transformationsschritten mit der bereitgestellten Datenbasis nicht nachgewiesen werden kann, muß der Benutzer auf diesen Umstand hingewiesen werden, damit er den Beweis dementsprechend anpassen kann. Am Ende einer erfolgreichen Verifikation sollte die Möglichkeit bestehen, einen annotierten Beweis auszugeben, der die während der Überprüfung aufgetretenen Bindungen und Regelanwendungen auflistet. Die oben angeführten Anforderungen können in etwa so wie in Abbildung 3.1 zusammengefaßt werden.

Das vorgestellte System unterstützt diese Anforderungen, aber es bietet dem Benutzer zusätzlich die Möglichkeit, interaktiv nach anderen Lösungsvarianten suchen zu lassen, die andere Regelanwendungen einschließen und sich hinsichtlich der aufgetretenen Bindungen unterscheiden.

Da sich ER-Beweise vor allem für reine funktionale Sprachen anbieten, da dort Funktions- und Wertdeklarationen als Gleichungen gelesen werden können,[1] wird Haskell als Eingabesprache gewählt. Das System selbst ist ebenfalls in Haskell realisiert.

3.4.2 Einschränkung auf eine Gleichung

Die Restriktion, daß pro Beweisschritt nur eine Regel verwendet wird, liegt zum einen darin begründet, daß dadurch die Anwendung der Regeln transparent bleibt und zum anderen, daß der Suchraum wesentlich eingeschränkt wird. Das bringt den Vorteil, daß mit vertretbarem Aufwand entschieden werden kann, ob zwei aufeinanderfolgende Beweisschritte durch höchstens eine Regelanwendung nachvollziehbar sind.

[1] in Abschnitt 2.4.3 wurde auf diesen Aspekt näher eingegangen

3.5 Umformung von Haskell-Konstrukten

Haskell unterstützt eine Reihe von Sprachkonstrukten, die allein dazu dienen, eine einfachere und strukturiertere Programmierung zu ermöglichen. Sie können aber ohne weiteres auf andere, elementarere Konstrukte zurückgeführt werden. Bei der Gleichheitsprüfung ist dies von Vorteil, da sich dadurch die Anzahl der Fälle verringert, die während der Verifikation behandelt werden müssen. Es reicht nämlich, die Prüfung nur für die Basiskonstrukte durchzuführen.

Grundlage für diese Transformationen ist *Equational Reasoning*, indem höhere Sprachkonstrukte sukzessive durch einfachere ausgedrückt werden [30]. Diese Umformungen werden vom System implizit angewendet, ohne daß dazu spezielle Regeln anzugeben sind.

Die Transformationen werden in den folgenden Abschnitten jeweils durch den Pfeil „$\Rightarrow$“ ausgedrückt.

3.5.1 Listen-Komprehension

Listen-Komprehensionen werden dazu verwendet, aus bereits bestehenden Listen neue Listen aufzubauen [2.4.6]. Die Notation ist kompakt und übersichtlich. Sie erinnert an die mengentheoretische Schreibweise $\{fx \mid x \in M \wedge Px\}$. Das ist die Menge aller Elemente fx, wobei x aus der Menge M stammt und die Bedingung P erfüllt. Man muß nur $\{$, $\in$ und $\wedge$ durch `[`, $\leftarrow$ und `,` ersetzen und bei der verbalen Erklärung Menge durch Liste austauschen. Wenn man nun noch vernachlässigt, daß die Elemente einer Liste im Gegensatz zu Mengen geordnet sind und mehrmals vorkommen können, stimmen beide Definitionen überein.

Eine List-Comprehension (LC) läßt sich durch eine Reihe von vordefinierten Haskell-Funktionen umschreiben. Darunter leidet aber die Lesbarkeit, sobald sich mehrere sog. *Qualifier* in der LC befinden.

Eine List-Comprehension hat folgenden Aufbau:

LC	$\rightarrow$	`[ exp` $\mid$ $\texttt{qual}_1, \ldots, \texttt{qual}_n$`]`	
`qual`	$\rightarrow$	`pat` $\leftarrow$ `exp`	*(Generator)*
	$\mid$	`exp`	*(Filter)*
	$\mid$	`let decls`	*(lokale Bindung)*

Es gibt drei Arten von Qualifiern ($\texttt{qual}_i$), nämlich sog. *Generatoren*, *Filter* und *lokale Bindungen*. Die Qualifier sind von links nach rechts abzuarbeiten, d.h. die Reihenfolge ist entscheidend und hat Auswirkungen auf das Resultat der neuen Liste. Der Generator `pat` $\leftarrow$ `exp` bewirkt, daß das Pattern `pat` nacheinander mit den Elementen der Liste in `exp` unifiziert wird. Wenn die Unifikation erfolgreich verläuft, werden dabei entstehende Bindungen für die Abarbeitung der restlichen Qualifier weitergegeben. Ansonsten wird das entsprechende Listenelement übersprungen und beim nächsten fortgefahren. Der Filter `exp` ist ein boolescher Ausdruck. Evaluiert er zu `False`, werden die betrachteten Bindungen verworfen und die nächste Alternative kommt zum Zuge. Wenn sich der Ausdruck zu `True` ergibt, wird der folgende Qualifier behandelt. Mit den lokalen Definitionen `let decls`, deren Bindungen bei den darauffolgenden Qualifiern in Erscheinung treten, wird innerhalb der LC ein `let`-Konstrukt [2.4.5] zur Verfügung gestellt. Erst nachdem alle Qualifier der LC erfolgreich abgearbeitet wurden, wird der neue Listeneintrag mit den aktuellen Bindungen generiert.

Die folgenden Transformationsregeln [30, 3.11] reduzieren eine LC in eine Folge von Funktionsanwendungen. Damit braucht das Konstrukt der LC nicht speziell bei der Gleichheitsprüfung behandelt zu werden.

```
[ e | b ]           ⇒ if b then [e] else []                 (1)
[ e | let decls ]   ⇒ let decls in [e]                       (2)
[ e | q1, q2 ]      ⇒ concat [ [e | q2 ] | q1]               (3)
[ e | p ← l ]       ⇒ let ok p = True; ok _ = False
                       in map (\p → e) (filter ok l)         (4)
```

`filter` und `concat` sind wie `map`, das bereits in Abschnitt 3.2 eingeführt wurde, vordefinierte Haskell-Funktionen und werden hier nur der Vollständigkeit halber angegeben.

```
filter p [] = []
filter p (x:xs)
 = if p x then x : filter p xs
          else     filter p xs

concat []     = []
concat (l:ls) = l++concat ls
```

`filter` übernimmt aus einer Liste nur diejenigen Elemente, die das Prädikat `p` erfüllen, und `concat` fügt eine Liste von Listen zu einer einzigen Liste zusammen.

Betrachten wir die obigen Umformungsregeln an einem kleinen Beispiel:

```
loglist xs = [ log x | x ← xs, x>0 ]
      → {- (3) -}
loglist xs = concat [ [log x | x>0] | x ← xs ]
      → {- (1) -}
loglist xs = concat [ if x>0 then [log x] else [] | x ← xs ]
      → {- (4) -}
loglist xs = concat (let ok x = True; ok _ = False
             in map (\x → if x>0 then [log x] else [])
                    (filter ok xs)
```

Das System behandelt allerdings Generatoren der Form `v ← vs` gesondert, indem abweichend zur Regel (4) keine `let`-Anweisung und kein Filter-Schritt eingefügt werden, da die boolesche Funktion `ok` in diesem Fall konstant `True` ist. Stattdessen verwendet man die einfachere Transformation `map (\p→e) l`, wodurch obiges Beispiel zu

```
loglist xs = concat (map (\x → if x>0 then [log x] else []) xs)
```

reduziert wird.

3.5.2 Konditionale Verzweigung

Mit dem Konstrukt `if-then-else`, das auch in anderen Programmiersprachen wohlbekannt ist, können Ausdrücke in Abhängigkeit einer Bedingung ausgewertet werden. Diese Anweisung läßt sich auf eine elementare `case`-Anweisung reduzieren.

```
                         case b of
if b then t else f ⇒       True  → t
                           False → f
```

Dabei wird die Bedingung mit den beiden booleschen Konstruktoren verglichen und abhängig davon der entsprechende Zweig ausgewertet.

Obwohl die `case`-Anweisung zunächst komplizierter aussehen mag als das `if-then-else`, bietet sie dennoch den Vorteil, daß bei der Auswertung kein Spezialfall behandelt werden muß. Ferner braucht man keine Regel, die eine Beziehung zwischen `if-then-else` und einer `case`-Anweisung herstellt, die auf einem booleschen Ausdruck operiert.

3.5.3 Funktionsdeklaration

Wie bereits in Abschnitt 2.4.1 angesprochen, können Funktionsdeklarationen in Wertzuweisungen transformiert werden. Während in Haskell grundsätzlich auf die *monomorphism restriction* [30, 4.5.4] geachtet werden muß, kann man diese Einschränkung für die interne Repräsentation außer Acht lassen. Denn die Verifikation der Beweisschritte benötigt zunächst keinerlei Typinformation. Erst nach Abschluß wird das Ergebnis extern auf Typkonsistenz geprüft. Dafür werden diese Repräsentationen dann entsprechend aufbereitet.

Eine *Guard* ist eine boolescher Ausdruck, der bei der Deklaration auf die formalen Parameter folgt und durch „|“ von ihnen abgegrenzt ist. Sehen wir uns dazu die Deklaration von `filter` aus dem vorherigen Abschnitt an, bei der jetzt Guards verwendet werden.

```
filter p []              = []
filter p (x:xs) | p x    = x : filter p xs
                | True   =     filter p xs
```

Nur wenn eine Guard bei der Auswertung zu `True` evaluiert, kommt der hinter „=“ folgende Ausdruck zur Auswertung. Sonst wird nach einer anderen Alternative gesucht. Die Vorgehensweise ist dabei analog zum *Pattern Matching*. Damit spielt die Reihenfolge der Guards wieder eine entscheidende Rolle für das Programmverhalten.

Verwendet die Funktionsdeklaration Pattern in den formalen Argumenten oder hängt die Ausführung des Rumpfes von einer Guard ab, wird diese Information in eine `case`-Anweisung übertragen.

In Abbildung 3.2 werden die verschiedenen Fälle, die bei der Deklaration mit Pattern und Guards auftreten können, in einer einzigen Funktion zusammengefaßt, auf welche dann exemplarisch die Transformation angewendet wird.

`case`-Anweisung

Eine `case`-Anweisung wird in ihrem Aufbau nicht weiter verfeinert, d.h. die angegebenen Pattern und die zugehörigen Guards werden keiner weiteren Standardisierung unterworfen. Es wäre durchaus vorstellbar, etwa verschachtelte Pattern aufzulösen, so daß am Ende innerhalb einer `case`-Anweisung nur noch Pattern der Gestalt `_`, `x` und `C` $\mathtt{x}_1 \ldots \mathtt{x}_n$ $(n \geq 0)$ auftreten. Darauf wird aber verzichtet und stattdessen wertet das System `case`-Anweisungen ohne weitere Aufbereitung gemäß der Haskell-Semantik [30, 3.17] aus, was bedeutet:

- bearbeite Pattern der Reihe nach von oben nach unten
- gehe innerhalb eines Pattern von links nach rechts vor
- wenn Guard scheitert, versuche nächste Alternative, sonst verwende den hinter der Guard angegebenen Ausdruck

$$
\begin{array}{l}
\mathrm{f}\ pat_{1,1} \ldots pat_{1,n} \mid g_{1,1} = exp_{1,1} \ldots \mid g_{1,k} = exp_{1,k} \\
\mathrm{f}\ pat_{2,1} \ldots pat_{2,n} = exp_2 \\
\vdots \\
\mathrm{f}\ pat_{m,1} \ldots pat_{m,n} \mid g_m = exp_m \\
\qquad\qquad\qquad\qquad \Downarrow \\
\mathrm{f}\ x_1 \ldots x_n = \texttt{case}\ (x_1, \ldots, x_n)\ \texttt{of} \\
\qquad\qquad\qquad\qquad (pat_{1,1}, \ldots, pat_{1,n}) \mid g_{1,1} \rightarrow exp_{1,1} \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad (pat_{1,1}, \ldots, pat_{1,n}) \mid g_{1,k} \rightarrow exp_{1,k} \\
\qquad\qquad\qquad\qquad (pat_{2,1}, \ldots, pat_{2,n}) \rightarrow exp_2 \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad (pat_{m,1}, \ldots, pat_{m,n}) \mid g_m \rightarrow exp_m
\end{array}
$$

Abbildung 3.2: Transformation von Funktionsdeklarationen

Genauere Information zur weiteren Verfeinerung von `case`-Anweisungen findet sich unter [30, 3.17].

3.5.4 λ-Abstraktion

Haskell läßt auch in λ-Abstraktionen Pattern in den formalen Argumenten zu. Dabei ist aber zu bedenken, daß mit *refutable* Pattern [2.4.3] partielle Funktionen definiert werden. Damit besteht immer die Gefahr, sich einen Laufzeitfehler einzuhandeln, wenn das Pattern dann doch nicht mit dem konkreten Wert übereinstimmen sollte. Bevor jedoch die verwendeten Pattern mittels einer `case`-Anweisung aufgelöst werden, bringt man die λ-Abstraktionen zunächst auf eine einheitliche Form:

$$
\begin{array}{lcl}
\backslash x_1 x_2 \ldots x_n \rightarrow exp & \Rightarrow & \backslash x_1 \rightarrow \backslash x_2 \rightarrow \ldots \rightarrow \backslash x_n \rightarrow exp \\
\backslash x \rightarrow (\backslash y \rightarrow exp) & \Rightarrow & \backslash x \rightarrow \backslash y \rightarrow exp \\
\backslash pat_1 \ldots pat_n \rightarrow exp & \Rightarrow & \backslash x_1 \ldots x_n \rightarrow \texttt{case}\ (x_1, \ldots, x_n)\ \texttt{of} \\
 & & \qquad\qquad \{\ (pat_1, \ldots, pat_n) \rightarrow \mathrm{exp}\ \}
\end{array}
$$

Dabei wird die letzte Transformation nur dann ausgeführt, wenn mindestens eines der Pattern pat_i keine Variable ist.

3.5.5 Arithmetische Sequenz

Die Transformation der arithmetischen Sequenzen fällt sehr einfach aus, denn sie beschränkt sich darauf, eine entsprechene Standard-Funktion Haskells mit den geeigneten Argumenten aufzurufen. Insgesamt sind vier Fälle zu unterscheiden:

$$
\begin{array}{lcl}
[\mathrm{exp}_1\ \texttt{..}\ \mathrm{exp}_2] & \Rightarrow & \texttt{enumFromTo}\ \mathrm{exp}_1\ \mathrm{exp}_2 \\
[\mathrm{exp}_1, \mathrm{exp}_2\ \texttt{..}\ \mathrm{exp}_3] & \Rightarrow & \texttt{enumFromThenTo}\ \mathrm{exp}_1\ \mathrm{exp}_2\ \mathrm{exp}_3 \\
[\mathrm{exp}\texttt{..}] & \Rightarrow & \texttt{enumFrom}\ \mathrm{exp} \\
[\mathrm{exp}_1, \mathrm{exp}_2\ \texttt{..}] & \Rightarrow & \texttt{enumFromThen}\ \mathrm{exp}_1\ \mathrm{exp}_2
\end{array}
$$

Der wesentliche Vorteil besteht hier auch wieder darin, daß man es im weiteren Verlauf mit gewöhnlichen Funktionsapplikationen zu tun hat und nicht mit speziellen Konstrukten.

Kapitel 4

Ablauf der Gleichheitsprüfung im System

In diesem Kapitel wird der Algorithmus, der für die Verifikation verwendet wird, durch eine Reihe von Ableitungsregeln beschrieben. Zu Beginn wird jedoch gezeigt, wie das System lokale Deklarationen handhabt. Die interne Repräsentation, in der lokale und globale Deklarationen gleich behandelt werden, stellt in dieser Form zunächst kein korrektes Haskell-Programm mehr da. Erst bei der Ausgabe des Beweises, die am Ende dieses Kapitels in diesem Zusammenhang betrachtet wird, erfolgt eine Umsetzung in korrekten Haskell-Code. Was die Auswertung von Haskell-Ausdrücken betrifft, wird auf Vorsichtsmaßnahmen eingegangen, die für einen sicheren Zustand des Systems sorgen, selbst wenn eine Funktion behandelt wird, deren Auswertung nicht terminiert. Den weitaus größten Teil in diesem Kapitel nehmen die oben angesprochenen Ableitungsregeln in Anspruch, die in den einzelnen Phasen des Verifikationsprozesses zur Anwendung kommen.

4.1 Behandlung von `let`-Ausdrücken

Bei der Auflösung von `let`-Ausdrücken könnte man zunächst daran gehen, solche Funktions- und Wertdeklarationen, die nicht über Rekursion definiert werden, im zugehörigen Ausdruck zu ersetzen oder mit Hilfe von λ-Abstraktionen zu umschreiben. Auf diese Weise läßt sich

```
let { z=5; f x = x+x } in z + f 3
```

umformen

mittels Substitution `[5/z,(\x→x+x)/f]`: `5 + (\x→x+x) 3`
mittels λ-Abstraktion: `(\z→ \f→z + f 3) 5 (\x→x+x)`

Allerdings kann man mit diesen Methoden keine rekursiven Definitionen behandeln.[1] Aus diesem Grund werden die oben genannten Transformationen nicht durchgeführt, sondern man wählt von vornherein einen allgemeineren Ansatz, der auf alle Funktions- und Wertdeklarationen anwendbar ist. Man ist ja eigentlich nur daran interessiert, auf

[1] es sei denn man arbeitet mit Fixpunkt-Operatoren, was sich hier aber nicht als zweckmäßig erweist

den Wert einer Deklaration (z.B. einen Funktionsrumpf) zuzugreifen, wenn man innerhalb eines Ausdrucks auf deren Namen stößt. Daher erscheint es sinnvoll, alle lokalen Deklarationen zusammen mit den *toplevel*-Deklarationen zu verwalten. Somit kann man mit einem einheitlichen Mechanismus darauf zugreifen. Das Problem von Namenskonflikten wird dadurch ausgeschaltet, daß man anstelle des textuellen Namens intern einen Identifikator verwendet. So kann man in dem Programmausschnitt

```
x = 5::Int
f y = x + let x=y*y in x
```

das linke `x` in der Funktion `f` der Deklaration `x=5` zuordnen und das rechte `x` der lokalen Deklaration. Die beiden Deklarationen von `x` (`x=5, x=y*y`) können durch Identifikatoren unterschieden werden und verursachen dadurch bei einem Zugriff keine Konflikte.

Der wesentliche Unterschied zwischen lokalen und *toplevel*-Deklarationen ist der, daß lokale Deklarationen freie Variablen enthalten können. Das ist aber in dem hier gewählten Ansatz unproblematisch, da diese Variablen implizit bei der Verwendung einer solchen Deklaration gebunden werden. Diese Verwendung ist nämlich mit der textuellen Substitution des Namens gleichzusetzen und somit findet sich der eingesetzte Ausdruck in der Umgebung wieder, die durch den `let`-Ausdruck gegeben war.

Sehen wir uns dazu ein kleines Beispiel an. Die Verwendung von Identifikatoren soll dabei durch die Angabe eines Subskripts angedeutet werden.

```
times m x = m*x
multlist c xs
 = let times ys = map (*c) ys
       filtered = filter (>0) xs
   in  times filtered
```

$$\Downarrow$$

$\texttt{times}_{\texttt{top0}}$ `m x = m*x`
$\texttt{multlist}_{\texttt{top1}}$ `c xs =` $\texttt{times}_{\texttt{loc0}}$ $\texttt{filtered}_{\texttt{loc1}}$
$\texttt{times}_{\texttt{loc0}}$ `ys = map (*c) ys`
$\texttt{filtered}_{\texttt{loc1}}$ `= filter (>0) xs`

In $\texttt{times}_{\texttt{loc0}}$ tritt `c` als freie Variable auf, da die ursprüngliche Deklaration von `times ys` aus ihrem umgebenden Kontext genommen wurde. Das transformierte Programm, das oben angegeben wurde, stellt somit in dieser Form kein korrektes Haskell-Programm dar. Es handelt sich hierbei nur um eine interne Repräsentation. Erst bei der Ausgabe, die unter Abschnitt 4.5 beschrieben wird, wird diese Darstellung in ein korrektes Programm überführt.

Da $\texttt{times}_{\texttt{loc0}}$ nur in der dafür vorgesehenen Umgebung verwendet werden kann (nur innerhalb der entsprechenden Umgebung kann bei der obigen Transformation eine derartig abgeleitete Funktion eingefügt werden und nirgendwo anders), ist dort die Variable `c` an das formale Argument von $\texttt{multlist}_{\texttt{top1}}$ gebunden.

Lokale Deklarationen werden auf diese Weise genauso wie *toplevel*-Deklarationen behandelt, d.h. wenn auf eine Funktion oder einen Wert über den zugehörigen Identifikator zugegriffen wird, erhält man die entsprechende Information, indem die Menge der Deklarationen inspiziert wird.

4.2 Auswertung von Haskell-Ausdrücken

Das vorgestellte System unterstützt die Auswertung von Haskell-Ausdrücken [31, 33]. Immer wenn sich die Gleichheit nicht direkt oder über eine Regelanwendung zeigen läßt, werden die beiden beteiligten Ausdrücke unabhängig voneinander ausgewertet und die Ergebnisse verglichen. Die Auswertung erfolgt lazy, so daß unendliche Strukturen, wie etwa `ones = 1:ones`, kein Problem darstellen. Dennoch muß darauf geachtet werden, daß die Auswertung nicht unkontrolliert verläuft. Würde nämlich eine rekursive Funktion abgearbeitet, die nicht terminiert, würde das System in einem undefinierten Zustand geraten. Aus diesem Grund wird eine Schranke für die Auswertungstiefe vorgegeben. Sobald diese Schranke überschritten wird, bricht das System die Evaluierung ab. Natürlich besteht hierbei die Gefahr, die vollständige Auswertung eines Ausdrucks durch eine ungeeignete Schranke zu verhindern, andererseits muß irgendwann abgebrochen werden, um einen definierten Zustand des Systems zu gewährleisten.

Sehen wir uns dazu das client-server-Modell aus Abschnitt 2.4.3 an.

```
reqs  = client initial resps
resps = server reqs

client init ~(rp:rps) = init : client (next rp) rps
server (rq:rqs)       = process rq : server rqs

-- data Nat = N | S Nat
next x = S x
```

Das Symbol `{-==-}` wird bei der Eingabe eines Beweises dazu verwendet, die zu vergleichenden Ausdrücke voneinander abzugrenzen. Die genaue Angabe der Syntax, die das System voraussetzt, folgt im 6. Kapitel. Wenn das System die Gleichheit von

```
[x,S y,S z] {-==-} case reqs of { a:b:c:_ -> [a,b,c] }
```

nachweisen soll, unter der Annahme, daß `x,y` und `z` freie Variablen sind und keine passende Regel vorhanden ist, gelangt es über die Auswertung zu folgendem Ergebnis:

```
x : S y : [S z]
{-
   x = initial
   y = process initial
   z = process (next (case resps of { l4_2:l4_3 -> l4_2 }))
-}
case reqs of { l6_1:l6_2:l6_3:l0_0 -> l6_1:l6_2:[l6_3] }
```

Hier ist zu beachten, daß die lokalen Variablen `a`, `b` und `c`, die in der `case`-Anweisung aufgeführt sind, und alle formalen Funktionsparameter durch einen eindeutigen Namen ersetzt werden, wie etwa `l6_3` bzw. `l4_2`, um Namenskonflikten vorzubeugen. Zwischen den Kommentarklammern `{-` und `-}` stehen die Bindungen für die Variablen, die nötig sind, um den Übergang zwischen den beiden Ausdrücken durchführen zu können.

Es können sich im auszuwertenden Ausdruck durchaus freie Variablen oder Symbole befinden, wie etwa `initial` im obigen Beispiel, das im Programm selbst nicht definiert wurde.[2] Solange der Wert eines solchen Ausdrucks nicht explizit gebraucht wird, wird die Auswertung dadurch nicht behindert. Andernfalls muß mit der Evaluierung abgebrochen werden.

[2]Um diese Eingabe durch den Typ-Checker zu bringen, muß allerdings an anderer Stelle die entsprechende Typ-Information abgelegt werden [6.1.4].

4.3 Phasen der Gleichheitsprüfung

Der Versuch, die Gleichheit zweier (Teil-)Ausdrücke nachzuweisen, kann verschiedene Bearbeitungsschritte erfordern. Es sei hier nochmals darauf hingewiesen, daß man an der Verifikation von ER-Beweisen interessiert ist, d.h. daß höchstens eine Regelanwendung aus der Datenbasis pro Beweisschritt beteiligt sein soll. Diese Vorgabe wurde bei der Entwicklung des Algorithmus miteinbezogen, so daß es sich hierbei um ein sehr spezielles Verfahren handelt. Es wurde nicht darauf ausgelegt, automatisch Zwischenschritte in den Beweis einzufügen und ihn zu vervollständigen, wenn der Benutzer nicht nachvollziehbare Transformationsschritte durchführt. Das System liefert in diesem Fall aber geeignete Information, anhand derer der Benutzer den Beweis feiner strukturieren kann.

Die einzelnen Phasen, die während der Verifikation durchlaufen werden können, lassen sich in Abbildung 4.1 erkennen.

Der Ausgangspunkt für die Beweisprüfung ist eine *Menge von Teilproblemen.* In unserem Fall sind das die einzelnen Transformationsschritte, deren Gültigkeit gezeigt werden soll. Besteht der Beweis aus den Umformungen

$$\mathtt{exp}_1 \rightarrow \mathtt{exp}_2 \rightarrow \mathtt{exp}_3 \rightarrow \ldots \rightarrow \mathtt{exp}_{n-1} \rightarrow \mathtt{exp}_n,$$

so müssen die Gleichungen

$$\mathtt{exp}_1 \equiv \mathtt{exp}_2\texttt{,}\ \ \mathtt{exp}_2 \equiv \mathtt{exp}_3\texttt{,}\ \ \ldots\ \ \texttt{,}\mathtt{exp}_{n-1} \equiv \mathtt{exp}_n$$

nachgewiesen werden. Bevor eine Gleichung näher untersucht wird, werden die beiden Seiten zunächst normalisiert. In der *Normalisierung I* werden die Namen von lokalen Deklarationen durch den zugehörigen Rumpf ersetzt. Ferner werden λ-Abstraktionen beseitigt und `case`-Anweisungen eliminiert, sofern dies möglich ist. Diese Normalisierung beschränkt sich im wesentlichen auf den Teilausdruck, der bei einer Funktionsapplikation ganz links steht, d.h. es werden zunächst keine Transformationen innerhalb von Funktionsargumenten durchgeführt. Kommt es zu einem Vergleich zwischen einer λ-Abstraktion und einer anderen Funktion, wird versucht, die Überprüfung auf den Rumpf der Abstraktion zurückzuführen. Dazu werden die beiden Funktionen auf ein *Pseudo*-Argument angewendet [4.3.4]. Die weitere Normalisierung bewirkt, daß mit der anschließenden β-Reduktion der Vergleich auf dem Funktionsrumpf weitergeführt wird. Während die erstgenannten Normalisierungsschritte unabhängig voneinander verlaufen, sind bei der Einführung eines *Pseudo*-Arguments beide Ausdrücke beteiligt.

Nachdem die Seiten der Gleichung in Normalform gebracht wurden, sind prinzipiell zwei Fälle zu unterscheiden:

- Die Ausdrücke sind so einfach aufgebaut, daß sie direkt verglichen werden können, etwa $\mathtt{5} \equiv \mathtt{x}$ (`x` (noch) freie Variable) oder $\texttt{Just 0} \equiv \texttt{Nothing}$.
- Der Vergleich der beiden Ausdrücke läßt sich auf eine Menge von Teilproblemen zurückführen. So kann etwa der Ausdruck $\mathtt{f}\ \mathtt{exp}_1\ \mathtt{exp}_2 \equiv \mathtt{g}\ \mathtt{exp}_3$ auf die Probleme $\mathtt{f}\ \mathtt{exp}_1 \equiv \mathtt{g}$ und $\mathtt{exp}_2 \equiv \mathtt{exp}_3$ reduziert werden. Sollte sich diese Zerlegung als üngünstig herausstellen, kann über das *Backtracking* der ursprüngliche Ausdruck erneut unter einer anderen Strategie betrachtet erden.

Im letzten Fall wird bei der *Zerlegung* das ursprüngliche Problem durch eine Menge neuer Gleichungen ersetzt, die nachgewiesen werden müssen. Die Anzahl der zu bearbeitenden Teilprobleme nimmt also nicht ab. Nur im ersten Fall, wenn die beiden Seiten übereinstimmen, verringert sich die Anzahl der Probleme (*Elimination*).

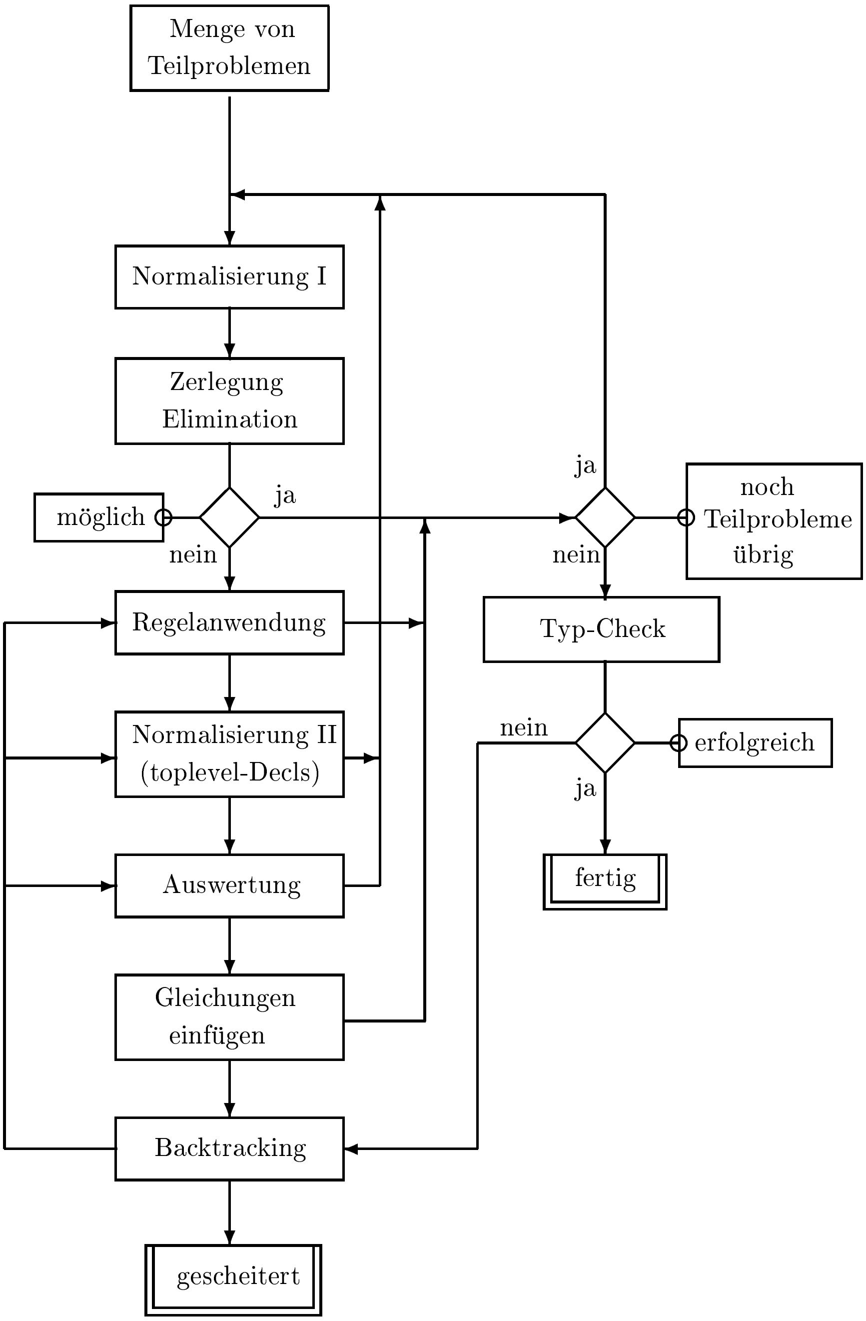


Abbildung 4.1: Ablauf-Schema der Gleichheitsprüfung

`f x = let n=x in n`	$\underbrace{\underbrace{\texttt{f 2}}_{\texttt{n}}}_{2} \equiv \underbrace{\underbrace{\texttt{f 3}}_{\texttt{n}}}_{3}$	die Namensgleichheit von `n` reicht nicht aus

Abbildung 4.2: Freie Variablen in lokalen Deklarationen

Wenn die vorliegende Gleichung $\texttt{exp}_1 \equiv \texttt{exp}_2$ nicht gezeigt werden kann, wird in der Datenbasis nach einer geeigneten Regel gesucht. Dazu geht man auf folgende Teilprobleme über, wobei `R` eine beliebige Regel aus der Datenbasis darstellt:

$\texttt{exp}_1 \equiv$ *linke Seite von Regel* `R`
$\texttt{exp}_2 \equiv$ *rechte Seite von Regel* `R`
Gilt Prämisse von Regel `R`

Daß bei der Überprüfung, ob die Regel `R` mit dem ursprünglichen Problem $\texttt{exp}_1 \equiv \texttt{exp}_2$ in Übereinstimmung gebracht werden kann, keine weitere Regelanwendung beteiligt sein darf, ergibt sich aus der Einschränkung in Abschnitt 3.4.1. Dadurch ist sichergestellt, daß die ungerichteten Regeln keine Zyklen in der Herleitung verursachen. Auf diese Weise lassen sich die in Frage kommenden Regeln sukzessive auf ihre Tauglichkeit hin überprüfen. Gibt es eine geeignete Regel, dann wird mit der Verifikation der verbleibenden Teilprobleme fortgefahren.

Wenn keine Regel paßt, werden in der *Normalisierung II* die Namen von *toplevel*-Deklarationen ersetzt. In der *Normalisierung I* werden diese Namen noch nicht berücksichtigt, da es weniger Aufwand bedeutet, die Namen von *toplevel*-Funktionen zu vergleichen als deren Rümpfe. Dies gilt u.a. auch bei der Suche nach einer passenden Regel. Wenn sich also zwei Namen jeweils auf eine *toplevel*-Deklaration beziehen und beide Namen gleich sind, dann stimmen auch die an sie gebundenen Ausdrücke überein. Bei lokalen Namen muß dies nicht gelten. Selbst wenn die Namen dem gleichen `let`-Ausdruck entstammen (und nur dann ist ein Vergleich auf Namensebene erlaubt), ist daran zu denken, daß eine lokale Deklaration freie Variablen enthalten kann, die dann zusätzlich berücksichtigt werden müssen. Abbildung 4.2 soll diesen Fall verdeutlichen. Bei *toplevel*-Deklarationen kann dies nicht auftreten. Aus diesem Grund werden lokale Namen sofort ersetzt und die anderen erst nach erfolgloser Suche nach einer geeigneten Regel.

Deklarationsnamen dürfen aber bei der *Normalisierung I* und *II* nicht beliebig oft ersetzt werden. Die Bearbeitung eines Ausdruckes der Form

```
let f=f in f
```

würde sonst nicht terminieren. Daher werden Namen von Deklarationen nur *einmal* innerhalb eines Ausdrucks berücksichtigt und dann von der weiteren Normalisierung ausgeschlossen. Damit ist auch für rekursive Deklarationen – lokale und *toplevel* – eine sichere Behandlung möglich. Da die Namen von Deklarationen nur dann ersetzt werden, wenn sie bei der Normalisierung (I/II) des Ausdrucks nicht schon verwendet wurden, kann es vorkommen, daß die *Normalisierung II* keine Veränderung bringt. In diesem Fall wird eine kontrollierte *Auswertung* der beiden Ausdrücke durchgeführt. Es gelten die in Abschnitt 4.2 beschriebenen Einschränkungen. Nach der Auswertung werden die beiden Ergebnisse miteinander verglichen. Wie bei der erfolgreichen Ersetzung von *toplevel*-Funktionen kehrt man wieder zur *Normalisierung I* zurück und durchläuft von dort aus alle weiteren Phasen.

Wenn die Auswertung aber zu keiner Veränderung der beiden Ausdrücke führt, da sich z.B. freie Variablen oder Symbole in ihnen befinden, wird eine neue Beweisbedingung hinzugenommen. Es wird einfach angenommen, daß die beiden Ausdrücke gleich sind. Diese spezielle Gleichung, unter der der Beweis weitergeführt wird, dient bei der Beweisausgabe dazu, dem Benutzer die Konfliktstellen in seinem Beweis zu zeigen. Nur für diejenigen Konfliktpaare werden derartige Zusatzbedingungen eingefügt, bei denen eine vernünftige Möglichkeit besteht, die Gleichheit später nachzuweisen. So macht es etwa wenig Sinn $3 \equiv 4$ anzunehmen. In solchen Fällen wird nach einem anderen Lösungsweg gesucht. Damit beschäftigt sich die *Backtracking*-Komponente. Dazu werden gezielt Entscheidungen im vorangegangenen Verifikationsprozeß rückgängig gemacht. Stellt sich heraus, daß die Zerlegung in weitere Teilprobleme keinen Erfolg brachte, wird stattdessen im nächsten Versuch nach einer passenden Regel gesucht. Wenn sich die Wahl einer Regel als falsch herausstellt, kann später eine andere versucht oder können *toplevel*-Funktionen ersetzt werden. Auf diese Weise werden systematisch verschiedene Lösungsvarianten ausprobiert. Erst wenn keine Rücksetzpunkte mehr vorhanden sind, ist die Verifikation des Beweises endgültig gescheitert.

Wenn sich alle Teilprobleme erfolgreich bearbeiten lassen und keine Gleichung mehr nachzuweisen ist, muß das Ergebnis noch auf Typkonsistenz überprüft werden. Damit wird u.a. sichergestellt, daß (freie) Variablen an Ausdrücke mit entsprechendem Typ gebunden wurden. Während der Verifikation des Beweises berücksichtigt das System keine Typen. Es wird lediglich die Typinformation der freien Variablen und derjenigen Variablen verwaltet, die in den Regeln auftauchen. Daraus wird zusammen mit den angegebenen Transformationsschritten und den Voraussetzungen, die sich während der Verifikation ergaben, eine Datei erstellt, die von einem externen Typ-Überprüfer auf Typkonsistenz hin untersucht wird. Im Fehlerfall wird mittels *Backtracking* nach einer anderen Lösung gesucht, sonst kann ein typkorrekter, annotierter Beweis ausgegeben werden. Auf dieser Lösung aufbauend kann der Benutzer wiederum mittels *Backtracking* gezielt Variablenbindungen und Bedingungen – u.a. auch die vom System künstlich eingefügten Gleichungen – zurücknehmen lassen und nach anderen Lösungsvarianten suchen.

4.3.1 Notation

Zur Beschreibung des Algorithmus wird die Funktion `Verify` verwendet, die auf einer Liste von (Teil-)Problemen arbeitet. Ein Zustand während der Verifikation läßt sich damit folgendermaßen beschreiben:

$$\mathtt{Verify}^{\mathtt{M,B}}_{\mathtt{E,C}}(\mathtt{problems})$$

In `E` (Environment) werden die Bindungen von freien Variablen gesammelt und `C` (Condition) enthält die Bedingungen, die eingehalten werden müssen. Für die *Backtracking*-Komponente werden in `B` die noch abzuarbeitenden Rücksetzpunkte verwaltet. Mit `M` werden die Modi `UseRule`, `NoRule` und `TestRule` unterschieden, womit man dann z.B. festlegen kann, ob Regeln verwendet werden dürfen oder nicht.

Im folgenden werden die einzelnen Probleme aufgeführt, die in der Liste `problems` erscheinen können. Die Liste `D`, die bei fast allen angegeben ist, beschreibt die Rücksetzpunkte für diejenigen gebundenen Variablen, die während der Verifikation verwendet wurden. Immer wenn der Wert einer gebundenen Variable benutzt wird, wird die Information, wo und wie man diese Bindung rückgängig machen kann, in die Liste `D` eingefügt (vgl. dazu die Regeln (R3), (R5) und (R22) in den folgenden Abschnitten).

- `Step(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: bearbeite den nächsten Beweisschritt
- $\texttt{Norm}_\texttt{D}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: führe *Normalisierung I* durch
- $\texttt{Proof}_\texttt{D}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: zeige Gleichheit von exp_1 und exp_2
- $\texttt{Rule}_\texttt{D}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: suche in der Datenbasis nach einer Regel, die die Gleichheit von exp_1 und exp_2 belegt.
- $\texttt{NormII}_\texttt{D}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: ersetze *toplevel*-Deklarationen (*Normalisierung II*)
- $\texttt{Evaluate}_\texttt{D}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: werte die beiden Ausdrücke aus
- $\texttt{Insert}_\texttt{D}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: versuche die Bedingung $\text{exp}_1 \equiv \text{exp}_2$ als Prämisse des Beweises aufzunehmen, wenn sich die Gleichheit von exp_1 und exp_2 nicht zeigen läßt.
- $\texttt{Test}_\texttt{D}$`(c)`: überprüfe die Bedingung `c` auf ihre Gültigkeit
- $\texttt{Pseudo}_\texttt{D}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: wenn exp_1 bzw. exp_2 eine λ-Abstraktion repräsentiert, dann versuche den Vergleich auf den Rumpf der Abstraktion zurückzuführen, indem beide Ausdrücke auf ein gemeinsames „virtuelles" Argument angewendet werden.
- $\texttt{Back}^{\texttt{M,D}}_{L,\texttt{E,C}}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)`: führe *Backtracking* in Abhängigkeit von L durch, wobei L eines der Attribute `Rule, NextRule, NormII, Evaluate` und `Insert` ist. Die Phase, in der sich der Verifikationsprozeß beim Nachweis der Gleichheit von exp_1 und exp_2 befindet, legt L fest.
- $\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}$ `problems`: bestimme den nächsten Aufsetzpunkt, an den zurückgekehrt werden soll.
- `BackTo (R,BS)`: kehre zu Aufsetzpunkt `R` zurück und behandle die restlichen *Backtrack*-Punkte `BS` ggfs. zu einem späteren Zeitpunkt.

Auf der Liste der Probleme ist noch eine Hilfsfunktion `Clear` definiert, die alle Elemente am Anfang der Liste entfernt, die ungleich $\texttt{Back}^{\texttt{M,D}}_{L,\texttt{E,C}}$`(`$\texttt{exp}_1$`,`$\texttt{exp}_2$`)` sind. Sie hat also die Gestalt:

```
Clear [] = []
Clear ls@(Back^{M,D}_{L,E,C}(_,_):_) = ls
Clear (_:ls) = Clear ls
```

Diese Funktion wird dazu verwendet, einen möglichen Aufsetzpunkt während des *Backtrackings* zu bestimmen.

Handelt es sich bei dem Ausdruck x um den Namen einer Deklaration, liefert `Value(`x`)` den zugehörigen Rumpf, aber nur dann, wenn diese Deklaration auf dem jeweiligen Bearbeitungspfad noch nicht verwendet wurde. In allen anderen Fällen ist `Value(`x`)` mit x gleichzusetzen.

Die Auswertung eines Ausdrucks *exp* wird durch `Eval(`*exp*`)` ausgedrückt. Für `Eval(`*exp*`)` gelten die unter Anschnitt 4.2 beschriebenen Vorsichtsmaßnahmen, um Nichttermination zu vermeiden.

`DEval(exp)` gibt die Liste der gebundenen Variablen an, die während der Auswertung von `Eval(exp)` ersetzt wurden.

Der Rücksetzpunkt `R` für eine Variablenbindung oder für eine Bedingung, die von einer Regelanwendung eingeführt wird, wird in der Form `(x=`$\texttt{exp}_\texttt{x}$`)<R>` bzw. `c<R>` angegeben.

Das Prädikat `Free(`f`)` zeigt an, ob f eine freie Variable ist.

4.3.2 Grundlegende Ableitungsregeln

Bevor in den nächsten Abschnitten den einzelnen Phasen der Gleichheitsprüfung Ableitungsregeln zugeordnet werden, sind hier zunächst drei Regeln aufgeführt, die unabhängig von den Phasen zur Anwendung kommen.

- Der Rücksetzpunkt der am Kopf der Liste steht, wird nicht mehr benötigt, da die Gleichheit von $\exp_1$ und $\exp_2$ bereits gezeigt werden konnte.

$$\text{(R1)}\quad \frac{\mathtt{Verify}^{\mathtt{M,B}}_{\mathtt{E,C}}(\mathtt{Back}^{\mathtt{M',D'}}_{L,\mathtt{E',C'}}(\mathtt{exp}_1,\mathtt{exp}_2)\mathtt{:problems})}{\mathtt{Verify}^{\mathtt{M,B}}_{\mathtt{E,C}}(\mathtt{problems})}$$

- Das Problem `Step` leitet die Bearbeitung des nächsten Beweisschrittes ein. Eine Folge von `Step`-Problemen entspricht einem Beweis, den der Benutzer überprüfen läßt. Da in jedem Schritt eine Regel angewendet werden darf, setzt man zu Beginn den Modus auf `UseRule` zurück. Die gesammelte Information in `B, E` und `C` wird von den vorherigen Schritten übernommen.

$$\text{(R2)}\quad \frac{\mathtt{Verify}^{\mathtt{M,B}}_{\mathtt{E,C}}(\mathtt{Step}\,(\mathtt{exp}_1,\mathtt{exp}_2)\mathtt{:problems})}{\mathtt{Verify}^{\mathtt{UseRule,B}}_{\mathtt{E,C}}(\mathtt{Norm}_{[]}(\mathtt{exp}_1,\mathtt{exp}_2)\mathtt{:problems})}$$

- Wenn der gegebene Beweis erfolgreich verifiziert werden kann, liegt die Situation

$$\mathtt{(Succ)}\qquad \mathtt{Verify}^{\mathtt{M,B}}_{\mathtt{E,C}}\mathtt{[]}$$

  vor, d.h. die Liste der noch zu bearbeitenden Probleme ist leer. In `E` finden sich die abgeleiteten Variablenbindungen und in `C` die Voraussetzungen, die zusätzlich erfüllt sein müssen.

4.3.3 Normalisierung I

Bei der Bearbeitung des Problems $\mathtt{Norm}_{\mathtt{D}}(\mathtt{Exp}_1,\mathtt{Exp}_2)$ werden die beiden Ausdrücke auf eine einheitliche Form gebracht. Bei der *Normalisierung I* wird immer der Kopf eines Ausdrucks betrachtet. Unter dem Kopf eines Ausdrucks `exp` von der Form $\mathtt{f}\ \mathtt{exp}_1 \ldots \mathtt{exp}_n$ $(n \geq 0)$ ist dabei der Teilausdruck `f` zu verstehen. Modifikationen an `exp` schließen immer `f` mit ein, d.h. es wird nicht innerhalb eines der $\mathtt{exp}_i$ operiert.

Die Normalisierung von $\mathtt{Exp}_1$ und $\mathtt{Exp}_2$ erfolgt zunächst unabhängig voneinander nach derselben Strategie. Der kürzeren Darstellung wegen werden daher nur Normalisierungsschritte für $\mathtt{Exp}_1\mathtt{=f}\ \mathtt{exp}_1 \ldots \mathtt{exp}_n$ $(n \geq 0)$ angegeben. Die entsprechenden Schritte für $\mathtt{Exp}_2$ ergeben sich in analoger Weise, indem man die Rollen von $\mathtt{Exp}_1$ und $\mathtt{Exp}_2$ vertauscht.

- $\texttt{f}$ ist eine gebundene Variable mit Rücksetzpunkt $\texttt{R}$, d.h. $\texttt{(f=exp}_\texttt{f}\texttt{)<R>}$ ist in $\texttt{E}$

$$(\mathrm{R3})\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_{\texttt{(R:D)}}\texttt{(exp}_\texttt{f}\ \texttt{exp}_1 \ldots \texttt{exp}_n\texttt{,Exp}_2\texttt{):problems)}}$$

Der Rücksetzpunkt $\texttt{R}$ zeigt an, wo die entsprechende Variablenbindung aufgelöst werden kann, wenn es im weiteren Verlauf zu Konflikten kommen sollte.

- $\texttt{f}$ ist der Name einer lokalen Deklaration $\wedge$ $\texttt{Value(f)} \neq \texttt{f}$

$$(\mathrm{R4})\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(Value(f)}\ \texttt{exp}_1 \ldots \texttt{exp}_n\texttt{,Exp}_2\texttt{):problems)}}$$

Hier ist zu erwähnen, daß der Name $\texttt{f}$ einer lokalen Deklaration im Verlauf der Normalisierung von $\texttt{Value(f)}\ \texttt{exp}_1 \ldots \texttt{exp}_n$ für die Ersetzung gesperrt wird, um die Termination des Verfahrens zu gewährleisten. Ziel dieses Normalisierungsschrittes ist es, am Kopf des resultierenden Ausdrucks etwas zu erhalten, das nicht weiter ersetzt werden kann bzw. darf. Namen von *toplevel*-Deklarationen werden hier nicht berücksichtigt, weil sie später noch bei der Suche nach Regeln benutzt werden können.

- $\texttt{f = case exp of}\ \{\ldots\}\ \wedge\ \texttt{Eval(f)} \neq \texttt{f}$

$$(\mathrm{R5})\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_{\texttt{(DEval(f)}++\texttt{D)}}\texttt{(Eval(f)}\ \texttt{exp}_1 \ldots \texttt{exp}_n\texttt{,Exp}_2\texttt{):problems)}}$$

Es wird versucht, `case`-Anweisungen zu eliminieren, um dadurch die Komplexität der Ausdrücke zu verringern. Es kann allerdings passieren, daß `Eval(f)` keinen Effekt zeigt, z.B. wenn `exp` eine freie Variable ist. Dann muß mit dem unveränderten Ausdruck weitergearbeitet werden.

- $\texttt{f = \textbackslash x}\rightarrow\texttt{exp}_x$ $(n \geq 1)$

$$(\mathrm{R6})\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(exp}_x\texttt{[exp}_1\texttt{/x]}\ \texttt{exp}_2 \ldots \texttt{exp}_n\texttt{,Exp}_2\texttt{):problems)}}$$

In diesem Fall wird eine Funktionsapplikation, an deren Kopf eine λ-Abstraktion steht, durch β-Reduktion vereinfacht.

- auf $\texttt{Exp}_1$ und $\texttt{Exp}_2$ paßt keine der Regeln (R3)-(R6)

$$(\mathrm{R7})\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Pseudo}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}$$

Es werden in gewissen Fällen sog. *Pseudo*-Argumente eingefügt, was im nächsten Abschnitt ausführlich behandelt wird.

4.3.4 *Pseudo*-Argument

Hinter einem *Pseudo*-Argument verbirgt sich ein Symbol, das für einen beliebigen Wert steht. Es simuliert gewissermaßen ein fehlendes Funktionsargument. Damit ein solches Argument eingefügt wird, müssen die beiden Ausdrücke in $\texttt{Pseudo}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{)}$ folgende Eigenschaften erfüllen:

- $\texttt{Exp}_1\texttt{=(\textbackslash x}\rightarrow\texttt{exp)}$
- $\texttt{Exp}_2$ ist nicht der Name einer Deklaration bzw. freien Variablen

Die Rollen von $\texttt{Exp}_1$ und $\texttt{Exp}_2$ dürfen dabei natürlich vertauscht werden. Wenn man diese Voraussetzung mit dem Prädikat $\texttt{InsertPseudo(Exp}_1\texttt{,Exp}_2\texttt{)}$ beschreibt, läßt sich folgende Normalisierungsregel angeben:

- $\texttt{InsertPseudo(Exp}_1\texttt{,Exp}_2\texttt{)}$

  $$\text{(R8)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Pseudo}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_\texttt{D}\texttt{(Exp}_1\ \textit{pseudo}\ \texttt{,Exp}_2\ \textit{pseudo}\texttt{):problems)}}$$

  Bei pseudo handelt es sich um ein neues Symbol, das die Stelle des fehlenden Arguments einnimmt.

- sonst

  $$\text{(R9)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Pseudo}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}$$

Die *Pseudo*-Argumente, die in Regel (R8) vergeben werden, sind durch einen Identifikator eindeutig bestimmt, d.h. sie sind alle unterscheidbar und es wird angenommen, daß ein *Pseudo*-Argument nur mit sich selbst übereinstimmt und mit keinem anderen.

Pseudo-Argumente werden durch β-Reduktion (R6) in den Funktionsrumpf transferiert und der Nachweis der Gleichheit zweier Funktionen wird somit intuitiv auf ihren Rümpfen weitergeführt. Es wird also versucht, die Gleichheit *extensional* nachzuweisen, d.h. indem man die Funktionen unter Angabe aller ihrer Argumente auswertet und vergleicht.

Im Gegensatz zu freien Variablen können an *Pseudo*-Argumente keine Ausdrücke gebunden werden. Es handelt sich – wie bereits oben erwähnt – um Symbole, die einen nicht näher definierten, konstanten Wert haben. Da ein *Pseudo*-Argument nur ein temporäres Hilfsmittel darstellt, darf es in der Beweisausgabe nicht erscheinen. Sollte dies der Fall sein, gilt der Lösungsweg als gescheitert. Diese Restriktion liegt darin begründet, daß es sehr schwierig ist, dem Benutzer diese Information sinnvoll mitzuteilen, vor allem wenn die betrachteten Ausdrücke kompliziert und damit auch unübersichtlich sind. Eine Aussage wie „Das *dritte* Argument der Funktion f an der Stelle l im Ausdruck E muß die Eigenschaft P erfüllen“ trägt sicherlich mehr zur Verwirrung als zum Nutzen bei.

4.3.5 Zerlegung und Elimination

Bei der Bearbeitung von $\texttt{Proof}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{)}$ wird versucht, das gegebene Problem auf kleinere Teilprobleme zurückzuführen oder ggfs. triviale bzw. elementare Fälle zu beseitigen, so daß damit die von `Verify` zu bearbeitende Liste verkürzt wird.

Im weiteren gelte nun $\texttt{Exp}_1 = f\, exp_1^f \ldots exp_n^f$ und $\texttt{Exp}_2 = g\, exp_1^g \ldots exp_m^g$ $(n, m \geq 0)$. Es wird o.B.d.A immer angenommen, daß $n \leq m$ gilt, um sich die Behandlung symmetrischer Fälle zu ersparen. Zur Verankerung eines Rücksetzpunktes wird die Funktion `backto` verwendet. Wenn der Zustand

$$\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}$$

gegeben ist, dann ist `backto` definiert als

$$\begin{array}{lll} \multicolumn{3}{l}{\texttt{backto = case M of}} \\ \quad\texttt{TestRule} & \texttt{->} & \texttt{problems} \\ \quad\texttt{NoRule} & \texttt{->} & \texttt{Back}^{\texttt{M,D}}_{\texttt{NormII,E,C}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems} \\ \quad\texttt{UseRule} & \texttt{->} & \texttt{Back}^{\texttt{M,D}}_{\texttt{Rule,E,C}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems} \end{array}$$

Im Modus `TestRule` wird also an dieser Stelle kein neuer Rücksetzpunkt gesetzt. Wenn die Suche nach Regeln verboten ist, wird sofort zur Ersetzung von *toplevel*-Deklarationen übergegangen.
Damit können nun folgende Ableitungsregeln festgehalten werden:

- $n = m \;\wedge\; m > 0$:

$$\text{(R10)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\begin{array}{c}\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_\texttt{D}\texttt{(}f, g\texttt{):} \\ \texttt{Norm}_\texttt{D}\texttt{(}exp_1^f, exp_1^g\texttt{):}\ldots\texttt{:Norm}_\texttt{D}\texttt{(}exp_n^f, exp_n^g\texttt{):backto)}\end{array}}$$

- $\texttt{Free(}f\texttt{)} \wedge n = 0 \;\wedge f$ ist nicht in $\texttt{Exp}_2$ enthalten (*occur check*):

$$\text{(R11)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{(f=\texttt{Exp}_2)<\texttt{Clear(backto)}>):\texttt{E,C}}\texttt{(problems)}}$$

  Durch `Clear(backto)` ist der Rücksetzpunkt festgelegt, mit dem man später die Variablenbindung von f zurücknehmen kann.

- $(\texttt{Free(}f\texttt{)} \wedge n = 0) \wedge \neg(\texttt{Free(}g\texttt{)} \wedge m = 0) \wedge f$ *ist* in $\texttt{Exp}_2$ enthalten:

$$\text{(R12)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}\ \texttt{(Clear(backto))}}$$

  Der *occur check* ist an dieser Stelle gescheitert, damit kann die Variable f nicht an $\texttt{Exp}_2$ gebunden werden.

- $(\texttt{Free}(f) \vee \texttt{Free}(g)) \ \wedge \ n > 0$:

$$\text{(R13)} \quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\begin{array}{c}\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_{\texttt{D}}\texttt{(}f, g\, exp^g_1 \ldots exp^g_{m-n}\texttt{):}\\ \texttt{Norm}_{\texttt{D}}\texttt{(}exp^f_1, exp^g_{m-n+1}\texttt{):}\ldots\texttt{:Norm}_{\texttt{D}}\texttt{(}exp^f_n, exp^g_m\texttt{):backto)}\end{array}}$$

- $n = m = 0 \ \wedge \ f \equiv g$:

$$\text{(R14)} \quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(problems)}}$$

  Dabei handelt es sich bei f und g um elementare Objekte, wie etwa Symbole oder Konstanten, die nicht mehr weiter zerlegbar sind und deren Übereinstimmung direkt getestet werden kann.

  Wenn f und g beides `case`-Anweisungen sind (also zeigte die Regel (R5) keine Wirkung), dann wird überprüft, ob beide Anweisungen – bis auf die Namen von gebundenen Variablen – syntaktisch übereinstimmen. So sind die beiden Anweisungen

```
case x of { Just y -> y; Nothing -> x}   und
case x of { Just x -> x; Nothing -> x}
```

  in diesem Sinne gleich. Der semantisch äquivalente Ausdruck

```
case x of { Just z -> z; _ -> x}
```

  unterscheidet sich allerdings bei diesem einfachen Prüfungsverfahren von den beiden vorher genannten.

- sonst

$$\text{(R15)} \quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Proof}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}\texttt{(Clear(backto))}}$$

  Damit wird das Problem an die *Backtrack*-Komponente weitergegeben.

Die nachfolgenden Komponenten unterscheiden sich von den beiden bis jetzt behandelten dadurch, daß sie nur über den *Backtracking*-Mechanismus aktiviert werden.

4.3.6 Regelanwendung

Wenn die Gleichheit zweier (Teil-)Ausdrücke nicht durch die Zerlegung in ihre Einzelteile nachgewiesen werden kann, dann wird in der Datenbasis nach einer geeigneten Regel gesucht, auf die sich das Problem zurückführen läßt. Die vom Benutzer angegebenen Regeln werden dabei nicht als gerichtet aufgefaßt, d.h. es hat keinen Einfluß auf die Lösbarkeit, ob in der Regelmenge `l≡r` oder `r≡l` steht. Die in Frage kommenden Regeln werden der Reihe nach auf ihre Anwendbarkeit hin getestet.

Betrachtet wird jetzt das Problem $\texttt{Rule}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{)}$, wenn eine Liste `R` von Regeln zur Verfügung steht, in der nach einer passenden gesucht werden soll. Ein Element aus `R` hat dabei die Gestalt $\texttt{c}\Rightarrow\texttt{l}\equiv\texttt{r}$, was bedeuten soll, daß die Regel $\texttt{l}\equiv\texttt{r}$ nur unter der Bedingung `c` gültig ist. Falls anstelle von `c` die boolesche Konstante `True` steht, kann man auch einfach von der Gleichung $\texttt{l}\equiv\texttt{r}$ sprechen.

In der Liste `RS` soll nun für jede Regel $\texttt{c}\Rightarrow\texttt{l}\equiv\texttt{r}$ aus `R` auch das symmetrische Gegenstück $\texttt{c}\Rightarrow\texttt{r}\equiv\texttt{l}$ enthalten sein, damit man die Richtung der Regel im weiteren Verlauf vernachlässigen kann. Da man höchstens *eine* Regelanwendung pro Beweisschritt zuläßt [3.4.1], stellt sich das Problem von Zyklen bei der Suche nach geeigneten Regeln nicht.

Wenn nun $\texttt{c}\Rightarrow\texttt{l}\equiv\texttt{r}$ das erste Element der Liste `RS` ist und `TRS` den Rest umfaßt, läßt sich das Vorgehen wie folgt beschreiben:

$$\text{(R16)}\quad \frac{\texttt{Verify}^{\texttt{UseRule,B}}_{\texttt{E,C}}\texttt{(Rule}_\texttt{D}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\begin{array}{c}\texttt{Verify}^{\texttt{TestRule,B}}_{\texttt{E,C}}\texttt{(Norm}_{[]}\texttt{(Exp}_1\texttt{,l):Norm}_{[]}\texttt{(Exp}_2\texttt{,r):Test}_{[]}\texttt{(c):}\\ \texttt{Back}^{\texttt{UseRule,D}}_{\texttt{NextRule(TRS),E,C}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}\end{array}}$$

Als erstes ist festzustellen, daß nach Regeln nur dann gesucht wird, wenn dies explizit mit `UseRule` erlaubt wurde. Damit wird verhindert, daß zwei Regeln innerhalb eines Beweisschrittes verwendet werden, was nach Vorgabe [3.4.1] ausgeschlossen werden soll.

Der Modus `TestRule` zeigt an, daß zunächst die beiden Seiten einer Regel mit den Ausdrücken $\texttt{Exp}_1$ und $\texttt{Exp}_2$ verglichen werden, um die Anwendbarkeit zu entscheiden. Damit für den Benutzer die Anwendung einer Regel transparent bleibt und der Aufwand nicht zu groß wird, muß die Übereinstimmung allein mit den Mitteln der *Normalisierung I* und der *Zerlegung und Elimination* nachgewiesen werden. Weiterführende Methoden, wie etwa Auswertung, sind nicht zugelassen.

Sollte die Regel prinzipiell passen, was sich dadurch ausdrückt, daß man zum Teilproblem `Test(c)` vorgestoßen ist, muß noch die Gültigkeit ihrer Prämisse nachgewiesen werden. Erst dann ist die Regelanwendung erfolgreich abgeschlossen. Nun kann es sein, daß die Bedingung nicht vollständig aufgelöst werden kann (freie Variablen) und nicht sicher ist, ob sie nun gilt oder nicht. In einem solchen Fall wird die Entscheidung auf einen später Zeitpunkt verschoben, z.B. wenn eine entsprechende freie Variable gebunden wird, und zunächst davon ausgegangen, daß sie wahr ist. Damit lassen sich folgende Regeln ableiten:

- Die Gültigkeit der Bedingung `c` kann gezeigt werden, d.h. $\texttt{Eval(c)}\equiv\texttt{True}$.

  $$\text{(R17)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Test}_\texttt{D}\texttt{(c):problems)}}{\texttt{Verify}^{\texttt{NoRules,B}}_{\texttt{E,C}}\texttt{(problems)}}$$

  Für diesen Beweisschritt wurde die zugebilligte Regelanwendung durchgeführt. Daher wird im weiteren Verlauf die Benutzung von Regeln mit `NoRules` verboten. Erst mit Beginn des nächsten Schrittes (`Step`-Problem) dürfen wieder Regeln verwendet werden.

- Die Voraussetzungen für die Anwendung der Regel sind nicht erfüllt, d.h. `Eval(c)`≡`False`.

$$\text{(R18)} \quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Test}_{\texttt{D}}\texttt{(c):problems)}}{\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}\texttt{(Clear(problems))}}$$

- Die Auswertung von `c` konnte nicht vollständig durchgeführt werden, d.h. `Eval(c)` ist weder `True` noch `False`, sondern enthält noch freie Variablen oder Symbole.

$$\text{(R19)} \quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Test}_{\texttt{D}}\texttt{(c):problems)}}{\texttt{Verify}^{\texttt{NoRules,B}}_{\texttt{E,(c<R>:C)}}\texttt{(problems)}}$$

  mit `R=Clear(problems)` wird der Rücksetzpunkt festgelegt, mit dem die Bedingung `c` zurückgenommen werden kann.

4.3.7 Ersetzung von *toplevel*-Deklarationen

An dieser Stelle wird die Ersetzung von *toplevel*-Deklarationen am Kopf von Ausdrücken nachgeholt, die bei der *Normalisierung I* in Regel (R4) noch nicht durchgeführt wurde. Sei also das Problem $\texttt{NormII}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{)}$ gegeben mit $\texttt{Exp}_1 = f\, exp^f_1 \ldots exp^f_n$ und $\texttt{Exp}_2 = g\, exp^g_1 \ldots exp^g_m$ $(n, m \geq 0)$, dann gilt:

- $\texttt{Value}(f) \neq f \vee \texttt{Value}(g) \neq g$

$$\text{(R20)} \quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(NormII}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\begin{array}{c}\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_{\texttt{D}}\texttt{(Value(}f\texttt{)}\, exp^f_1 \ldots exp^f_n\texttt{,} \\ \texttt{Value(}g\texttt{)}\, exp^g_1 \ldots exp^g_m\texttt{):Back}^{\texttt{M,D}}_{\texttt{Evaluate,E,C}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}\end{array}}$$

  Wenn f bzw. g eine *toplevel*-Deklaration ist, wird ihre Verwendung – analog zu (R4) – für die weitere Normalisierung gesperrt, um die Termination des Verfahrens sicherzustellen.

- sonst

$$\text{(R21)} \quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(NormII}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}\texttt{(Back}^{\texttt{M,D}}_{\texttt{Evaluate,E,C}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}$$

Dieses „verspätete“ Ersetzen von *toplevel*-Deklarationen hat damit zu tun, daß bei der Suche nach passenden Regeln aus Effizienzgründen und der Übersichtlichkeit halber, die Übereinstimmung ohne größere Manipulationen der Ausgangsterme nachgewiesen werden soll. Wenn *toplevel*-Deklarationen bereits in (R4) ersetzt werden, dann müssen sie auch bei der Suche nach Regeln ersetzt werden, d.h. die Vergleiche sind dann jeweils auf den i.allg. komplizierteren, eingesetzten Ausdrücken durchzuführen, was vermieden werden soll.

4.3.8 Auswertung

Sollten die bisherigen Anstrengungen noch nicht ausreichend gewesen sein, um die Gleichheit zweier Ausdrücke nachzuweisen, wird versucht, dieses Ziel mittels Auswertung zu erreichen. Die beiden Ausdrücke werden zunächst gemäß Abschnitt 4.2 solange ausgewertet, bis am Kopf jeweils ein nicht mehr zu ersetzendes Objekt steht. Das kann etwa der Name eines Konstruktors, eine freie Variable oder ein Namenssymbol sein. Nur wenn keiner der Ausdrücke auf diese Weise verändert werden kann, ist die Gleichheitsprüfung an dieser Stelle gescheitert. Mit dieser Situation beschäftigen sich die beiden nächsten Abschnitte.

Ansonsten gilt es, die Übereinstimmung der durch die Auswertung erhaltenen Ausdrücke zu verifizieren. Dies läßt sich in folgender Regel zusammenfassen:

- $\texttt{Eval(Exp}_1\texttt{)} \neq \texttt{Exp}_1 \;\vee\; \texttt{Eval(Exp}_2\texttt{)} \neq \texttt{Exp}_2$

$$\text{(R22)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Evaluate}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\begin{array}{c}\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Norm}_{\texttt{(DEval(Exp}_1\texttt{)++DEval(Exp}_2\texttt{)++D)}}\texttt{(Eval(Exp}_1\texttt{),Eval(Exp}_2\texttt{))}\\ \texttt{: Back}^{\texttt{M,D}}_{\texttt{Insert,E,C}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}\end{array}}$$

- sonst

$$\text{(R23)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}\texttt{(Evaluate}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}{\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}\texttt{(Back}^{\texttt{M,D}}_{\texttt{Insert,E,C}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{):problems)}}$$

Die Anzahl der Auswertungsschritte, die während $\texttt{Eval(Exp}_1\texttt{)}$ und $\texttt{Eval(Exp}_2\texttt{)}$ aufgetreten sind, wird auf das Teilproblem $\texttt{Norm}_{(\dots)}\texttt{(Eval(Exp}_1\texttt{),Eval(Exp}_2\texttt{))}$ angerechnet. Damit bleiben weiteren Auswertungen, deren Ursprung in der Verifikation dieses Teilproblems liegt, weniger Schritte, bevor die Tiefenbeschränkung greift. Auf diese Weise bleibt das System immer in einem definierten Zustand und Nichttermination wird vermieden.

4.3.9 Einfügen von Gleichungen

Nach erfolgloser Auswertung ist die Verifikation der Gleichheit zweier Teilausdrücke an sich gescheitert. Wir haben es jetzt mit dem Problem $\texttt{Insert}_{\texttt{D}}\texttt{(Exp}_1\texttt{,Exp}_2\texttt{)}$ zu tun. Bevor aber automatisch über den *Backtracking*-Mechanismus nach einem anderen Lösungsweg gesucht wird, probiert das System zunächst, ob es $\texttt{Exp}_1 \equiv \texttt{Exp}_2$ als Gleichung/Bedingung annehmen und die Beweisverifikation unter dieser Bedingung zu Ende führen kann. Diese speziellen Gleichungen werden bei der Beweisausgabe entsprechend gekennzeichnet und der Benutzer kann anhand dieser Information die problematischen Stellen in seinem Beweis erkennen.

Es werden aber nicht in jedem Fall solche Bedingungen eingefügt, sondern nur dann, wenn sich die Gleichheit wirklich noch einstellen kann. So besteht etwa keine (sinnvolle) Möglichkeit, die Gleichung $1\equiv 2$ zu zeigen oder die Übereinstimmung von `Nothing` und `Just 0`. Diese elementaren Datenobjekte sind verschieden. Der Benutzer soll aber später anhand der eingefügten Gleichungen entscheiden, ob er sie

- akzeptiert und im folgenden in die Datenbasis mitaufnimmt *oder*

- verwirft und nach anderen Lösungswegen suchen läßt

Daher machen offensichtlich ungültige Bedingungen keinen Sinn, da die Lösungen, die man unter ihnen erhält, ohne Bedeutung sind.

Gelte nun $\texttt{Exp}_1 = f\, exp_1^f \dots exp_n^f$ und $\texttt{Exp}_2 = g\, exp_1^g \dots exp_m^g$ $(n, m \geq 0)$, dann läßt sich folgendes festhalten:

- `Free(`f`)` $\vee$ `Free(`g`)` $\vee$ f,g sind *keine* Datenobjekte

$$\text{(R24)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}(\texttt{Insert}_{\texttt{D}}(\texttt{Exp}_1,\texttt{Exp}_2)\texttt{:problems})}{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,((Exp}_1\equiv\texttt{Exp}_2\texttt{)<R>:C)}}(\texttt{problems})}$$

Mit `R=Clear(problems)` wird der Rücksetzpunt bestimmt, um diese Bedingung wieder zurückzunehmen.

- sonst

$$\text{(R25)}\quad \frac{\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}(\texttt{Insert}_{\texttt{D}}(\texttt{Exp}_1,\texttt{Exp}_2)\texttt{:problems})}{\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}(\texttt{Clear(problems)})}$$

Da die Bedingungen in `C` von Zeit zu Zeit auf ihre Korrektheit überprüft werden, z.B. wenn eine freie Variable gebunden wird, ist das System auch selbständig in der Lage, eingefügte Gleichungen im Nachhinein zurückzunehmen, wenn sich herausstellt, daß sie nicht erfüllt sind. Dazu dient der in Regel (R24) mit `R` bezeichnete Rücksetzpunkt.

Unabhängigkeit von der Reihenfolge

Während der Beweisverifikation gibt es eine Menge von Teilproblemen zu lösen. Nun kann aber die Reihenfolge, in der diese abgearbeitet werden, Einfluß auf den Erfolg der Verifikation haben. Also ist man an sich gezwungen, eine geeignete Reihenfolge zu finden. Dies kann i.allg. sehr aufwendig sein, ohne überhaupt zu wissen, ob durch Umsortierung das Problem gelöst werden kann. Durch den Einsatz eingefügter Bedingungen spielt die Abarbeitungsreihenfolge keine Rolle mehr. Schauen wir uns dazu folgendes Beispiel an:

`Proof((f x,f), (x,\z`$\rightarrow$`z))`, wobei f und x freie Variablen

Der Nachweis der Gleichheit kann auf `Proof(f x, x)` und `Proof(f, \z`$\rightarrow$`z)` zurückgeführt werden.

Wenn zuerst `Proof(f, \z`$\rightarrow$`z)` behandelt wird, das die freie Variable `f` an \z$\rightarrow$z bindet, gibt es keine Komplikationen und die Verifikation kann erfolgreich durchgeführt werden.

Betrachtet man allerdings zuerst `Proof(f x, x)`, dann scheitert diese Abarbeitung, da die Unifikation wegen des fehlgeschlagenen *occur checks* nicht möglich ist und wir voraussetzen wollen, daß keine passende Regel existiert. Da sich das Ganze also nicht auf direktem Wege zeigen läßt, fügt das System zunächst noch die Bedingung `f x`$\equiv$`x` ein und geht zur Bearbeitung von `Proof(f, \z`$\rightarrow$`z)` über. Dabei wird `f` wie vorher an \z$\rightarrow$z gebunden. Damit liegt nun folgende Situation vor:

$$\texttt{f x} \equiv \texttt{x} \wedge \texttt{f} \equiv \texttt{\textbackslash z}\rightarrow\texttt{z} \quad \Rightarrow \quad \texttt{(f x,f)} \equiv \texttt{(x,\textbackslash z}\rightarrow\texttt{z)}$$

E	C	Regel	Teilprobleme von $\texttt{Verify}^{\texttt{M,B}}_{\texttt{E,C}}$
`[]`	`[]`	(R20)	`Proof(list x,1:ones) : []`
`[]`	`[]`	(R10)	`Proof(x:list x,1:ones) : []`
`[]`	`[]`	(R11)	`Proof(x,1) : Proof(list x,ones) : []`
`[x=1]`	`[]`	(R24)	`Proof(list 1,ones) : []`
`[x=1]`	`[list 1≡ones]`		`[]`

Abbildung 4.3: Verwendung als Regel (1)

Durch die Bindung von `f` kann aber die eingefügte Bedingung `f x` $\equiv$ `x` über

$$\texttt{f x} \equiv (\backslash\texttt{z}\to\texttt{z})\ \texttt{x} \equiv \texttt{x}$$

aufgelöst werden, was schließlich zur folgenden Lösung führt, die der erstgenannten Variante von vorher entspricht:

$$\texttt{f} \equiv \backslash\texttt{z}\to\texttt{z} \quad \Rightarrow \quad (\texttt{f x,f}) \equiv (\texttt{x,}\backslash\texttt{z}\to\texttt{z})$$

Verwendung als Regel

In manchen Fällen werden solche Bedingungen nicht nur eingefügt, um sie im weiteren Verlauf einfach auf ihre Konsistenz zu überprüfen, sondern sie werden gelegentlich auch als Regeln verwendet, ähnlich denen vom Benutzer vorgegebenen. Dieser Mechanismus kommt dann zum Tragen, wenn die Übereinstimmung zweier rekursiver Definitionen gezeigt werden soll.

Sehen wir uns dazu die beiden Deklarationen `list x = x:list x` und `ones = 1:ones` an. Es soll `Proof(list x, 1:ones)` gelöst werden. Hier sollte es nicht schwer fallen, zu erkennen, daß die Gleichheit für `x=1` erfüllt ist. Wenn davon ausgegangen wird, daß keine geeignete Regel vorhanden ist, geht das System bei `Proof(list x, 1:ones)` wie in Abbildung 4.3 vor. Zur Abkürzung werden nur die zur Lösung entscheidenden Schritte angegeben.

Da es sich bei `list` und `ones` um rekursive Definitionen handelt, wird später versucht, die Korrektheit der eingefügten Gleichung `list 1` $\equiv$ `ones` in einem Nachbearbeitungsschritt zu entscheiden.

Als Grundlage für das weitere Vorgehen dient folgender ER-Beweis:

```
tmp  = list 1          {- definiere tmp als list 1 -}
     = 1 : list 1      {- verwende Definition von list -}
     = 1 : tmp         {- verwende Definition von tmp -}
```

Geht man also von der willkürlichen Definition `tmp=list 1` aus, kann man für `tmp` die Gestalt `1:tmp` ableiten. Wenn nun `ones` und `tmp` strukturell übereinstimmen, also bis auf die Namen den gleichen Aufbau aufweisen, was hier der Fall ist, dann gilt auch die Gleichung `ones=list 1`. Nun kann man die Einführung von `tmp` und den strukturellen Vergleich mit `ones` dadurch zusammenfassen, daß die Rümpfe von `list` und `ones` miteinander verglichen werden und dabei die Regel `list 1` $\equiv$ `ones` verwendet wird.

$$\begin{array}{lcl} \texttt{ones} & = & \texttt{1 : }\boxed{\texttt{ones}} \\ \texttt{tmp} & = & \texttt{1 : tmp} \\ \texttt{list 1} & = & \texttt{1 : }\boxed{\texttt{list 1}} \end{array}$$

Regel	Teilprobleme von $\texttt{Verify}^{\texttt{UseRule,B}}_{\texttt{E=[x=1],C=[list1}\equiv\texttt{ones]}}$
(R3)	`Norm (x:list x, 1:ones) : []`
(R10)	`Proof(1:list 1, 1:ones) : []`
(R14)	`Proof(1,1) : Proof(list 1,ones) : []`
(R16)	`Proof(list 1,ones) : []`
`(Succ)`	`[]`

Abbildung 4.4: Verwendung als Regel (2)

Wenn `ones` und `tmp` strukturell gleich sind, dann muß an jeder Stelle `ones` im Rumpf der Definition von `ones` der entsprechende Teilterm von `list 1` die Gestalt `list 1` haben. Wenn dies nicht der Fall ist, kann die Gleichheit nicht nachgewiesen werden.

In unserem Fall gilt es also, das Problem `Norm(x:list x, 1:ones)` unter der Regel `list 1`$\equiv$`ones` und der Bindung `x=1` zu lösen (Abbildung 4.4).

Auf diese Weise wird der Nachweis der eingefügten Bedingung `list 1`$\equiv$`ones` auf das bisherige Verifikationsschema zurückgeführt. Wenn der Nachweis der Gültigkeit erbracht wird, kann diese Bedingung in der Beweisausgabe weggelassen werden.

Dieser Mechanismus war ursprünglich nur dazu gedacht, lokale Definitionen, mit denen der Benutzer vermutlich dasselbe meint, zu erkennen und triviale Gleichungen von der Beweisausgabe auszuschließen. In dem vom Benutzer angegebenen Problem

```
let ones=1:ones in (\x→x) ones {-===-} let ones=1:ones in ones
```

verbergen sich für das System hinter den `ones‘` in den `let`-Anweisungen zwei verschiedene Deklarationen. Die Namensgleichheit mag hier für der Benutzer von Bedeutung sein, aber nicht für das System. Um in der Ausgabe solche triviale Bedingungen, wie etwa $\texttt{ones}_{loc1} \equiv \texttt{ones}_{loc2}$, zu vermeiden, wurde dieser Mechanismus ins Leben gerufen.

4.3.10 Backtracking

Wenn während der Beweisverifikation ein Teilproblem nicht direkt gelöst werden kann, muß nach einem anderen Lösungsweg gesucht werden. Mit dem *Backtracking*-Mechanismus kann man zu einem früheren Zustand des Verifikationsprozesses zurückkehren, um dort einen anderen Weg einzuschlagen.

Betrachten wir dazu die Gleichung

```
(1, f (g x) z) {-===-} (z, f (g 2) 3),
```

wobei `x, y` als freie Variablen und `f, g` als *toplevel*-Funktionen gegeben sein sollen. Zur besseren Darstellung werden die Probleme `Norm` und `Proof` mit `N` bzw. `P` abgekürzt und statt $\texttt{Back}^{\texttt{M,D}}_{L,\texttt{E,C}}$ wird $\texttt{Back}^{\texttt{M,D},L}_{\texttt{E,C}}$ geschrieben. In Abbildung 4.5 wird ein Ausschnitt der Bearbeitungsschritte angegeben, die für den Nachweis der Gleichung durchgeführt werden.

Suche nach Aufsetzpunkt

Als erstes muß ein günstiger Aufsetzpunkt für die Rückkehr gefunden werden. Dieser wird mit der Funktion $\texttt{SearchBackPoint}^{\texttt{B}}_{\texttt{D}}$ bestimmt, die folgendermaßen definiert ist:

$\texttt{Verify}^{(\texttt{M=UseRule}),\texttt{B}}_{\texttt{E,C}}$(Step ((1,f (g x) z),(z,f (g 2) 3)))

E	Regel	Teilprobleme von Verify
[]	(R2)	Step (exp$_1$=(1,f (g x) z),exp$_2$=(z,f (g 2) 3)) : []
[]	(R7) (R9)	N$_{[]}$(exp$_1$,exp$_2$) : []
[]	(R10)	P$_{[]}$(exp$_1$,exp$_2$) : []
[]	(R14)	P$_{[]}$((,),(,)):N$_{[]}$(1,z):N$_{[]}$(exp$_3$=f (g x) z,exp$_4$=f (g 2) 3): **A**=(Back$^{\texttt{M},[],\texttt{Rule}}_{[],\texttt{C}}$(exp$_1$,exp$_2$) : [])
[]	(R7) (R9)	N$_{[]}$(1,z) : N$_{[]}$(exp$_3$,exp$_4$) : **A**
[]	(R11)	P$_{[]}$(1,z) : N$_{[]}$(exp$_3$,exp$_4$) : **A**
[z=1<**Z**>]	(R7) (R9)	N$_{[]}$(exp$_3$,exp$_4$) : **A** mit **Z**=(Back$^{\texttt{M},[],\texttt{Rule}}_{[],\texttt{C}}$(1,z) : N$_{[]}$(exp$_3$,exp$_4$) : **A**)
[z=1<**Z**>]	(R10)	P$_{[]}$(exp$_3$,exp$_4$) : **A**
[z=1<**Z**>]	(R14)	P$_{[]}$(f,f) : N$_{[]}$(g x,g 2) : N$_{[]}$(z,3) : **B**=(Back$^{\texttt{M},[],\texttt{Rule}}_{[z=1<\mathbf{Z}>],\texttt{C}}$(exp$_3$,exp$_4$) : **A**)
[z=1<**Z**>]	(R7) (R9)	N$_{[]}$(g x,g 2) : N$_{[]}$(z,3) : **B**
[z=1<**Z**>]	(R10)	P$_{[]}$(g x,g 2) : N$_{[]}$(z,3) : **B**
[z=1<**Z**>]	(R14)	P$_{[]}$(g,g) : N$_{[]}$(x,2) : **C**=(Back$^{\texttt{M},[],\texttt{Rule}}_{[z=1<\mathbf{Z}>],\texttt{C}}$(g x,g 2) : N$_{[]}$(z,3) : **B**)
[z=1<**Z**>]	(R7) (R9)	N$_{[]}$(x,2) : **C**
[z=1<**Z**>]	(R11)	P$_{[]}$(x,2) : **C**
[x=2<**X**>, z=1<**Z**>]	(R1)	Back$^{\texttt{M},[],\texttt{Rule}}_{[z=1<\mathbf{Z}>],\texttt{C}}$(g x,g 2) : N$_{[]}$(z,3) : **B** mit **X**=(Back$^{\texttt{M},[],\texttt{Rule}}_{[z=1<\mathbf{Z}>],\texttt{C}}$(x,2) : **C**)
[x=2<**X**>, z=1<**Z**>]	(R3) (R7) (R9)	N$_{[]}$(z,3) : **B**
[x=2<**X**>, z=1<**Z**>]	(R15)	P$_{[\mathbf{Z}]}$(1,3) : **B**
[x=2<**X**>, z=1<**Z**>]		SearchBackPoint$^{\texttt{B}}_{[\mathbf{Z}]}$ **R**=(Back$^{\texttt{M},[\mathbf{Z}],\texttt{Rule}}_{[x=2<\mathbf{X}>,z=1<\mathbf{Z}>],\texttt{C}}$(1,3) : **B**)

Abbildung 4.5: Auflistung von Bearbeitungsschritten

- Zunächst wird der Aufsetzpunkt bestimmt, der dem zuletzt gescheiterten Teilproblem zugeordnet ist. Dieser befindet sich immer am Kopf der Liste, da die Funktionen, die `SearchBackPoint` aufrufen, vorher mit `Clear` alle anderen Elemente am Anfang der Liste beseitigen. In Abbildung 4.5 handelt es sich dabei um **R**.

  Dann wird aus

  - der Liste `B`, in der sich die noch abzuarbeitenden Rücksetzpunkte befinden, die sich bereits früher während der Verifikation ergeben haben
  - der Liste `D`, welche die Stellen der Variablen-Bindungen enthält, die bei diesem Konflikt beteiligt sind
  - dem neuen, zuletzt gewonnenen Rücksetzpunkt

  der jüngste *Backtrack*-Punkt bestimmt.

  Rücksetzpunkt `x` ist jünger als `y` (i.Z. $\leq$), wenn `x` nach `y` erzeugt wurde.

  So ergibt sich für das Beispiel in Abbildung 4.5 die Ordnung der Punkte zu **R**$\leq$**X**$\leq$**C**$\leq$**B**$\leq$**Z**$\leq$**A**, da sie in der umgekehrten Reihenfolge erzeugt wurden.

$$\text{(R26)}\quad \frac{\mathtt{SearchBackPoint}^{\mathtt{B}}_{\mathtt{D}}\mathtt{(back@(Back}^{\mathtt{M,D'}}_{L,\mathtt{E,C}}\mathtt{(Exp_1,Exp_2):\_))}}{\begin{array}{c}\mathtt{let\ (R,BS)\ =}\ \text{jüngsterPunkt}\mathtt{(B{+}{+}D{+}{+}[back])\ in}\\ \mathtt{BackTo\,(R,BS)}\end{array}}$$

$$\text{(R27)}\quad \frac{\mathtt{SearchBackPoint}^{\mathtt{B}}_{\mathtt{D}}\,\mathtt{[]}\ \wedge\ \mathtt{(B\neq[]}\ \vee\ \mathtt{D\neq[])}}{\mathtt{let\ (R,BS)\ =}\ \text{jüngsterPunkt}\mathtt{(B{+}{+}D)\ in\ BackTo\,(R,BS)}}$$

  Nachdem der Aufsetzpunkt `R` und die verbleibenden Punkte `BS` bestimmt wurden, kehrt man mit `BackTo (R,BS)` zu diesem früheren Zustand zurück.

- Wenn es keine Rücksetzpunkte mehr gibt, dann ist die Verifikation endgültig gescheitert. Dies läßt sich in der Bedingung

$$\mathtt{(Fail)}\qquad \mathtt{SearchBackPoint}^{\mathtt{B}}_{\mathtt{D}}\mathtt{[]}\ \wedge\ \mathtt{B\equiv[]}\ \wedge\ \mathtt{D\equiv[]}$$

  zusammenfassen.

Rückkehr zu Aufsetzpunkt

Nachdem der eigentliche Aufsetzpunkt bestimmt wurde, kann man nun dorthin zurückkehren, wie mit der Funktion `BackTo` beschrieben. Es ist zu beachten, daß die Information, die zwischen Rückkehrpunkt und Konfliktsituation gewonnen wurde, verloren geht.

- Solange noch Regeln zur Verfügung stehen, die auf das Konfliktpaar $\mathtt{Exp}_1$ und $\mathtt{Exp}_2$ angewendet werden können, werden diese sukzessive durchprobiert. Es wird also die erste Regel $\mathtt{(c{\Rightarrow}l{\equiv}r)}$ aus `RS` genommen und, analog zu Abschnitt 4.3.6, auf ihre Anwendbarkeit hin überprüft. Die Liste `TRS` enthält die noch verbleibenden Regeln, auf die bei Bedarf in einem späteren Schritt zurückgegriffen werden kann.

$$\text{(R28)}\quad \frac{\mathtt{BackTo\,(Back^{M,D}_{NextRule(RS),E,C}(Exp_1,Exp_2):problems,BS)}}{\mathtt{Verify^{TestRule,BS}_{E,C}(Norm_{[]}(Exp_1,l):Norm_{[]}(Exp_2,r):Test_{[]}(c):} \atop \mathtt{Back^{M,D}_{NextRule(TRS),E,C}(Exp_1,Exp_2):problems)}}$$

- Wenn keine Regel mehr vorhanden ist, dann werden die *toplevel*-Deklarationen am Kopf der beiden Ausdrücke ersetzt, um mehr Information für die weitere Verifikation zur Verfügung zu haben.

$$\text{(R29)}\quad \frac{\mathtt{BackTo\,(Back^{M,D}_{NextRule([]),E,C}(Exp_1,Exp_2):problems,BS)}}{\mathtt{Verify^{M,BS}_{E,C}(NormII_D(Exp_1,Exp_2):problems)}}$$

- Falls es sich bei L um `Rule`, `NormII`, `Evaluate` oder `Insert` handelt, wird das entsprechende Problem behandelt, das bereits in einem der vorherigen Abschnitte beschrieben wurde.

$$\text{(R30)}\quad \frac{\mathtt{BackTo\,(Back^{M,D}_{\mathit{L},E,C}(Exp_1,Exp_2):problems,BS)}}{\mathtt{Verify^{M,BS}_{E,C}(\mathit{L}_D(Exp_1,Exp_2):problems)}}$$

In `D` befinden sich, wie bereits früher angesprochen, die Rückkehrpunkte für diejenigen gebundenen Variablen, die während der Abarbeitung eines Teilproblems benutzt wurden. Damit werden beim *Backtracking* nur solche Variablenbindungen explizit aufgelöst, die wirklich zum Konflikt beitragen.

So werden bei dem Beispiel zu Beginn dieses Abschnittes bei dem Konflikt $\mathtt{Proof}_{(\mathtt{D}=[\mathbf{Z}])}\mathtt{(1,3)}$ nur die Rücksetzpunkte **Z** und **R** berücksichtigt. Die Stelle **X**, bei der die Variable `x` gebunden wurde, wird nicht behandelt (ist nicht in `D`), weil sie zur Beseitigung des Problems nichts beisteuern kann.

Gezieltes Zurücksetzen von Bindungen und Bedingungen

Natürlich kann der Benutzer auch nach der Beweisverifikation auf den *Backtracking*-Mechanismus zurückgreifen. So kann er in einer Lösung, die vom System gefunden wurde, gezielt Variablenbindungen, eingefügte Gleichungen und Prämissen von Regeln verwerfen und das System dazu veranlassen, nach einer neuen Lösung zu suchen.

```
f (0 * x) x
{-
  16    x = 2
  18    Rule: mult_zero
-}
f 0 2

Backtrack (y/n) [y] y
Which Numbers [All]: 18
```

Durch Angabe der entsprechenden Kennzahl, z.B. 18, wird vom System eine Bindung bzw. Bedingung zurückgenommen und nach einer anderen Variante gesucht.

Erst wenn es keine Aufsetzpunkte mehr gibt und das *Backtracking* aus diesem Grund nicht mehr möglich ist, scheitert die Beweisverifikation schließlich.

4.3.11 Typüberprüfung

Während der Beweisverifikation selbst wird von Seiten des Systems nicht auf Typen geachtet. Um dennoch typkorrekte Lösungen zu garantieren, schließt sich an die eigentliche Beweisprüfung eine externe Typüberprüfung an.[3] Erst wenn diese erfolgreich abgeschlossen wird, wird von einer korrekten Lösung gesprochen. Andernfalls ist dieser Lösungsversuch als gescheitert aufzufassen und es wird mittels *Backtracking* nach einer anderen Lösung gesucht.

Betrachten wir dazu das Problem `Proof(const 1 x,const 1 [])`, wobei `x` eine freie Variable vom Typ `Int` sein soll. Die erste vorläufige Lösung bindet `x` an die leere List `[]`, was aber aus Gründen der Typkorrektheit nicht zulässig ist. Der Typ-Checker liefert daher auch einen Typ-Fehler und veranlaßt so das System, die Bindung `x=[]` als gescheitert aufzufassen. Wenn in diesem Beispiel `const` mit der Deklaration `const x y=x` zur Auswertung freigegeben wurde, findet das System schließlich die Gleichheit der beiden Ausdrücke, die keinerlei Restriktionen beinhaltet.

4.3.12 Termination

Ein exakter Beweis der Termination mit einer Abstiegsfunktion würde an dieser Stelle zu weit führen, weil allein die Annotation der vorher angegebenen Regeln mit den diversen Schranken, die die Auswertung und das Ersetzen von Deklarationen betreffen, mehr zur Verwirrung als zur Klarheit beitragen würden. Stattdessen wird hier nur ein Plausibilitäts-„Beweis“ geliefert.

- Zu einer Konfliktstelle wird höchstens viermal zurückgekehrt, bis dieser Rücksetzpunkt endgültig abgearbeitet und der nächst ältere betrachtet wird. „Vier“ ergibt sich aus den Problemen `Rule`, `NormII`, `Evaluate` und `Insert`, mit denen ein Rücksetzpunkt `Back` annotiert werden kann.

- Zwei zu vergleichende Ausdrücke können im Verlaufe der Verifikation nur in endlich viele Teilprobleme aufgespaltet bzw. transformiert werden. Dafür sorgen die Tiefenbeschränkung bei der Auswertung (Abschnitt 4.2) und das *ein*malige Ersetzen von Deklarationen in den Regeln (R4) und (R20). Danach ergeben sich keine Änderungen mehr. Insgesamt können also nur endlich viele *Backtrack*-Punkte aus den beiden Ausdrücken neu hervorgehen.

Da also im Konfliktfall nur endlich viele Rücksetzpunkte zu bearbeiten sind, gelangt man schließlich nach endlich vielen Schritten zu einem Ergebnis.

4.4 Start der Verifikation

Wenn wir davon ausgehen, daß der Benutzer die Korrektheit der Beweisschritte

$$E_1 \equiv E_2 \equiv \ldots \equiv E_{n-1} \equiv E_n \ (n \geq 2)$$

zeigen möchte, dann läßt sich dies mit dem Aufruf

$$\texttt{Verify}^{\texttt{UseRule},[]}_{[],[]}(\ \texttt{Step}(E_1,E_2),\texttt{Step}(E_2,E_3),\ldots,\texttt{Step}(E_{n-1},E_n)\ )$$

[3] für das vorliegende System wird z.Zt. hugs-1.3 verwendet

erreichen. Zu Beginn ist die Menge der Rücksetzpunkte, Bindungen und Bedingungen leer. Erst im Laufe der Abarbeitung ergeben sich zusätzliche Anforderungen, die in diese Mengen aufgenommen werden. Die Rücksetzpunke werden dazu verwendet, einschränkenden Voraussetzungen zurückzunehmen, um nach einer evtl. günstigeren Lösung zu suchen.

4.5 Ausgabe von lokal rekursiven Funktionen

Bei der Ausgabe einer Lösung benötigen lokal rekursive Funktionen eine besondere Behandlung. Da deren Funktionsnamen nur lokal bekannt sind und mit anderen Deklarationen in Konflikt stehen können (mit lokalen oder *toplevel*-Deklarationen), ist es bei der Ausgabe nicht einfach möglich, diese Namen zu verwenden. Es darf nicht vergessen werden, daß `let`-Konstrukte aufgelöst wurden (Abschnitt 4.1), und diese nachträglich wieder einzuführen, kann sich auf das textuelle Erscheinungsbild der Ausgabe nachteilig auswirken, ganz abgesehen von den Problemen, die geschachtelte `let`-Ausdrücke und freie Variablen in diesem Zusammenhang bereiten.

Bei nicht rekursiven Definitionen besteht die Möglichkeit, den entsprechenden Rumpf bei der Ausgabe einzusetzen und somit dieses Problem zu umgehen. Bei rekursiven Definitionen — egal, ob direkt oder indirekt rekursiv — kommt man mit diesem Ansatz nicht zum Ziel. Für derartige Deklarationen wird jeweils eine *toplevel*-Deklaration angelegt, die die freien Variablen der lokalen Deklaration als formale Parameter erhält [20, 31]. Bei der Ausgabe wird nun anstelle des nur lokal verfügbaren Namens die entsprechende *toplevel*-Definition verwendet, die die freien Variablen als Argumente erhält. Die neu eingefügten Funktionen werden im Anschluß an die eigentliche Beweisausgabe aufgeführt, damit der Benutzer über sie in Kenntnis gesetzt wird. Betrachten wird dazu folgendes Beispiel:

```
let list = y : list in list {-===-} let list=[] in x,
```

wobei `x` und `y` freie Variablen vom Typ `[Int]` bzw. `Int` sind. Hier wird also die Variable `x` an die lokale Definition `list` der linken Seite gebunden. Allerdings kann bei der Ausgabe für die Bindung von `x` nicht einfach `x=list` geschrieben werden. Für die lokale Definition `list` mit freier Variable `y` erzeugt man zunächst eine globale Deklaration `list_1 y = y : list_1 y` und erhält auf diese Weise die Ausgabe

```
    list_1 y {- x = list_1 y -} x
mit
    list_1 y = y : list_1 y
```

wie sie das System für das obige Problem liefern würde.[4]

Diese Transformationen von lokalen Deklarationen zu *toplevel*-Deklarationen findet erst bei der Ausgabe statt, d.h. es wird zu Beginn darauf verzichtet, die Eingabe dahingehend zu normalisieren, sondern sie wird nur bei Bedarf durchgeführt.

[4] die Namensvergabe der neu eingefügten Funktion und die Formatierung kann abweichen

Kapitel 5

Implementierung

Die Darstellung in diesem Kapitel beschreibt lediglich einen Teil der Funktionen und Datentypen, die für den Betrieb des Systems notwendig sind. Die Funktionen lassen sich grob in drei Klassen einteilen, die für die

- Benutzeroberfläche
- Verifikation des Beweises
- allgemeine Ein-/Ausgabe

verwendet werden. Dieses Kapitel beschäftigt sich vor allem mit dem zweiten Punkt, mit der Umsetzung der im vorangegangen Kapitel gezeigten Ableitungsregeln. Dazu werden zunächst die wichtigsten Datentypen beschrieben, die u.a. für die Repräsentation von Haskell-Ausdrücken und zur Darstellung der abzuarbeitenden Teilprobleme verwendet werden. Dann erfolgt eine Auflistung der zentralen Funktionen, die für die Überprüfung des Beweises verantwortlich sind. Diese Funktionen kommen, je nachdem in welcher Phase sich der Verifikationsprozeß befindet, zum Zuge und werden mit den ihnen zugrundeliegenden Ableitungsregeln in Beziehung gesetzt. Auf die Beschreibung, wie aus den Eingabedaten bzw. den Daten, die während der Beweisprüfung gewonnen wurden, eine Datei für die externe Typ-Überprüfung erstellt wird, wird an dieser Stelle verzichtet. Ihr Aufbau läßt sich allerdings leicht an einer vom System erzeugten Datei ablesen. Eine Aufstellung der verwendeten Dateien erfolgt im letzten Abschnitt dieses Kapitels. Zu Beginn wird jedoch auf die Eingabesprache eingegangen, welche Einschränkungen gegenüber Standard-Haskell vorgenommen wurden und mit welchen Hilfsmitteln die Grammatik erstellt wurde. Dabei steht die Umsetzung des *Layout Styles* im Mittelpunkt, die mit den gegebenen Mitteln nicht direkt erreichbar war.

5.1 Unterstützte Eingabesprache

Das System dient zur Verifikation von ER-Beweisen in Haskell. Damit für den praktischen Einsatz die Anpassungen an die Eingabesprache so gering wie möglich ausfallen, werden — was die Angabe von Deklarationen und Ausdrücken betrifft — beinahe alle Sprachkonstrukte Haskells unterstützt. Ferner ist auch der sog. *Layout-Style* [30, 1.5] bei der Eingabe verfügbar, d.h. durch geeignete Formatierung lassen sich die Kontrollstrukturen „`;`“, „`{`“ und „`}`“ einsparen, was mit zu einer übersichtlicheren Programmgestaltung beiträgt. Der Benutzer ist überdies nicht nur an die von Haskell vordefinierten

Infix-Operatoren gebunden, sondern er ist in der Lage über `infixl, infix` und `infixr` eigene Operatoren mit der gewünschten Präzedenz zu definieren, so wie in Haskell auch.

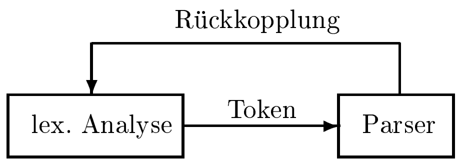
Einschränkungen

Bei der Eingabesprache wurde auf die Unterstützung von sog. *labeled fields* und *labeled patterns* [30, 3.15] verzichtet, da deren Umsetzung einen nicht vertretbaren Aufwand bedeutet hätte. Der Einsatz von `do`-Anweisungen, die für die Notation von sog. monadischen Operationen gedacht ist [40], wird ebenfalls nicht unterstützt. Der Striktheits-Operator `seq`, den Haskell bietet, wird nicht behandelt, d.h. die Auswertung der Ausdrücke erfolgt immer lazy. Um die Grammatik für die Eingabesprache etwas einfacher zu gestalten, müssen negierte Ausdrücke geklammert werden, d.h. ein Ausdruck der Form `-x+y` muß als `(-x)+y` notiert werden. Schließlich sollten die Zeichenkonstanten `hypo`, `equiv` und `inductive` nicht für die Definition eigener Funktionen verwendet werden, da diese bereits vom System benutzt werden. Dies gilt ebenfalls für *Bezeichner<Num>_<Num>*, wobei *<Num>* für eine beliebige natürliche Zahl steht. Diese Namen werden verwendet, um lokale Bezeichner durch eindeutige Bezeichner zu ersetzen. Dies sind die wesentlichen Einschränkungen, da sie die Angabe von Haskell-Ausdrücken und -Definitionen betreffen.

Was typbezogene Sprachkonstrukte, wie etwa `data, type, newtype, class` und `instance`, betrifft, so werden sie aus dem Grunde nicht beachtet, weil diese Information für die Beweisverifikation selbst ohne Bedeutung ist. Diese Angaben müssen aber u.U. dem Typ-Checker in einer dafür vorgesehenen Datei übergeben werden. Genauere Angaben hierzu finden sich in Abschnitt 6.1.4.

5.2 Parser

Bei der Entwicklung der Grammatik für die Eingabesprache wurde der Haskell-1.3 Report zugrundegelegt. Die Umsetzung erfolgte mit dem Parser-Generator *happy-0.9a*,[1] der das funktionale Gegenstück zu *yacc* [14] bzw. *bison* darstellt, die unter UNIX für imperative Programme weit verbreitet sind. *happy* erzeugt Haskell-Code, aber befindet sich in dieser Version noch in einem nicht voll ausgereiftem Alpha-Stadium, das u.a. keine Fehlerbehebung (vgl. *yacc* mit `error`-Token) und keine Rückkopplung zur lexikalischen Analyse zuläßt[2]. Aber gerade diese beiden Aspekte sind für die Behandlung des *Layout-Styles* erforderlich.



Die lexikalische Analyse (`Lexer.lhs`) ist weitgehend selbständig in der Lage die durch die Formatierung eingesparten Kontrollstrukturen wieder einzusetzen. Aber ein Teil der *Layout*-Regel [30, 1.5] besagt, daß immer dann im *Layout*-Modus das Token „}“ einzufügen ist, wenn ein anderes Token im Eingabestrom seinen Platz einnimmt und es folglich zu einem Syntax-Fehler kommt. Dies entzieht sich der Kontrolle der

[1] happy kann über `ftp://ftp.dcs.gla.ac.uk/pub/haskell/happy` bezogen werden

[2] diese Rückkopplung soll laut Dokumentation in einer der folgenden Versionen verfügbar sein

lexikalischen Analyse und wird nur vom Parser erkannt. Der *Layout*-Modus wird dann eingeleitet, wenn auf die Schlüsselwörter `of`, `let` und `where` kein „{“ folgt.

Umsetzung des *Layout-Styles*

Da der *Layout-Styles* fester Bestandteil der Haskell-Programmierung ist und deswegen nicht unbehandelt bleiben sollte, aber man auf der anderen Seite bei der Entwicklung des Parsers nicht auf die Vorzüge eines Parser-Generators verzichten möchte, werden die notwendigen Manipulationen in der von *happy* erzeugten Ausgabe-Datei vorgenommen. Da es sich dabei nur um vier Stellen handelt, hält sich der Aufwand in Grenzen. Außerdem braucht man jeweils nur den Aufruf `happyFail` durch

`insertTokenCloseCurBrackLayout` *nextState* 73

zu ersetzen, wobei *nextState* den Folgezustand des `LALR`-Parsers angibt.

Um diese vier Stellen zu finden, an denen neben dem Einfügen von „}“ auch die Verbindung zur lexikalischen Analyse hergestellt wird, benötigt man Grundkenntnisse über die Funktionsweise von `LALR`-Parsern, welche von *happy* erzeugt werden [1].

Von großem Vorteil bei der Suche ist eine Informationsdatei, die *happy* mit dem Kommandozeilenargument `--info` erzeugt, die eine textuelle Auflistung des deterministischen endlichen Automaten zeigt, der für die syntaktische Analyse verantwortlich ist. Der folgende Ausschnitt aus dieser Datei zeigt eine der für unser Problem entscheidenden Stellen, die anderen drei haben einen analogen Aufbau.

```
State 145

        decllist -> "{" decls . "}"                      (rule 24)
        decllist -> "{" decls . ';' "}"                  (rule 25)
        decls -> decls . ';' decl                        (rule 26)

        ';'            shift, and enter state 152
        "}"            shift, and enter state 153
```

Der Automat erwartet in diesem Zustand entweder ein `"}"` oder ein `';'` als nächstes Token. Die aktuelle Position innerhalb der Grammatik-Produktion, die bis zu diesem Zeitpunkt erkannt wurde, ist durch den Punkt „`.`“ gekennzeichnet. In der Terminologie des Compilerbaus spricht man hier auch von einem `LR`-Element. Entscheidend ist an dieser Stelle nur, daß jedes andere Token zu einem syntaktischen Fehler führt, was bei *happy* durch den Aufruf von `happyFail` ausgedrückt wird. Da sich der Automat in der Informationsdatei und dessen Umsetzung in Haskell bei diesem Parser-Generator in offensichtlicher Weise entsprechen, findet man mit der `State`-Angabe die zugehörige Haskell-Funktion (`action_145`), die diesen Zustand beschreibt und zu manipulieren ist. Mit `73` wird das Token `"}"` repräsentiert, was in der Funktion `happyNewToken` in der von *happy* erzeugten Ausgabedatei abgelesen werden kann. Der Folgezustand, der der Funktion `insertTokenCloseCurBrackLayout` als erstes Argument übergeben wird, läßt sich an der Zeile

```
"}"            shift, and enter state 153
```

ablesen und ergibt sich in diesem Fall zu `action_153`.

5.3 Wichtige Datentypen

Zur Darstellung eines Haskell-Ausdruckes wird der Datentyp `ResExp` verwendet. Es handelt sich hierbei um eine flache Repräsentation, d.h. nicht alle Haskell-Konstrukte werden durch einen eigenen Konstruktor in `ResExp` beschrieben. Vielmehr faßt man die Eigenschaften der meisten von ihnen in einem einzigen Konstruktor zusammen und unterscheidet sie mit Hilfe des Datentyps `ResSpec`. Dadurch lassen sich auf `ResExp` kompaktere Funktionen schreiben, da weniger Fälle beim Pattern-Matching berücksichtigt werden müssen als bei einer feinen Repräsentation. Für die Beschreibung von Variablen wird jedoch ein eigenständiger Konstruktor verwendet, da sie u.a. bei der Unifikation eine wichtige Rolle spielen. In `ResExp` werden auch die speziellen Pattern *Wildcard, As-Pattern* und *Lazy-Pattern* gesondert behandelt, was die Tatsache unterstreicht, daß man Pattern als spezielle Ausdrücke auffassen kann.

Der Datentyp `DoTest` beschreibt im wesentlichen die Probleme, die während der Verifikation eines Beweises nachzuweisen sind. Im Gegensatz zur feinen Strukturierung in Abschnitt 4.3.1, beschränkt man sich hier auf wenige Teilprobleme, da viele Aufgaben, wie etwa die Suche nach einer Regel, direkt von der *Backtracking*-Komponente heraus erledigt werden und nicht indirekt über eine Ableitungsregel, die für ein entsprechendes Teilproblem konzipiert ist.

Schließlich ist das Ergebnis der Verifikation von besonderem Interesse. Dafür wird der Datentyp `Result` verwendet. Im Erfolgsfall muß man darin ablesen können, unter welchen Voraussetzungen die Gültigkeit des Beweises gegeben ist. Also befinden sich dort die nötigen Variablenbindungen und Bedingungen, die etwa durch Regelanwendungen eingeführt wurden. Ferner ist auch Information enthalten, die zum *Backtracking* notwendig ist, wenn der Benutzer ausgehend von dieser Lösung noch andere Varianten ableiten will.

Am Typsynonym `ProofType` lassen sich die Argumente ablesen, die bei der Verifikation benötigt werden. So erkennt man dort u.a. die Datenbasis der Regeln und die Menge der Deklarationen, die für die Normalisierung gebraucht werden.

5.3.1 `ResExp`

Haskell-Ausdrücke werden durch den algebraischen Datentypen `ResExp` repräsentiert. Die Unterscheidung der einzelnen Haskell-Konstrukte, wie etwa λ-Abstraktionen oder `case`-Anweisungen, wird an dieser Stelle nicht direkt durch Vergabe eines entsprechenden Konstruktors erreicht. Stattdessen beschränkt man sich auf lediglich drei Konstruktoren (`ResPat, ResVar` und `ResExp`), deren vollständige Bedeutung erst durch weitere Argumente festgelegt wird.

```
data ResExp
   =    ResPat Pattern [EnvId] [ResExp]
   |    ResVar ExpEnv VarType EnvId
   |    ResExp ExpEnv ResSpec [ResExp]
   deriving Show
```

Die Struktur `ExpEnv` beschreibt die Umgebung, in der die Werte von Variablen zu finden sind, die durch λ-Abstraktionen gebunden werden. Die Implementierung führt nämlich bei einer β-Reduktion keine explizite Substitution durch, sondern verwaltet die Bindungen in `ExpEnv`. `EnvId` ist ein Identifikator, der jeder Variablen zugewiesen wird und mit dem ein effizienter Zugriff innerhalb `ExpEnv` und verwandter Datenstrukturen möglich ist. In der Liste `[ResExp]` befinden sich diejenigen Argumente, die von dem repräsentierten Haskell-Konstrukt benötigt werden. Für einen Binäroperator z.B. sind in dieser Liste die Beschreibung des Operators (wie etwa `(+)`,`(*)`) und die beiden zugehörigen Operanden, also insgesamt drei Elemente.

Mit `ResPat` werden die speziellen Pattern für *Wildcard*, *As-Pattern* und *Lazy-Pattern* behandelt, deren Unterscheidung durch den Aufzählungstyp `Pattern` getroffen wird. Ein Pattern wird in diesem Sinne als besonderer Haskell-Ausdruck vom Typ `ResExp` aufgefaßt, der nur innerhalb von `case`-Anweisungen auf der linken Seite der einzelnen Alternativen auftreten darf.

Die Menge der Variablen, welche durch `ResVar` gekennzeichnet sind, wird gemäß `VarType` in drei Klassen eingeteilt.

- `VarLambda`: Diese Variable ist durch eine λ-Abstraktion gebunden. Ihr Wert kann durch Inspektion der zugehörigen Umgebung bestimmt werden.
- `VarFree`: Die Variablen, die vom Benutzer als frei deklariert werden[3], sind durch dieses Attribut gekennzeichnet. Im Gegensatz zu `VarLambda`-Variablen, die immer durch eine β-Reduktion an einen Wert gebunden sind, können `VarFree`-Variablen nur durch Unifikation an einen Ausdruck gebunden werden. Diese Bindungen werden in der Struktur `Bindings` verwaltet (vgl. Abb. 5.2).
- `VarDefined`: Streng genommen handelt es sich hier nicht um „echte“ Variablen, die ihren Wert ändern können, sondern sie bezeichnen die Namen der Deklarationen, die im Programm angegeben wurden. Es wird, wie im Abschnitt 4.1 angegeben, nicht zwischen lokalen und *toplevel*-Deklarationen unterschieden. Mit dem angegebenen Identifikator greift man über die Struktur *Declarations* (vgl. Abb. 5.2), auf die entsprechende Deklaration zu. Die in lokalen Deklarationen u.U. vorkommenden „freien“ `VarLambda`-Variablen werden dabei durch `ExpEnv` implizit gebunden, entsprechend den Erläuterungen in Abschnitt 4.1.

Die restlichen Haskell-Konstrukte werden durch den Konstruktor `ResExp` repräsentiert. Eine genaue Differenzierung erfolgt über den Datentypen `ResSpec`, der folgende Alternativen beinhaltet.

`data ResSpec`

`= ResApply ApplyType`
Eine Folge von Funktionsapplikationen wird aus Gründen der Effizienz nicht durch eine Reihe von binären Applikations-Operatoren beschrieben, sondern sie wird in einer Struktur zusammengefaßt, wie Abb. 5.1 zeigt. `ApplyType` gibt dabei an, wieviele Applikationen miteinander kombiniert werden und ob es sich bei der Funktion ganz „links“ in der Folge um einen Infix-Operator handelt, was für die Ausgabe wichtig ist. Der Zugriff auf die linksstehende Funktion war für diese Art der Repräsentation ausschlaggebend, da bei der herkömmlichen, wie man an der Baumdarstellung in Abb. 5.1 erkennen kann, die Struktur linksrekursiv durchlaufen werden muß.

`| ResLambda`
damit werden λ-Abstraktionen gekennzeichnet, d.h. das erste Element der `ResExp`-Liste enthält den formalen Parameter und das zweite den Rumpf der Abstraktion

`| ResCase`
beschreibt eine `case`-Anweisung. Im ersten Element der `ResExp`-Liste befindet sich

[3] vgl. dazu `{-Types-}`-Angabe in Abschnitt 6.1.1

der Ausdruck, auf den die in dieser Liste folgenden Pattern (repräsentiert durch `ResCaseAlt`-Elemente) angewendet werden.

`| ResCaseAlt`
in der zugehörigen `ResExp`-Liste findet sich zum einen das Pattern und zum anderen der Ausdruck, der an dieses Pattern gebunden ist.

`| ResLiteral Literal`
kennzeichnet Literale für `Int`, `Float` und `Char`

`| ResCons Constructor`
damit werden Konstruktoren repräsentiert, die mittels `Constructor` unterschieden werden. Neben den speziellen, vordefinierten Konstruktoren, wie `()`,`[]` und Tupel beliebiger Stelligkeit, sind die restlichen durch ihr textuelles Erscheinungsbild festgelegt. Sie beginnen entweder mit einem Großbuchstaben oder sind von zwei „`:`“ eingerahmt. Eine Ausnahme bildet in dieser Hinsicht der Listenkonstruktor „`:`“.

`| ResVarStr String NameType`
In `String` befindet sich der Name eines Symbols, d.h. dieser Name konnte keiner Deklaration, freien Variablen oder einem formalen Parameter zugeordnet werden. Die Verwendung von Symbolen wird in 6.1.4 beschrieben. `NameType` enthält Information, die für die Ausgabe benötigt wird. So muß z.B. für `elem` bei der Verwendung als Infix-Operator `` `elem` `` ausgegeben werden und für `+` als Präfix-Operator `(+)`, um sich an die Haskell-Konvention zu halten.

`| ResMarked Mark EnvId`
dient einerseits dazu, um bereits verwendete, Deklarationen zu markieren, wie in 4.3.3 und 4.3.7 angesprochen, andererseits werden damit auch die *Pseudo*-Argumente beschrieben, die in 4.3.4 eingefügt werden. `EnvId` enthält jeweils einen Identifikator, um die einzelnen Elemente zu unterscheiden.

`| ResError String`
Fehlermeldung von Seiten des Parsers aufgrund einer ungültigen Eingabe, die im Gegensatz zu Syntax-Fehlern erst bei der semantischen Analyse erkannt werden, z.B. der Versuch einer wiederholten Funktionsdefinition.

```
deriving (Show,Eq)
```

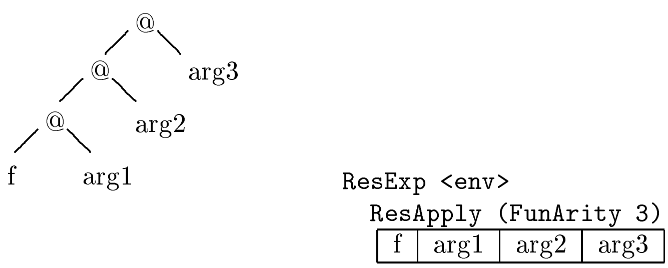


Abbildung 5.1: Funktionsapplikation

Die Verwendung des Datentyps `ResExp` ermöglicht es, auf einfache Weise Funktionen zu schreiben, die im großen und ganzen unabhängig von den Haskell-Konstrukten sind. In diesen Fällen sind nur wenige Alternativen, d.h. Pattern, notwendig, um das Problem zu behandeln. Ein Beispiel hierfür ist die Suche nach freien Variablen in einem Ausdruck, bei der rekursiv auf die in `[ResExp]` gegeben Argumente zugegriffen werden kann. Wenn für jedes Haskell-Konstrukt ein eigenständiger Konstruktor in `ResExp` verwendet würde, müßten diese alle in der entsprechenden Funktion behandelt werden.

Durch diese „flache“ Repräsentation geht etwas an Information verloren, da kein Zusammenhang zwischen den Haskell-Konstrukten und der Anzahl und Art ihrer Argumente hergestellt werden kann. Die Konstruktoren `ResPat` und `ResExp` erzwingen keine bestimmte Ausprägung in Abhängigkeit von `Pattern` bzw. `ResSpec`, da die Argumente einheitlich in einer Liste von `ResExp`-Elementen zusammengefaßt werden. Eine gültige Zuordnung ergibt sich nur indirekt über den Wert der beim Konstruktor angegebenen Argumente vom Typ `Pattern` bzw. `ResSpec`.

5.3.2 `DoTest`

Die Elemente des algebraischen Datentyps `DoTest` beschreiben im wesentlichen die Teilprobleme, die während des Verifikationsprozesses zu lösen sind und entsprechen den Elementen, auf denen die Funktion $\mathtt{Verify}^{\mathtt{M,B}}_{\mathtt{E,C}}$ aus 4.3.1 arbeitet.

`data DoTest`

`= OnApply {u_exp1, u_exp2::ResExp, u_drefVs::[DerefVar]}`
zeigt an, daß auf `u_exp1` und `u_exp2` die *Normalisierung I* aus [4.3.3] anzuwenden ist. `u_drefVs` enthält Angaben über die bis zu diesem Zeitpunkt verwendeten `VarFree`-Variablen, die im Fehlerfall bei der Suche nach Aufsetzpunkten verwendet werden. Dies ist die Umsetzung der `D`-Komponente, die im Abschnitt 4.3.1 in den Teilproblemen, wie etwa $\mathtt{Norm}_{\mathtt{D}}$, als Attribut auftritt.

`| OnSuccess {u_exp1, u_exp2::ResExp, u_drefVs::[DerefVar]}`
hier befinden sich `u_exp1` und `u_exp2` bereits in Normalform. Auf der Basis dieser Darstellung wird versucht das Probem in kleinere Teilprobleme aufzuspalten. Es handelt sich somit um das Gegenstück von `Proof` aus [4.3.5].

`| OnReplace`
`{ u_insCond::[DoTest], u_Success::DoTest, u_mode::UModes,`
`u_info::Info, u_ID::EnvId, u_bfc::BFCT }`
darin ist die Information enthalten, die nach dem Rücksetzen bei einem Fehler wiederhergestellt werden muß, um ausgehend von diesem geretteten Zustand nach einem neuen Lösungsweg zu suchen. Dieser Konstruktor ist mit $\mathtt{Back}^{\mathtt{M,D}}_{L,\mathtt{E,C}}$ gleichzusetzen und wird von der Funktion `failure` verwendet, die für das *Backtracking* verantwortlich ist und später erläutert wird. `u_insCond` enthält Aufzeichnungen, die beim Einfügen von Bedingungen gemäß [4.3.9] zu beachten sind. Die beiden zu vergleichenden Ausdrücke befinden sich in `u_Success`. In `u_mode` und `u_info` findet sich zum einen der Modus, in dem sich die Beweisprüfung befindet, d.h. u.a ob Regeln angewendet werden dürfen oder nicht, zum anderen Angaben über die zuletzt verwendete Regel. Für die Vergabe von neuen Identifikatoren wird der zuletzt gültige Wert in `u_ID` gerettet. In `u_bfc` werden schließlich die Bindungen der `VarFree`-Variablen verwaltet und die Menge der Bedingungen, die für den Beweis erfüllt sein müssen.

```
| Succeeded Int
```
dient nur zur Abgrenzung der einzelnen Beweisschritte, die zu überprüfen sind, um dem Benutzer über den Stand der Verifikation unterrichten zu können. Der `Int`-Wert gibt an, wieviele Beweisschritte noch zu bearbeiten sind.

```
| Utility Utilities
```
damit werden veschiedene „Aufräumaktionen“ initiiert. Die abschließende Überprüfung der Prämisse einer Regel – `Test(c)` – wird hier u.a. in Auftrag gegeben.

```
deriving Show
```

Die Umsetzung der Teilprobleme `Rule`, `NextRule`, `NormII`, `Evaluate` und `Insert`, die erst nach einem Fehlerfall auftreten, erfolgt direkt aus der Funktion `failure` heraus, in der die zugehörigen Funktionen aufgerufen werden.

5.3.3 `Result`

Das Ergebnis der Beweisverifikation wird in `Result` ausgedrückt, das folgenden Aufbau hat[4].

```
data Result
  = UnifySuccess [DoTest] Log [DerefVar] Info UModes EnvId BFCT
  | UnifyFailed Log
```

Wenn der Beweis nicht erfolgreich nachgewiesen werden kann, wird dies durch den Konstruktor `UnifyFailed` ausgedrückt. In `Log` werden alle Aktionen protokolliert, die während der Verifikation durchgeführt wurden. Diese Information ist allerdings nicht für den Benutzer gedacht, sondern diente zur Fehlersuche. `UnifySuccess` signalisiert, daß die Überprüfung des Beweises erfolgreich verlaufen ist. In `BFCT` (vgl. Abb. 5.2) befinden sich die erforderlichen Bindungen und Bedingungen, die dafür notwendig sind.

5.3.4 `ProofType`

Die Funktionen, die für die eigentliche Beweisverifikation verantwortlich sind, haben den Typ `ProofType` oder einen davon abgeleiteten Typ, der durch partielle Applikation dieser Funktionen entsteht. Es wurde an dieser Stelle auf den Einsatz von Monaden [38, 40] verzichtet, da viele Funktionen nur sehr wenige der in `ProofType` aufgeführten Elemente benötigen und selten auf alle zugreifen müssen. Dadurch, daß die häufig verwendeten Argumente als erste übergeben werden, reicht es in den meisten Fällen aus, bei der Funktionsdefinition nur einige wenige Argumente anzugeben und die anderen unberücksichtigt zu lassen. Bei dieser Art der Programmierung kommt die Verwendung von *Funktionen höherer Ordnung* und der Einsatz von *partiellen* Funktionsapplikationen verstärkt zum Tragen, was funktionale Sprachen von imperativen deutlich unterscheidet. Die Beschreibung von `ProofType` befindet sich in Abbildung 5.2.

5.4 Wichtige Funktionen

Im folgenden werden die wichtigsten Funktionen beschrieben, die während der Überprüfung eines Beweises zum Einsatz kommen. Auf die Darstellung der Funktionen, die

[4]in der Implementierung gibt es noch weitere Konstruktoren, die das *Pattern-Matching* und die Auswertung von Bedingungen betreffen

```
type ProofType =
  [DoTest]          die Liste der zu bearbeitenden Teilprobleme
  -> Log            Protokollierung der durchgeführten Aktionen
  -> [DerefVar]     die noch zu bearbeitenden Rücksetzpunkte
  -> Info           Information über die zuletzt benutzte Regel
  -> UModes         Modus, in dem sich der Verifikationsprozeß befindet
  -> EnvId          fortlaufender Wert für die Vergabe von Identifiaktoren
  -> BFCT           type BFCT =
                      (Bindings,([EnvId],[EnvId]),[Condition],ETypes)
                    enthält die Bindungen und Bedingungen, die während
                    der Überprüfung gewonnen wurden. In ETypes befin-
                    det sich die Typ-Information derjenigen Variablen, die
                    durch die Regelanwendungen neu hinzukommen. Diese
                    wird später für die externe Typ-Überprüfung benötigt.
                    Die Menge der freien und bereits gebundenen VarFree-
                    Variablen wird in ([EnvId],[EnvId]) aufgeteilt.
  -> Declarations   Struktur, die alle – lokale und toplevel – Deklarationen
                    enthält
  -> [RULE]         Liste von Regeln, die beim Auflösen von Konfliktsitua-
                    tionen verwendet werden können
  -> Result         das Ergebnis des Verifikationsprozesses
```

Abbildung 5.2: `ProofType`

die Benutzeroberfläche betreffen oder die Eingabedaten so aufbereiten, daß die Verifikation durchgeführt werden kann, wird hier verzichtet.

Mit `expCompare` wird eine Liste von Teilaufgaben behandelt, die aus Datenelementen vom Typ `DoTest` besteht. So ist diese Funktion u.a. für die Normalisierung, für die Bindung von freien Variablen und für die Zerlegung in weitere Teilprobleme verantwortlich. Dazu werden entsprechende Hilfsfunktionen aufgerufen. Die Funktion `replaceLocals` ersetzt während der Normalisierung nur lokale Namen. Mit `replaceTopLevel` werden hingegen auch *toplevel*-Funktionen berücksichtigt. Für diese beiden Funktionen gilt, daß sie die Umformungen von der Funktion `getValues` durchführen lassen. Die Abhängigkeiten, die zwischen den hier beschriebenen Funktionen bestehen, lassen sich wie in Abbildung 5.3 darstellen. `getValues` ist dafür verantwortlich, daß eine rekursive Definition zunächst nur *einmal* verwendet und bei der allgemeinen Auswertung die Tiefenbeschränkung eingehalten wird. Dies gewährleistet einen sicheren Zustand des Systems. `replaceTopLevel` leitet die Auswertung der Ausdrücke ein, wenn zuvor keine *toplevel*-Definitionen ersetzt werden konnte.

Die Funktion `failure` ist für das eigentliche *Backtracking* zuständig. Sie läßt, je nachdem in welcher Phase sich der Verifikationsprozeß befindet, mit der Funktion `searchRules` nach geeigneten Regeln suchen. `searchRules` nimmt nur eine grobe Vorauswahl unter den Regeln in der Datenbasis vor, die für das Konfliktpaar passen könnten. In der Funktion `nextRule` werden diese Regeln dann nacheinander auf ihre Verwendbarkeit hin untersucht. Wenn eine der Regeln dem ersten Anschein nach paßt, wird die Beweisprüfung unter dieser Regelanwendung weitergeführt. Erst wenn sich später herausstellt, daß es mit dieser Regel zu Konflikten kommt, da sich etwa unvereinbare Variablenbindungen ergeben, probiert `nextRule` – wieder über einen Aufruf von `failure` – als nächstes eine weitere Regel von denjenigen, die zuvor von `searchRules` als mögliche

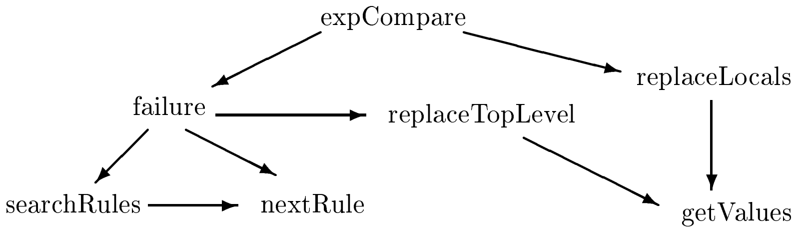


Abbildung 5.3: Zusammenspiel der Funktionen

Kandidaten festgestellt wurden.

5.4.1 `expCompare`

Die Funktion `expCompare` ist für die Bearbeitung der Teilprobleme zuständig, die in der Liste `[DoTest]` gegeben ist.
`expCompare :: ProofType`

- `(Utility util : dotest)`

  Es wird die Aktion durchgeführt, die mit `util` gegeben wird, und im Erfolgsfall wird mit der Bearbeitung der restlichen Teilprobleme in `dotest` fortgefahren. Mit der Aktion `util` werden im wesentlichen Konsistenzbedingungen überprüft und u.a. darauf geachtet, ob die Prämisse einer Regel eingehalten wird.

- `(OnReplace { } : dotest)`

  Der am Kopf der Liste stehende Rücksetzpunkt wird nicht mehr benötigt und es kann mit der Verifikation von `dotest` fortgefahren werden.

- `(OnApply` $\texttt{exp}_1$ $\texttt{exp}_2$ `_ : dotest)`

  Handelt es sich bei $\texttt{exp}_1$ und $\texttt{exp}_2$ um die speziell markierte Anwendung einer Induktionshypothese (vgl. dazu 6.2.1)?
  Ja: Diese Anwendung wird im Beweis vermerkt und anschließend `dotest` bearbeitet.
  Nein: Es wird mit `replaceLocals` die grundlegende Normalisierung in Auftrag gegeben.

- `(OnSuccess` $\texttt{exp}_1$ $\texttt{exp}_2$ `_ : dotest)`

  Handelt es sich bei einem der Ausdrücke um eine freie `(VarFree)` Variable?
  Ja: Die Variable wird an den entsprechenden Ausdruck gebunden, sofern der *occur check* erfolgreich verlaufen ist.
  Nein: Sind die Ausdrücke einfach genug aufgebaut, um zu entscheiden, ob sie gleich sind?
    Ja: Bei Gleichheit wird mit der Überprüfung von `dotest` fortgefahren, ansonsten wird die Fehlerbehandlung `failure` aufgerufen.
    Nein: Das Problem wird in kleinere Teilprobleme aufgespaltet.

- `(Succeeded i : dotest)`

  Mit `Succeeded` werden die einzelnen Beweisschritte abgegrenzt, die vom Benutzer vorgegeben wurden. Wenn dieses „Teilproblem“ erreicht wird, kann ein Teilerfolg signalisiert werden. Der Benutzer wird darüber informiert, daß noch `i` Schritte nachgewiesen werden müssen, was als Fortschrittsindikator eingesetzt wird. Außerdem wird der Verifikationsmodus wieder auf die Verwendung von Regeln zurückgesetzt, denn nach einer erfolgreichen Regelanwendung wurde die Benutzung von Regeln kurzzeitig außer Kraft gesetzt.

- `[]`

  Alle Probleme konnten erfolgreich gelöst werden und somit ist die Gültigkeit des Beweises unter den einschränkenden Bedingungen gezeigt, die während der Verifikation gefunden wurden. Die Bearbeitung dieses Lösungsweges ist also abgeschlossen.

- *Alle anderen Fälle*

  Die Gleichheit der Ausdrücke läßt sich nicht direkt zeigen und es muß versucht werden, eine andere Lösungsvariante zu finden. Dafür wird die Funktion `failure` verwendet, die im Anschluß beschrieben wird.

5.4.2 `failure`

Wie bereits unter [5.3.2] angesprochen, hat die Funktion `failure` im Fehlerfall die Aufgabe, einen geeigneten Rücksetzpunkt zu finden und die Suche nach einem anderen Lösungsweg zu starten.

```
failure::[DoTest]->Log->[DerefVar]->Declarations->[RULE]->Result
```

```
failure [] log [] = \_ _ -> UnifyFailed log
```
Es gibt keine weiteren Rücksetzpunkte mehr und damit besteht keine Möglichkeit, die Gültigkeit des Beweises noch zu zeigen.

```
failure dotest log bkpts
```

Aus `dotest` und `bkpts` wird der Rücksetzpunkt bestimmt, der als nächstes zu verwenden ist. Da Rücksetzpunkte immer mit `OnReplace { }` beginnen, wird dazu das in diesem Konstruktor befindliche Attribut `u_ID` verwendet. Der Punkt mit der höchsten Identifikator-Nummer wird als nächstes abgearbeitet.

Bezeichnet jetzt `back@(OnReplace { u_mode } : dotest)` diesen neuen Aufsetzpunkt, dann ergibt sich der weitere Verlauf in Abhängigkeit von dem Modus `u_mode`.

- `U_UseRules`

  Wenn die Konfliktsituation durch eine Regelanwendung aufgelöst werden darf, wird das Problem an die Funktion `searchRules` deligiert.

- `U_OtherRules rules`

  Ein zuvor gestarteter Versuch, eine Regel anzuwenden, ist gescheitert. Es können allerdings noch die in `rules` angegebenen Regeln auf ihre Tauglichkeit hin getestet werden. Dazu wird die Funktion `nextRule` in Anspruch genommen, die in Abschnitt 5.4.6 besprochen wird.

- `U_NoRules`

  Sind keine Regelanwendungen erwünscht, wird sofort die Ersetzung von *toplevel*-Deklarationen mit der Funktion `replaceTopLevel` eingeleitet.

- `U_Eval`

  Dieser Modus besagt, daß die Auswertung der Ausdrücke bereits gescheitert ist. Aus diesem Grund wird überprüft, ob die vorherrschende Konfliktsituation durch eine Zusatzbedingung beseitigt werden kann, die dann zu den bereits bestehenden Bedingungen hinzuzufügen ist. Falls dies nicht möglich ist, wird mit `failure` nach weiteren Rücksetzpunkten gesucht und abgearbeitet.

5.4.3 replaceLocals

Die Umsetzung der *Normalisierung I* [4.3.3] erfolgt durch die Funktion `replaceLocals`. Es werden bei ihr lediglich lokale Deklarationen ersetzt und jede solche Deklaration wird anschließend markiert, damit sie zunächst von einer weiteren Ersetzung ausgeschlossen wird. Andernfalls würde ein Ausdruck der Form `let f=f in f` zur Nichttermination des Systems führen. Die eigentliche Umformung der Ausgangsterme führt jedoch die Funktion `getValues` durch, die von `replaceLocals` aufgerufen wird.

5.4.4 replaceTopLevel

Die Funktion `replaceTopLevel` beschreibt die Bearbeitung der Teilprobleme `NormII` und `Evaluate` aus Abschnitt 4.3.7 und 4.3.8. Es werden hier zum einen die Namen von *toplevel*-Funktionen durch den an sie gebundenen Wert ersetzt und zum anderen, wenn dies nicht möglich ist, da z.B. die entsprechenden Deklarationen bereits verwendet und damit markiert wurden, wird eine komplette Auswertung vorgenommen. Auch hier wird die eigentliche Ersetzung von der Funktion `getValues` übernommen, die u.a. darauf achtet, daß die Tiefenbeschränkung bei der Bearbeitung der Probleme eingehalten wird.

5.4.5 searchRules

In der Funktion `searchRules` wird eine Vorauswahl unter den Regeln getroffen, die bei der Suche verwendet werden sollen. Nur wenn prinzipiell die Möglichkeit besteht, die beiden Seiten der Regeln und die in Konflikt stehenden Ausdrücke zur Deckung zu bringen, wird eine genauere und damit aufwendigere Analyse in Betracht gezogen.

Die Arbeitsweise von `searchRules` läßt sich grob in zwei Schritte zusammenfassen.

1. Unter allen Regeln werden diejenigen bestimmt, die im weiteren Verlauf näher untersucht werden sollen. Ein Kriterium hierfür sind z.B. die Funktionen, die ganz links in einer Applikation stehen (vgl. `f` in Abb. 5.1). Bezeichnen wir dieses Zwischenresultat mit *testrules*.

2. Wie in der Funktion `failure` wird auch hier die eigentliche Regelanwendung mit der Funktion `nextRule` durchgeführt. Mit dem Aufruf von

   ```
   nextRule (OnReplace { u_mode=U_OtherRules testrules } : ...)
   ```

   werden die in *testrules* gegebenen Regeln auf ihre Brauchbarkeit hin überprüft.

5.4.6 nextRule

Die Funktion `nextRule` erhält eine Liste von Regeln, die auf ein Konfliktpaar von Ausdrücken angewendet werden sollen. Zunächst wird für die erste Regel dieser Liste untersucht, ob ihre beiden Seiten mit den zu vergleichenden Ausdrücken in Übereinstimmung gebracht werden kann und ob die zugehörige Prämisse gültig ist. Die restlichen folgen bei Bedarf, wenn sich herausstellt, daß die gewählte Regel nicht verwendet werden kann. Wenn man z.B. die Regel[5]

```
{-Name-}   arithPlusMult
{-Types-}  Num a => x :: a , a
{-Steps-}
           x+x {-===-} 2*x
```

betrachtet, in der die Variable `x` bei der Anwendung mit jedem beliebigen numerischen Wert belegt werden darf, muß dafür gesorgt werden, daß eine mehrmalige Anwendung der Regel `arithPlusMult` keine Konflikte bei der Bindung von `x` verursacht. Aus diesem Grund werden die freien Variablen einer Regel – sie folgen hinter dem `{-Types-}`-Konstrukt – gewissermaßen umbenannt, um solche Konflikte zu vermeiden.

Der Aufbau von `nextRule` kann in etwa so beschrieben werden:

```
nextRule (OnReplace { u_Success = OnSuccess { u_exp1 u_exp2 _ },
                      u_mode = U_OtherRules (rule:rules) } :
          dotest ) =
```

1. Für die Variablen in `rule` werden neue `VarFree`-Variablen erzeugt.

2. Die genaue Analyse wird durch den Aufruf von `expCompare` erledigt.

   ```
   expCompare (
      OnApply u_exp1 (linke Seite von rule) :
      OnApply u_exp2 (rechte Seite von rule) :
      Utility (Prüfe die Prämisse von rule) :
      OnReplace { u_mode = U_OtherRules rules } :
      dotest )
   ```

   Nachdem sichergestellt wurde, daß die Seiten der Regel passen, wird mittels `Utility` untersucht, ob auch ihre Voraussetzung erfüllt ist. Wenn sich dabei herausstellt, daß die Prämisse nicht eingehalten wurde, werden über die Fehlerbehandlung `failure` die restlichen Kandidaten in `rules` behandelt.

Falls keine der Regeln paßt, die zu Beginn von `searchRules` vorgeschlagen wurden, gelangt man schließlich zu folgender Situation.

```
nextRule (OnReplace { u_mode = U_OtherRules [] })
```

Damit beginnt die nächste Stufe des Verifikationsprozesses. Das bedeutet in unserem Fall, daß auch *toplevel*-Funktionen ersetzt werden dürfen. Daher erfolgt der Aufruf von `replaceTopLevel`, der somit an dieser Stelle die Phase der Regelanwendung abschließt.

[5] die Beschreibung der Eingabe-Syntax folgt im nächsten Kapitel

5.4.7 `getValues`

Die Funktion `getValues` ist für die Umformung von Ausdrücken verantwortlich, d.h. sie ersetzt:

- `VarDefined`-Variablen, die Deklarationen entsprechen
- `VarLambda`-Variablen, die ihren Wert durch eine β-Reduktion erhalten haben
- `VarFree`-Variablen, die im Rahmen der Unifikation gebunden wurden

Zusätzlich übernimmt sie die Aufgabe, *Pseudo*-Argumente einzufügen, d.h. bei einer unvollständigen (partiellen) Funktionsapplikation werden für die fehlenden Argumente „virtuelle" Parameter hinzugefügt, um den Vergleich der beiden Ausdrücke auf die Funktionsrümpfe zurückzuführen. Sowohl `replaceLocals` als auch `replaceTopLevel` greifen auf `getValues` zurück.

Wenn sich bei der Bearbeitung keine Änderungen der Ausgangsterme einstellen, wird dies durch die Rückgabe `Nothing` signalisiert. Andernfalls enthält der Konstruktor `Just` die transformierten Ausdrücke.

`getValues ::`

`Bool`	`True`: führe Normalisierung durch `False`: Normalisierung wurde bereits durchgeführt
`-> Bindings`	Bindungen der `VarFree`-Variablen
`-> Declarations`	die Menge aller Deklarationen
`-> ProofType`	Funktion, die das *Pattern-Matching* durchführt
`-> (ResExp,ResExp)`	die zu verändernden Ausdrücke
`-> Maybe (ResExp,ResExp)`	Initialwert für die Rückgabe
`-> STATUS (Maybe (ResExp,ResExp))`	das Resultat wird in der Monade `STATUS` zurückgegeben

`getValues off bind decls pM (exp1,exp2)`

1. Wenn *Pseudo*-Argumente eingefügt werden können, wird dies gemacht, andernfalls wird sofort bei 2. fortgefahren.

2. Jeder der beiden Ausdrücke wird dann unabhängig voneinander nach folgenden Gesichtspunkten bearbeitet, wenn er die angegebene Gestalt zeigt:

   `ResVar _ VarFree fId`

   Wenn diese Variable bereits gebunden wurde, befindet sich ihr Wert in dem Argument `bind`. Falls sie noch frei ist, wird `Nothing` zurückgegeben.

   `ResVar env VarLambda lId`

   In der Umgebung `env` ist der Wert der `VarLambda`-Variablen mit Identifikator `lId` abgelegt, die durch eine frühere β-Reduktion gebunden wurde. Diese Ersetzung entspricht gewissermaßen der expliziten Substitution, die bei der Definition der β-Reduktion in 2.2.1 angegeben wurde.

   `ResVar _ VarDefined dId`

   Wenn eines der Kriterien für die Ersetzung einer Deklaration erfüllt ist, d.h

   - eine Deklaration nicht schon früher verwendet wurde oder
   - *toplevel*-Deklarationen ersetzt werden dürfen oder

- im Auswertungsmodus die Anzahl der zur Verfügung stehenden Schritte noch nicht ausgeschöpft wurde,

wird der entsprechende Wert verwendet, der in dem Argument `decls` zu finden ist, das `getValues` übergeben wird. Nur wenn keine dieser Bedingungen erfüllt ist, wird `Nothing` zurückgegeben.

`ResExp env (ResApply a) (f:args)`

Handelt es sich bei `f` um eine λ-Abstraktion, dann wird deren formaler Parameter an den ersten Ausdruck in `args` gebunden und diese Bindung in `env` aufgenommen. Die Bearbeitung wird auf dem neu entstehenden Ausdruck fortgesetzt, der sich aus dem Rumpf der Applikation und den restlichen Elementen in `args` ergibt.

`ResExp env ResCase (exp:alts)`

Bei einer `case`-Anweisung wird versucht, diese komplett auszuwerten, da das Resultat auf jeden Fall einfacher aufgebaut ist als der Ausgangsterm und damit leichter zu handhaben ist. Für das *Pattern-Matching* wird die Funktion `pM` verwendet, die als Argument übergeben wurde. Paßt eine der Alternativen in `alts` auf `exp`, dann wird der entsprechende Ausdruck mit dem Konstruktor `Just` geliefert. Da sich nicht alle `case`-Anweisungen an dieser Stelle auswerten lassen, z.B. wenn der Ausdruck `exp` eine freie Variable oder ein Symbol repräsentiert, wird in diesen Fällen `Nothing` zurückgegeben.

Wenn sich einer der beiden Ausdrücke durch die oben angegebenen Regeln modifizieren läßt, d.h. man für höchstens einen das Ergebnis `Nothing` erhält, beginnt die Bearbeitung wieder bei Punkt 1. Zuvor aber wird dieses Zwischenergebnis als „erfolgreich" vermerkt. Erst wenn sich keine Änderungen mehr ergeben, wird das letzte erfolgreiche Ergebnis als Resultat von `getValues` zurückgeliefert. Falls sich die Ausdrücke nie modifizieren ließen, erhält man `Nothing`.

5.5 Vom System verwendete Dateien

Das System benötigt für einen korrekten Ablauf vier Dateien:

`.tcheck`	Shell-Skript für den Aufruf der externen Typ-Überprüfung
`.check.hs`	enthält die zu überprüfende Typ-Information
`.error.hs`	Ausgabe der externen Typ-Überprüfung
`.proof.prj`	enthält die vom Benutzer angegebenen Dateien, die bei der Verifikation behandelt werden [6.1.1].

Die Skript-Datei `.tcheck`, die vorgegeben und für die Verwendung von *hugs* abgestimmt ist, muß über den Pfad der ausführbaren Programme (vgl. PATH-Umgebungsvariable in UNIX) erreichbar sein, sonst bricht das System mit einer Fehlermeldung ab. Die restlichen Dateien werden während des Betriebs erzeugt.

Kapitel 6

Bedienung des Systems

Nachdem man einen ER-Beweis ausgearbeitet hat, sollte man vorsichtshalber nochmal überprüfen, ob alle Transformationen korrekt sind. Bevor man das Ganze per Hand durchführt, ist es sicherer und bequemer, die Überprüfung automatisch ablaufen zu lassen. Was muß nun aber alles definiert und dem System übergeben werden? In vielen Fällen sind zwei Dateien völlig ausreichend, und zwar:

- Beweisdatei
- Regeldatei

In der Beweisdatei befindet sich der Beweis, der vom System verifiziert werden soll, und die Regeldatei dient dazu, eine Menge von Regeln zur Verfügung zu stellen. Das System muß ja die Regelanwendungen des Benutzers innerhalb der Beweisdatei nachvollziehen können. Jedesmal wenn die Unifikation von (Teil-)Ausdrücken scheitert, wird nach einer passenden Regel gesucht.

Obwohl man mit den beiden angegebenen Dateien bereits sehr viele Beweise verifizieren kann, erlaubt das System noch eine flexiblere Beweisführung. Dazu dienen:

- Deklarationsdatei
- Importdatei

In der Deklarationsdatei gibt man einfach diejenigen Funktionen an, die während der Beweisprüfung ausgewertet werden dürfen, wenn sich die Gleichheit nicht direkt zeigen läßt. Das bringt u.a. den Vorteil mit sich, daß an der einen oder anderen Stelle Zwischenschritte im Beweis eingespart werden können. Bei der Auswertung wird die Haskell-Semantik zugrunde gelegt. Ferner dürfen sich durchaus freie Variablen und Symbole im Ausdruck befinden, ohne daß die Auswertung damit von vornherein zum Scheitern verurteilt ist.

Für die Verwendung von Symbolen kommt die Importdatei ins Spiel, denn dort müssen sie für die externe Typüberprüfung bekannt gemacht werden. Ein Symbol steht für einen Ausdruck, dessen Wert für den Benutzer uninteressant ist. Daher bekommt er die Möglichkeit, für diesen Ausdruck einen symbolischen Namen einzuführen.

Welche Resultate können sich nun während der Beweisverifikation ergeben? Das System unterscheidet prinzipiell drei Fälle:

- Beweis ist gültig:
  Man bekommt etwaige Bindungen von anfangs freien Variablen und die verwendeten Regeln zusammen mit ihren einschränkenden Prämissen angezeigt.

- Beweis ist definitiv falsch, z.B die Gleichung `1 {-==-} 2`
- Beweis ist nicht vollständig nachvollziehbar:
  Neben den Bindungen und Regelanwendungen gibt das System diejenigen (Teil-)Ausdrücke an, deren Gleichheit mit der bis dahin angegebenen Information nicht nachgewiesen werden konnte. Diese fehlenden Gleichungen werden zunächst als Zusatzbedingungen angenommen und die Verifikation unter diesen zusätzlichen Voraussetzungen weitergeführt.

Den letzten Fall betreffend ist nun der Benutzer gefordert, die fehlende Information bereitzustellen. Er kann:

- eine neue Regel spendieren
- zusätzliche Zwischenschritte einfügen
- die Zusatzbedingungen verwerfen

Was die ersten beiden Alternativen betrifft, so läßt sich dies durch Modifikation der betreffenden Dateien, d.h. Beweis- oder Regeldatei, erreichen. Danach wird die Prüfung einfach nochmals durchgeführt. Die dritte Möglichkeit kann dem System direkt mitgeteilt werden, indem der *Backtracking*-Mechanismus verwendet wird [6.2.5]. Der Benutzer ist in der Lage, jede Bindung und jede Regelanwendung gezielt rückgängig zu machen. Das gilt auch für gültige Beweise. Damit kann auf einfache Art und Weise nach weiteren Lösungswegen gesucht werden. Insbesondere kann mit diesem Mechanismus eine eingefügte Zusatzbedingung verworfen werden.

Sobald man mit einer der gelieferten Lösungen zufrieden ist, kann sie in Form eines annotierten Beweises abgespeichert werden. D.h. das System beschränkt sich nicht nur auf die Überprüfung, ob das Ganze korrekt ist oder nicht, sondern liefert zusätzlich eine „Proof outline“, die alle Bindungen und Regelanwendungen auflistet. Dabei kann man unter all den Lösungen wählen, die im Zuge des *Backtracking* gefunden wurden.

Nachdem die einzelnen Bestandteile grob umrissen wurden, die bei einer Beweisführung zum Tragen kommen, wird jetzt im Anschluß zunächst genauer auf den Aufbau der Eingabedateien eingegangen und zum Abschluß der Ablauf eines Beweises an einem Beispiel demonstriert.

6.1 Aufbau der Dateien

6.1.1 Beweisdatei

In dieser Datei befinden sich die Transformationsschritte, die auf Korrektheit überprüft werden sollen. Man kann in der Beweisdatei gleich mehrere Beweise angeben, aus denen dann nach Aufruf des Systems derjenige ausgewählt werden kann, dessen Überprüfung durchgeführt werden soll. Ein solcher Beweis enthält i.allg. folgende Information:

(1). Name des Beweises

(2). Namen und Typen der freien Variablen (optional)

(3). Typ der in den Beweisschritten auftretenden Ausdrücke

(4). Lokale Bindungen (optional)

(5). Prämisse, unter der der Beweis gültig ist (optional)

(6). Folge von elementaren Beweisschritten (mindestens zwei)

Betrachten wir folgende Beweisdatei, um einen ersten Eindruck zu erhalten.

```
{-Name-}  distrib
{-Types-} Num a =>
          x::a , y::a , z::a ,
          a
{-Steps-}
        id (x*(y+z))
{-==-}  (x*(y+z))
{-==-}  x*y+x*z

{-Name-}  fac
{-Types-} n::Int , Int

{-Locals-}
fac n = foldl (*) 1 [1..n]
{-Cond-}
n>0
{-Steps-}
fac n {-==-} n * fac (n-1)
```

Wie man an der abgebildeten Beweisdatei leicht erkennen kann, enthält sie zwei zu prüfende Beweise. Durch das Schlüsselwort `{-Name-}` wird die Angabe eines Beweises eingeleitet, was uns hier zu den Beweisen `distrib` und `fac` führt. Der erste besitzt die freien Variablen `x, y` und `z`, die alle denselben Typ haben und zwar einen aus der Klasse `Num`. Diese Information kann man direkt aus der hinter `{-Types-}` angegebenen Liste entnehmen. Das letzte Element der Liste ist dabei speziell zu interpretieren, denn es gibt nicht den Typ einer freien Variablen an, sondern legt den Typ fest, den die einzelnen Ausdrücke in den Beweisschritten haben. Die Beweisschritte selbst folgen hinter `{-Steps-}` und werden durch `{-==-}` voneinander getrennt. In unserem Fall bedeutet dies, daß `id (x*(y+z)), (x*(y+z))` und `x*y+x*z` den Typ `Num a => a` haben, also denselben Typ aufweisen wie die freien Variablen. Das Eingabeformat eines Beweises ist ziemlich einfach gehalten und umfaßt in der Regel nur die Abschnitte `{-Name-}`, `{-Types-}` und `{-Steps-}`. Nur wenn man noch eine Prämisse angeben muß, unter welcher der Beweis gültig ist, benötigt man den Abschnitt `{-Cond-}`. Er beinhaltet einen booleschen Ausdruck, der erfüllbar sein muß, wenn die Beweisprüfung erfolgreich abgeschlossen werden soll. Mit `{-Locals-}` wurde noch eine Möglichkeit geschaffen, für den Beweis lokale Definitionen anzugeben. Diese erstrecken sich über die Ausdrücke in `{-Cond-}` und `{-Steps-}`. Im Beweis `fac` wurde dies alles exemplarisch aufgenommen.

6.1.2 Regeldatei

Die Regeldatei ist analog zur Beweisdatei aufgebaut. Denn die einzelnen Regeln werden ebenfalls durch `{-Name-}`, `{-Types-}`, `{-Locals-}`, `{-Cond-}` und `{-Steps-}` definiert. Der Name der Regel, der in der Beweisausgabe erscheint, wird hinter `{-Name-}` angegeben. Der erste und letzte Ausdruck der hinter `{-Steps-}` folgenden Ausdrücke, legt die linke und rechte Seite einer Regel fest. Alle dazwischenliegenden werden ignoriert.

Ferner ist zu betonen, daß die Regeln ungerichtet sind. Das bedeutet, beide Richtungen werden bei der Suche nach einer passenden Regel berücksichtigt. Der unter `{-Cond-}` angegebene Ausdruck muß erfüllbar sein, um eine Regel erfolgreich anwenden zu können. Sollte sich im Verlaufe der Beweisprüfung herausstellen, daß die Bedingung nicht eingehalten werden kann, führt dies nachträglich zum Scheitern der Regelanwendung und es wird im Beweis an die Stelle zurückgekehrt, an der die – nun gescheiterte – Regel angewendet wurde, um eine andere Alternative zu suchen.

Betrachten wir die im vorherigen Abschnitt gezeigte Beweisdatei als Regeldatei, so erhalten wir folgende zwei Regeln:

Regel	linke Seite		rechte Seite	Bedingung
`distrib`	`id (x*(y+z))`	≡	`x*y+x*z`	
`fac`	`fac n`	≡	`n * fac (n-1)`	`n>0, fac n = foldl (*) [1..n]`

Wenn eine Funktion, die in Pattern-Schreibweise definiert wurde, in einzelne Regeln aufgeteilt werden soll, muß darauf geachtet werden, daß die Pattern disjunkt sind. Es sind die Einschränkungen aus Abschnitt 2.4.3 zu beachten, um ein korrektes Regelsystem zu erhalten.

6.1.3 Deklarationsdatei

In dieser Datei werden diejenigen Funktionen angegeben, die während der Beweisprüfung vom System ausgewertet werden dürfen. Dazu wird die entsprechende Funktion in Haskell-Syntax in der Deklarationsdatei abgelegt. Das System ist damit u.U. in der Lage, Ausdrücke weiter zu vereinfachen, wenn Unifikation und Regelanwendung allein nicht zum Ziel führen, die Gleichheit nachzuweisen. Es können dann Ausdrücke mit derartigen Funktionen schrittweise ausgewertet werden. Sehen wir uns dazu folgende Datei an:

```
import MyDefs

infixl << 5

(<<) :: a -> b -> a
x << y = x

map f [] = []
map f (x:xs) = f x : map f xs
```

Als erstes haben wir hier die Anweisung `import MyDefs`. Dadurch werden in Haskell die Definitionen aus der Datei `MyDefs.hs` eingelesen.[1] Somit darf in der Deklarationsdatei auf Funktionen aus `MyDefs.hs` zugegriffen werden. In unserem Fall sind `import`-Anweisungen nur für die externe Typüberprüfung von Bedeutung, für die Verifikation selbst spielen sie keine Rolle. Je nach Datei können `import`-Anweisungen auch weggelassen werden.

Damit der Anwender auch eigene Infix-Operatoren benutzen kann, werden die Haskell-Konstrukte

[1] `import` bezieht sich auf das Modulkonzept von Haskell. Das hier im Beispiel implizit auf `MyDefs.hs` zugegriffen wird stellt einen Spezialfall dar, der von *hugs* unterstützt wird.

- `infixl` (linksassoziativ)
- `infix` (nicht assoziativ)
- `infixr` (rechtsassoziativ)

unterstützt. Mit `infixl << 5` wird `(<<)` als neuer Infix-Operator eingeführt. Er ist linksassoziativ und hat Präzedenz 5. In Haskell bereits vordefinierte Infix-Operatoren, wie z.B. `(*)`, können auf diese Weise nicht redefiniert werden! In der Deklarationsdatei folgt nun die Typ-Signatur von `(<<)` und die dazugehörige Funktionsvorschrift. `(<<)` ist eine Art Projektion, die das erste Argument von zweien zurückliefert. Am Ende wird dann noch eine Definition der bekannten Funktion `map` angegeben. Nehmen wir nun beispielsweise folgendes Problem:

`map (\x->x) [1] {-===-} [1] << undefined`[2]

Die Gleichheit kann in diesem Fall durch einfaches Auswerten nachgewiesen werden. Beide Seiten werden zu `[1]` reduziert und danach führt Unifikation zum Ziel.

Da das System Typen und Klassen nicht direkt berücksichtigt, sondern nur die Korrektheit der Eingabe und Ausgabe extern auf Typkorrektheit prüfen läßt, werden alle typbezogenen Sprachkonstrukte, bis auf Typsignaturen, in dieser Datei nicht unterstützt. Das hat auch zur Folge, daß überladene Funktionen nicht behandelt werden können. In diesem Fall muß man auf das Überladen verzichten und die Funktion für den Datentyp explizit angeben.

6.1.4 Importdatei

Die Importdatei ist nur für die externe Typüberprüfung relevant, das System selbst ignoriert ihren Inhalt, was die Beweisverifikation angeht. Da alle Eingabedateien auf Typkorrektheit überprüft werden, muß man alle verwendeten Funktionen und Datentypen definieren. Was die von Haskell vordefinierten Funktionen und Datentypen betrifft, braucht man sich darum nicht zu kümmern, da diese automatisch bei der externen Typherleitung eingebunden werden. Auch alle Definitionen aus der Deklarationsdatei stehen ohne weiteres zur Verfügung. Was jetzt noch bleibt, sind:

- benutzerdefinierte (*algebraische*) Datentypen
- Funktionssymbole

Die algebraischen Datentypen werden einfach mit dem `data`-Konstrukt in der Importdatei deklariert. Damit sind die Konstruktoren und zugehörigen Typen bekannt. Was ist nun unter einem Funktionssymbol zu verstehen? Wenn man eine nicht vordefinierte Funktion in der Beweis-, Regel- oder Deklarationsdatei verwenden möchte, aber gleichzeitig sicherstellen will, daß diese Funktion nicht bei der Auswertung verwendet wird, darf man sie nicht in der Deklarationsdatei definieren. Andererseits wird diese Information aber für die Typüberprüfung gebraucht. Aus diesem Grund werden solche Deklarationen in der Importdatei abgelegt.

Alle Funktionen, die nicht in der Deklarationsdatei zu finden sind, werden als Funktionssymbole aufgefaßt. Konstruktoren sind in Haskell durch ihre Syntax festgelegt. Sie beginnen entweder mit einem Großbuchstaben oder sie werden durch das Symbol `:` eingerahmt. Also braucht das System keine `data`-Konstrukte auszuwerten, da die Typen

[2] in Haskell vordefiniert, repräsentiert einen undefinierten Wert

für die Beweisprüfung uninteressant sind. Dies alles rechtfertigt die Vorgehensweise, die Information in der Importdatei zu ignorieren.

Sehen wir uns dazu folgendes Beispiel an:

```
data Wochentag = Mo | Di | Mi | Do | Fr | Sa | So

istSonntag So = True
istSonntag _  = False

istWochenende :: Wochentag -> Bool
istWochenende = undefined
```

Hier wird zunächst mit `data` der Aufzählungstyp `Wochentag` eingeführt. Ferner werden die Funktionen `istSonntag` und `istWochenende` definiert. Da diese Datei nur dazu da ist, benötigte Typinformation zu liefern, reicht es völlig aus, nur die Typsignatur zusammen mit einer Pseudo-Deklaration anzugeben. Das kann man bei `istWochenende` ablesen. Wenn man nun die in der Importdatei definierten Funktionen `istSonntag` und `istWochenende` in der Beweis-, Regel- oder Deklarationsdatei verwendet, handelt es sich lediglich um Funktionssymbole. Das bedeutet, der Ausdruck `istSonntag Mo` wird nicht etwa zu `False` reduziert, sondern bleibt unverändert, da ja keine Funktionsvorschrift an `istSonntag` gebunden wurde.

Was die zu verwendenden Sprachkonstrukte in der Importdatei betrifft, gibt es keinerlei Einschränkungen, da diese Datei vom System nicht beachtet wird. Einzig und allein der externe Typ-Checker (*hugs-1.3*), der alle Haskell-Konstrukte unterstützt, wertet die darin enthaltene Information aus.

6.2 Anwendungsbeispiel

Als nächstes soll Schritt für Schritt gezeigt werden, wie man eine vorgegebene Folge von Beweisschritten bzw. Programmtransformationen so umformuliert, daß sie vom System bearbeitet und auf Korrektheit überprüft werden kann. Da das System auf Haskellausdrücken arbeitet, besteht die erste Aufgabe darin, eine abweichende Notation an die Haskell-Syntax anzugleichen. Auf diesen ersten Schritt wird hier nicht näher eingegangen. Ausgangspunkt für unsere Überlegungen ist bereits eine Folge von Haskellausdrücken. Was dann im weiteren zu erfolgen hat, wird exemplarisch an einem Lemma demonstriert, das [28] entnommen wurde. Es soll nun gezeigt werden:

```
Lemma:
        Wenn f assoziativ ist, dann gilt:
        foldl1 f (a:x:xs) = f a (foldl1 f (x:xs))
```

Die Funktion `foldl1` gehört mit zu den wichtigsten Funktionen, die auf Listen arbeiten. Sie leitet sich von der Funktion `foldl` ab, die folgendermaßen definiert ist:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)    = foldl f x xs
```

`foldl` verknüpft sukzessive die Elemente der Liste `xs` mit dem bis dahin berechneten Wert und der Funktion `f`. Am deutlichsten wird die Funktionsweise, wenn man sich die Expansion des Ausdrucks

```
foldl (\b x -> b && even x) True [1,2,3]
```

ansieht, der überprüft, ob alle Listenelemente gerade sind. Durch schrittweise Ersetzung der Funktionsvorschrift (Unfolding) und Ausnutzung des Infix-Operators `(&&)` erhält man:

```
(((True && even 1) && even 2) && even 3)
```

`foldl1` ist eine Spezialisierung von `foldl`. Zu beachten ist, daß `foldl1` nur eine partielle Funktion ist, da sie nicht auf der leeren Liste definiert ist. Ferner ist der Typ der Funktion `f` für die Argumente und das Resultat gleich.

Obiges Lemma besagt nun: man kann den Kopf einer mindestens zwei-elementigen Liste abspalten, auf die `foldl1` angewendet wird und stattdessen `foldl1` auf den verbleibenden Teil der Liste anwenden. Dieses Ergebnis wird dann zusammen mit dem vorher abgespaltenen Element mittels der bei `foldl1` angegeben Funktion verknüpft und das Ergebnis ist gleich der ursprünglichen Variante, sofern diese Funktion assoziativ ist.

6.2.1 Inhalt der Beweisdatei

Der Beweis dieses Lemmas kann mittels Induktion über die Länge der Liste `xs` geführt werden. Das bedeutet, daß der Beweis in einzelne Fälle aufgeteilt werden muß. Im vorliegenden Fall reicht ein Basisfall für `xs = []` und ein Induktionsfall für `xs = (y:ys)`, um die Korrektheit des Lemmas zu zeigen. Je nach zugrundeliegendem Datentyp und Beweis können auch mehrere Basisfälle und Induktionsfälle nötig sein.

Ein Beweis für dieses Lemma könnte nun so aussehen, wenn man ihn per Hand notiert und auf erklärende Erläuterungen zwischen den einzelnen Schritten verzichtet.

```
Basisfall:
  foldl1 f (a:x:[])
   = f a x
   = f a (foldl1 f (x:[]))

Induktionsfall:
  foldl1 f (a:x:y:ys)
   = foldl1 f (f a x:y:ys)
   = f (f a x) (foldl1 f (y:ys))
   = f a (f x (foldl1 f (y:ys)))
   = f a (foldl1 f (x:y:ys))
```

Bevor man nun daran gehen kann, die Beweisskizze in eine für das System brauchbare Form zu transformieren, muß man sich darüber klar werden, welche Variablen im Beweis als freie Variablen auftreten sollen. Hier bieten sich die Variablen `f, a und x` für den Basisfall und `f, a, x, y und ys` für den Induktionsfall an. Ferner muß man den Typ feststellen, den diese freien Variablen haben, damit bei der externen Typüberprüfung die abgeleiteten Typen mit den angegeben verglichen werden können.

Da das Lemma für Listen beliebigen Typs gilt, erhält man folgende Typsignaturen:

```
f     :: t -> t -> t
a,x,y :: t
ys    :: [t]
```

Basisfall

Nachdem man nun diese Information gesammelt hat, geht man daran, die Beweisdatei zu formulieren. Nehmen wir an, der Beweis soll in der Datei „Beweis.axiom“ abgelegt werden, dann fügt man für den Basisfall in diese Datei folgendes ein.

```
{-Name-}        basic_foldl1_Lemma
{-Types-}       f :: (t -> t -> t)  ,  a :: t  ,  x :: t  ,
                t
{-Steps-}
        foldl1 f (a:x:[])
{-==-}  f a x
{-==-}  f a (foldl1 f [x])
```

Der Name des Basisfalles wurde als „basic_fold1_Lemma“ bezeichnet, er könnte aber durch jeden anderen Haskell-Identifikator ersetzt werden, solange dieser nicht schon für einen anderen Beweis in der Datei „Beweis.axiom“ oder für den Namen einer vordefierten Funktion verwendet wird.

Induktionsfall

Bei Induktionsfällen besteht nun allgemein die Schwierigkeit, dem System mitzuteilen, wo die Induktionsvoraussetzung angewendet werden darf. Im formalen Fall liegt die Situation vor, daß man die Korrektheit des Lemmas für

`length xs = n (n>0)`

nachweisen will unter der Voraussetzung, daß

`foldl1 f (a:x:xs) = f a (foldl1 f (x:xs))`

für `length xs` $<$ `n` gültig ist.

Es wäre zwar nun denkbar, zumindest für Induktionsbeweise über einfache Strukturen, z.B. Listen, eine boolesche Funktion anzugeben, die entscheidet, ob die Hypothese an den in Frage kommenden Stellen anwendbar ist oder nicht. Aber dazu müßte der Benutzer i.allg. auch Beziehungen zu symbolischen Werten herstellen, wie z.B. daß `length xs = n` zu setzen ist. Dann könnte das System u.U. selbständig über die Anwendbarkeit einer Induktionsvoraussetzung entscheiden, ohne daß im Beweis noch Prädikate übrigbleiben, die nicht ausgewertet werden können. Da ein solches Vorgehen selbst bei einfachen Beweisen ziemlich aufwendig werden kann, wird diese Form des Induktionsbeweises nicht unterstützt.

Selbst der Programmieraufwand der für einige Spezialfälle nötig wäre, rechtfertigt an sich nicht den Nutzen, den man daraus ziehen könnte.

Stattdessen wurde eine sehr einfache Methode gewählt, dem System mitzuteilen, wo eine Induktionshypothese angewendet werden soll. Die betreffenden Stellen werden nämlich durch das Schlüsselwort `hypo` gekennzeichnet. Trifft nun das System beim Verifizieren auf das Teilproblem

`hypo exp1 = hypo exp2`,

dann wird die Gleichheit von `exp1` und `exp2` ungeprüft angenommen. Die Anwendung der Hypothese wird aber bei der Ausgabe des Beweises vermerkt, so daß keine Information verloren geht und die einzelnen Transformationsschritte im Beweis transparent bleiben. Das Schlüsselwort `hypo` entspricht, wenn man es als Haskell-Funktion auffaßt, der identischen Abbildung. Somit wird ein Vergleich von `hypo exp1 = exp2` implizit auf das Problem `exp1 = exp2` reduziert. Wenn man nun obige Konventionen zugrunde legt, läßt sich der Induktionsfall folgendermaßen formulieren:

```
{-Name-}        induct_foldl1_Lemma
{-Types-}       f :: (t->t->t) , a :: t , x :: t , y :: t ,
                ys :: [t] , t
{-Steps-}
        foldl1 f (a:x:y:ys)
{-==-}  hypo (foldl1 f (f a x : y : ys))
{-==-}  hypo (f (f a x) (foldl1 f (y:ys)))
{-==-}  f (f a x) (foldl1 f (y:ys))
{-==-}  hypo (f a (f x (foldl1 f (y:ys))))
{-==-}  hypo (f a (foldl1 f (x:y:ys)))
```

6.2.2 Inhalt der Deklarationsdatei

Die Angabe einer Beweisdatei allein reicht aber in der Regel nicht aus, um eine Beweis-Überprüfung sinnvoll durchzuführen. Denn in diesem Fall kommen nur vordefinierte Basistransformationen zum Zuge, wie etwa das Auflösen von λ-Abstraktionen und `let`-Ausdrücken. Komplexere Transformationen führen dazu, daß die Gleichheit zweier Teilausdrücke als Kontext-Bedingung angenommen und im Beweis vermerkt wird. Die Verifikation wird dann unter dieser Bedingung fortgesetzt. Ohne Angabe von weiterer Information, kann man also davon ausgehen, daß sehr viele solcher Kontext-Bedingungen eingefügt werden, was zu vermeiden ist.

Im obigen Beispiel erscheint es sinnvoll, dem System die Funktionsvorschrift von `foldl1` anzugeben. Dies kann auf zwei verschiedene Arten geschehen, und zwar als

- Regel
- Funktionsdeklaration

Der Unterschied besteht darin, daß bei der Regel-Variante die Anwendung von `foldl1` explizit in der Beweisausgabe erscheint, indem eben der Name der Regel angegeben wird. Bei der anderen Variante erhält man keine entsprechende Ausgabe, da die Ersetzung von `foldl1` durch den Funktionsrumpf mit zur Standard-Normalisierung gerechnet wird. Außerdem ist zu beachten, daß zwischen zwei Transformationsschritten höchstens eine Regel angewendet werden darf. Damit müssen die einzelnen Schritte in der Beweis-Datei bei der Regel-Variante dementsprechend fein abgestuft werden. Wenn die Definition als Haskell-Funktion vorliegt, entfällt i.allg. eine solche feine Unterteilung, da die mehrmalige Anwendung der Funktion, auch im Zusammenspiel mit anderen Funktionen, über die Auswertung von Haskell-Ausdrücken während der Verifikation abgedeckt wird. Die Auswahl einer Alternative bleibt also dem Benutzer überlassen. Man kann auch beide Varianten gleichzeitig wählen, was aber in der Regel beim *Backtracking* zu mehr Lösungen führt, da zwischen der Regelanwendung und der Auswertung unterschieden wird, obwohl semantisch dasselbe beschrieben wird.

Der Einfachheit halber wird hier der Ansatz mit der Funktionsdeklaration gewählt, da man in diesem Fall den Beweis kürzer halten kann. Dazu fügt man beispielsweise in die Datei „Deklaration.hs“ die Definitionen der Funktionen `foldl` und `foldl1` aus dem vorherigen Abschnitt ein. Unter [6.2.4] wird gezeigt, wie man dem System mitteilt, daß die Datei „Deklaration.hs“ die Funktionsdeklarationen enthält, die während der internen Auswertung zu berücksichtigen sind.

6.2.3 Inhalt der Regeldatei

Die Definition von `foldl1` reicht aber noch nicht aus, um den Beweis des Lemmas vollständig zu führen. Die Assoziativität der im Lemma von `foldl1` verwendeten Funktion `f` muß noch beschrieben werden. Ohne diese Information kann der entsprechende Transformationsschritt im Induktionsfall nicht aufgelöst werden, was das Einfügen einer Bedingung zur Folge hat, die der Benutzer nachträglich noch verifizieren muß.

Eine Funktion `h` ist assoziativ, wenn gilt:

```
h :: a -> a -> a
h x (h y z) == h (h x y) z ,      ∀ x,y,z mit Typ a
```

Diese Aussage kann man nun in etwas abgewandelter Form als Regel verwenden, was zu folgender Formulierung führt.

```
{-Name-}        assocFun
{-Types-}       h :: (t->t->t) , x :: t , y :: t , z :: t , t

{-Locals-}
isAssoc x = undefined

{-Cond-}
isAssoc h

{-Steps-}
(x `h` y) `h` z {-==-} x `h` (y `h` z)
```

Es ist hierbei zu beachten, daß `h` in der Regel-Definition eine freie Variable ist. Dies erkennt man daran, daß `h` im `{-Types-}`-Abschnitt angegeben wurde. Das bedeutet nun, die angegebene Gleichheit gilt nur dann, wenn die an `h` gebundene Funktion auch wirklich assoziativ ist. Aus diesem Grund wurde die lokale Definition `isAssoc x = undefined` eingefügt und das Prädikat `isAssoc h` als Bedingung mit aufgenommen, unter welcher die Regelanwendung zulässig ist. Dieses Prädikat erscheint dann zusammen mit dem Regelnamen `assocFun` und der aktuellen Bindung für `h` in der Beweisausgabe. Nur wenn diese Regel für eine ganz bestimmte Funktion formuliert wird, wie etwa `(+)`, kann ein solches Prädikat weggelassen werden, da diese Information in der fest vorgegeben Funktion steckt.

Die Vorschrift `isAssoc x = undefined` braucht allerdings nicht lokal definiert zu werden, sondern man könnte sie auch in „Deklaration.hs“ ablegen oder noch besser als Symbol in der Importdatei definieren. Entscheidend ist nur, daß ein derartiges Prädikat nicht auswertbar sein darf, damit es in der Beweisausgabe stehen bleibt, denn sonst treten die folgenden beiden Fälle auf:

- `isAssoc x = True` würde zu `True` reduziert und verschwinden. Der Benutzer erfährt also nicht, daß die in `assocFun` verwendete Funktion assoziativ sein muß.
- `isAssoc x = False` würde die Anwendung der Regel unmöglich machen

Wenn obige Regel-Definition nun etwa in die Datei „Regel.axiom“ eingefügt wird, hat man schließlich alles zusammen, was man zum Verifizieren des Lemmas braucht.

6.2.4 Durchführung des Beweises

Zum Überprüfen der einzelnen Schritte im Lemma wird das System mit `Vera`[3] aufgerufen. Beim allerersten Aufruf erhält man folgende Eingabemaske. Sonst kann hinter „`:`“ der Dateiname erscheinen, der bei der letzten Sitzung eingegeben wurde.

```
Proofs: none
Rules:  none
Decls:  none
Import: none

Enter:
 >P<roofs  >R<ules  >D<ecls  >I<mport  >E<xit  >H<elp
 >U<se  |  Do >N<ot use inserted conditions
> ?
```

Nun werden über die Menü-Punkte `P`, `R` und `D` die Dateien festgelegt, die bei dieser Prüfung zum Einsatz kommen sollen. Dabei sollten die Dateinamen ohne Suffix angegeben werden, also in unserem Fall `Beweis`, `Regel` und `Deklaration`. Andernfalls werden die Endungen automatisch abgeschnitten. Falls man eine Datei nicht mehr berücksichtigen möchte, gibt man unter dem entsprechenden Menü-Punkt den Leer-String als Dateinamen an. Eine „Import“-Datei ist für dieses Beispiel nicht nötig, da keine zusätzliche Information zur Typüberprüfung gebraucht wird. In Anhang A finden sich Beispiele, in denen auch diese Datei Anwendung findet. Nachdem man nun die Dateinamen übergeben hat, die die notwendige Information enthalten, kann mit der eigentlichen Überprüfung begonnen werden. Dazu muß aber erst der Inhalt der Dateien aufbereitet werden, damit man die Typkonsistenz der Eingabe feststellen kann. Mit `U` bzw. `N` werden die erforderlichen Transformationen in die Wege geleitet. Der Unterschied von `U` und `N` für die weitere Beweisprüfung liegt darin, daß bei `N` keine vom System eingefügte Gleichung in der Beweisausgabe erscheinen darf. Tritt dieser Fall auf, wird automatisch nach einer anderen Lösung gesucht. Das hat zur Folge, daß bei `N` die Beweisverifikation entweder erfolgreich durchgeführt werden kann oder komplett scheitert. Information, an welchen Stellen im Beweis es Unstimmigkeiten gibt, wird nur mit `U` geliefert. Für die weiteren Betrachtungen wird die Verwendung von `U` vorausgesetzt, um den Einsatz der Backtracking-Komponente besser zeigen zu können. Zur externen Typüberprüfung wird der Haskell-Interpreter *hugs-1.3*[4] verwendet. Wenn dabei keine Fehler auftreten, erscheint ein Menü, das eine Auswahl unter den Beweisen erlaubt, die in der Beweisdatei abgelegt wurden.

```
Choose Proof:
=============

0:      back to main menu
        -----------------
1:      induct_foldl1_Lemma
2:      basic_foldl1_Lemma
> ?
```

Was die Datei „Beweis.axiom“ betrifft, sind das `1) induct_foldl1_Lemma` und `2) basic_foldl1_Lemma`. Bei einem Fehler erhält man die von *hugs-1.3* gelieferte Fehler-

[3] **V**erification of **E**quational **Rea**soning proofs
[4] zu beziehen über `http://www.haskell.org`

meldung und kehrt dann zum Hauptmenü zurück. Nach der Modifikation der entsprechenden Stellen kann erneut mittels `U` bzw. `N` die Typüberprüfung durchgeführt werden. In unserem Beispiel treten keine Fehler auf und man kann sich durch die Wahl von `1)` oder `2)` eine typkorrekte Lösung für den entsprechenden Beweis anzeigen lassen.

6.2.5 Backtracking

Wenn man nun `induct_foldl1_Lemma` als den zu verifizierenden Beweis auswählt, erhält man zunächst das Resultat, wie es in Abbildung 6.1 wiedergegeben ist.

Zwischen den beiden Klammern `{-` und `-}` werden die Bedingungen angegeben, unter welchen der Übergang zwischen zwei aufeinanderfolgenden Beweisschritten durchführbar ist. Also besagt `{- -}`, daß keine weiteren Anforderungen gestellt werden müssen. Die Anwendung der Induktionshypothese ist durch die Funktion `inductive` gekennzeichnet, welche die beiden Teilausdrücke zeigt, auf die sie angewendet wurde. Die vom System eingefügten Zusatzbedingungen sind an `equiv` erkennbar. Nun sieht man in Abb. 6.1, daß bei einem Übergang die Bedingung `` f a x `equiv` a `` verwendet wird, welcher die Kennziffer `68` zugewiesen wurde. An dieser Lösung ist man nicht interessiert, weil sie zu spezielle Anforderungen stellt. Das System fordert aber nun dazu auf, beliebige Voraussetzungen zurückzunehmen. Dazu verwendet man die Kennzahlen auf der linken Seite. Durch die Angabe von `b` gelangt man in den Rücksetzmodus, in dem die Kennzahlen angegeben werden können.

```
Backtrack (y/n) [y]

Which Numbers [All]: 68
```

Wenn man die Bedingung mit der Kennzahl `68` zurücksetzt und auf weitere Lösungsvarianten verzichtet, erhält man den im nächsten Abschnitt angegebenen annotierten Beweis.

6.2.6 Ausgabe

Bei der Ausgabe kann man unter den erfolgreich abgeschlossenen Lösungsvarianten wählen, die aufsteigend durchnummeriert werden. Uns bieten sich zwei Möglichkeiten:

```
Print Proof-Outline:
====================

0:      Cancel
        ------
1:      Result 1
2:      Result 2
> ?
```

In unserem Fall war der letzte Versuch der günstigste, also wählen wir die Nummer 2.[5] Dann kann der Benutzer nochmals entscheiden, ob er ausgehend von dieser Lösung andere Varianten mittels `b (backtrack)` ableiten, über `w (write to file)` den Beweis abspeichern oder mit `g (go back)` abbrechen will. Bei `w` wird nach Angabe eines Dateinamens der annotierte Beweis wie in Abb. 6.2 abgespeichert:

Der einzige Unterschied zur Ausgabe am Bildschirm ist, daß keine Kennzahlen für *Backtracking*-Zwecke angegeben werden.

[5] auf den Abdruck der Beweisausgabe am Bildschirm wird hier verzichtet

```
Print Proof-Outline:
====================

0:      Cancel
        ------
1:      Result 1
> ? 1


foldl1 f (a : x : y : ys)
{-
-}
foldl1 f (f a x : y : ys)
{-
  57    foldl1 f (f a x : y : ys) ‘inductive‘
        f (f a x) (foldl1 f (y : ys))
-}
f (f a x) (foldl1 f (y : ys))
{-
-}
f (f a x) (foldl1 f (y : ys))
{-
  68    f a x ‘equiv‘ a
  69    foldl f y ys ‘equiv‘ f x (foldl1 f (y : ys))
-}
f a (f x (foldl1 f (y : ys)))
{-
  70    f a (f x (foldl1 f (y : ys))) ‘inductive‘
        f a (foldl1 f (x : y : ys))
-}
f a (foldl1 f (x : y : ys))

Enter:
 >w<rite to file   >b<acktrack   >g<o back   >h<elp
> ?
```

Abbildung 6.1: Erstes Resultat

```
foldl1 f (a : x : y : ys)
{-
-}
foldl1 f (f a x : y : ys)
{-
  Cond: foldl1 f (f a x : y : ys) ‘inductive‘
        f (f a x) (foldl1 f (y : ys))
-}
f (f a x) (foldl1 f (y : ys))
{-
-}
f (f a x) (foldl1 f (y : ys))
{-
        v70_4   =       f
  Cond: isAssociative_7_1 v70_4
  Rule: assocFun
-}
f a (f x (foldl1 f (y : ys)))
{-
  Cond: f a (f x (foldl1 f (y : ys))) ‘inductive‘
        f a (foldl1 f (x : y : ys))
-}
f a (foldl1 f (x : y : ys))


Local Declarations:
        isAssociative_7_1
         = \l8_1 -> undefined
```

Abbildung 6.2: Annotierter Beweis

Kapitel 7

Schlußbemerkungen

Equational Reasoning ist eine Beweismethode, die sich besonders für rein funktionale Programmiersprachen eignet. In diesen Sprachen dürfen Deklarationen als Gleichungen gelesen werden. Wenn man bei den Deklarationen allerdings Pattern zur Definition von einzelnen Alternativen verwendet, darf man diese Alternativen nur dann als Gleichungen auffassen, wenn sie sich nicht überschneiden. Sonst gelten die Betrachtungen aus Abschnitt 2.4.3. Neben den Gleichungen, die sich auf diese Weise implizit aus der Programmdefinition ergeben, können natürlich Regeln benutzt werden, die sich aus den „primitiven“ Gleichungen von Funktionsdeklarationen ableiten lassen. Die Korrektheit der Regeln läßt sich somit wiederum über Equational Reasoning nachweisen.

Die Äquivalenz zweier Ausdrücke kann durch Equational Reasoning nachgewiesen werden. Das Problem besteht nun darin, wie bei anderen Beweismethoden auch, die geeigneten Beweisschritte bzw. Regeln zu finden, um die Gleichheit zu zeigen. Darüber kann man keine allgemeingültige Aussage treffen, da das Vorgehen sehr von den zugrundeliegenden Ausdrücken abhängt. Was die einzelnen Beweisschritte bzw. Regelanwendungen betrifft, so muß sichergestellt sein, daß diese korrekt sind, damit der Beweis in seiner Gesamtheit gültig ist. Das vorgestellte System beschäftigt sich daher mit der Verifikation von Gleichheitsbeweisen.

Die Sprache Haskell ist ein bedeutender Vertreter der rein funktionalen Programmiersprachen, daher wurde sie als Eingabesprache für das System gewählt. Aus diesem Grund wurde darauf geachtet, möglichst viele der Sprachkonstrukte zu behandeln, die Haskell bietet. Daher braucht man – wenn überhaupt – nur wenige Änderungen an den zu vergleichenden Ausdrücken vorzunehmen, wenn sie in das Eingabeformat übertragen werden.

Der Benutzer ist allerdings gefordert, die einzelnen Beweisschritte so fein abzustufen, daß ein Übergang mit höchstens *einer* Regelanwendung nachzuvollziehen ist. Das hat zur Folge, daß Equational Reasoning Beweise leicht sehr lang und unhandlich werden können. Obwohl das für das System prinzipiell kein Problem darstellt, ist es vor allem bei der Herleitung eines neuen Beweises oft günstiger, diesen zunächst in einzelne Teile zu zerlegen und erst am Schluß diese zu einem Beweis zusammenzufügen. Die Anzahl der Beweisschritte hängt auch davon ab, welche Regeln man zur Verfügung stellt. Verwendet man hier nur Eigenschaften elementarer (vordefinierter) Funktionen, so ist ein Beweis in der Regel wesentlich länger als wenn man Regeln hinzunimmt, die aus diesen Eigenschaften abgeleitet werden können.

Beim Backtracking zeigt sich manchmal das verwirrende Verhalten, daß nach dem Zurücksetzen einer Variablenbindung oder Bedingung scheinbar wieder diesselbe Lösung

```
(a + b) + c
                    ∀x,y,z ∈ ℕ : (x+y)+z ≡ x+(y+z)
a + (b + c)
```

Abbildung 7.1: Unnötiges Einfügen von Gleichungen

geliefert wird. Das Resultat mag zwar übereinstimmen, aber die Wege, die dazu führten, unterscheiden sich. Dieses Verhalten tritt vor allem dann auf, wenn eine Eigenschaft zum einen durch eine Regel beschrieben ist und zum anderen auch direkt über den bereitgestellten Auswertungsmechanismus gezeigt werden kann.

Das Einfügen von Gleichungen seitens des Systems führt manchmal dazu, daß eine vorhandene Regel nicht sofort berücksichtigt wird. Erst wenn der Benutzer diese Gleichung zurücknimmt, kommt es zur Regelanwendung. Zur Verdeutlichung soll die Behandlung der Assoziativität in Abbildung 7.1 dienen. Die Dekomposition führt zunächst zu den Teilproblemen `a≡a+b` und `b+c≡c`. Diese lassen sich in der Regel nicht nachweisen. Daher nimmt das System die beiden Gleichungen als zusätzliche Voraussetzungen hinzu und führt die Verifikation unter den neuen Einschränkungen weiter. Diese Annahmen sind aber zu restriktiv (`b=0`) und unnötig, obwohl sie prinzipiell denkbar sind. Erst wenn der Benutzer eine der beiden Gleichungen im Backtrack-Modus zurücknimmt, kommt es daraufhin zur Regelanwendung der Assoziativität. Ein Vorteil der eingefügten Gleichungen ist aber, daß die Bearbeitungsreihenfolge der einzelnen Teilprobleme eine untergeordnete Rolle spielt. Weist man das System an, keine eingefügte Bedingung in der Beweisausgabe auftreten zu lassen [6.2.4], werden diese Bedingungen automatisch zurückgenommen, ohne daß der Benutzer eingreifen muß.

Bei Induktionsbeweisen wird die Anwendung der Induktionshypothese speziell markiert. Das bewirkt, daß die Gleichheit der markierten Ausdrücke ungeprüft angenommen wird. Die Induktionshypothese wird zwar in der Beweisausgabe vermerkt, aber ihre Angabe sagt nichts darüber aus, ob die Induktionsvoraussetzung erfüllt ist oder nicht. Dafür ist allein der Benutzer verantwortlich.

Durch den Einsatz eines externen Typ-Überprüfers ergibt sich eine gewisse Abhängigkeit von dem dazu verwendeten Programm. Man kann zwar das System – ohne das Quellprogramm zu ändern – ohne Typüberprüfung arbeiten lassen, allerdings ist die Gefahr von Typinkonsistenzen nicht mehr auszuschließen [4.3.11]. Die Typherleitung direkt in das System zu integrieren, wurde verworfen, da es einen großen Aufwand bedeutet hätte, das Typsystem zusammen mit dem Klassenkonzept zu verwirklichen, um Haskell-Konformität zu gewährleisten.

Das System ist in Haskell implementiert und kann mit dem Interpreter *hugs* oder dem Compiler *ghc* verwendet werden.

Als verwandte Arbeit ist das System ERA (Equational Reasoning Assistant) zu erwähnen, das an der Universität von Glasgow entwickelt wird. Dieses System soll den Benutzer bei der Durchführung von Equational Reasoning Beweisen unterstützen. Dabei wird ein (Teil-)Ausdruck innerhalb eines Ausdruckes markiert und eine Regel vorgegeben. ERA überprüft dann, ob eine Seite der Regel mit der markierten Stelle in Übereinstimmung gebracht werden kann und gibt im Erfolgsfall den Ausdruck zurück, in dem der markierte Teil durch die jeweils andere Seite der Regel ersetzt wurde. Über ERA kann man sich bei `http://www.dcs.gla.ac.uk/~andy/era.html` informieren.

Anhang A

Beispiel-Beweise

A.1 foldl1-Lemma

Es wird an dieser Stelle nochmals ein Beweis für das in Kapitel 6 beschriebene Lemma gezeigt, an dem die Durchführung eines Beweises exemplarisch beschrieben wurde. Es soll also nachgewiesen werden:

```
Lemma:
        Wenn f assoziativ ist, dann gilt:
        foldl1 f (a:x:xs) = f a (foldl1 f (x:xs))
```

Im Gegensatz zu der Variante in Kapitel 6 wird `f` nicht als freie Variable behandelt, sondern man verwendet das Symbol `assocFun`, das für eine beliebige assoziative Funktion steht. Weil Symbole benutzt werden, ist für diesen Ansatz auch die Import-Datei wichtig.

A.1.1 Beweisdatei

```
{-Name-}  basic_foldl1_Lemma
{-Types-} a :: All, x::All, All
{-Steps-}
        foldl1 assoc_fun (a:x:[])
{-==-}  assoc_fun a x
{-==-}  assoc_fun a (foldl1 assoc_fun [x])


{-Name-}  induct_foldl1_Lemma
{-Types-} a :: All, x::All, y :: All, ys::[All], All
{-Steps-}
        foldl1 assoc_fun (a:x:y:ys)
{-==-}  hypo (foldl1 assoc_fun (assoc_fun a x:y:ys))
{-==-}  hypo (assoc_fun (assoc_fun a x) (foldl1 assoc_fun (y:ys)))
{-==-}  assoc_fun (assoc_fun a x) (foldl1 assoc_fun (y:ys))
{-==-}  hypo (assoc_fun a (assoc_fun x (foldl1 assoc_fun (y:ys))))
{-==-}  hypo (assoc_fun a (foldl1 assoc_fun (x:y:ys)))
```

A.1.2 Regeldatei

Da die assoziative Funktion `assocFun` fixiert ist, benötigt man bei der Angabe der Regel kein Prädikat `isAssoc`, das in der Lösung von Kapitel 6 notwendig war.

```
{-Name-}   assocFun
{-Types-}  x :: All, y :: All, z :: All, All

{-Steps-}
(x ‘assoc_fun‘ y) ‘assoc_fun‘ z {-==-} x ‘assoc_fun‘ (y ‘assoc_fun‘ z)
```

A.1.3 Deklarationsdatei

```
foldl :: (a->b->a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

foldl1 :: (a->a->a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

A.1.4 Importdatei

In der Importdatei muß der Datentyp `All` und die Funktion `assocFun` eingeführt werden, die man in der Beweis- und Regeldatei verwendet, damit die externe Typüberprüfung durchgeführt werden kann.

```
data All = All

assoc_fun :: (All -> All -> All)
assoc_fun = undefined
```

A.1.5 Ergebnis des Induktionsfalles

```
foldl1 assoc_fun (a : x : y : ys)
{-
-}
foldl1 assoc_fun (assoc_fun a x : y : ys)
{-
  Cond: foldl1 assoc_fun (assoc_fun a x : y : ys) ‘inductive‘
        assoc_fun (assoc_fun a x) (foldl1 assoc_fun (y : ys))
-}
assoc_fun (assoc_fun a x) (foldl1 assoc_fun (y : ys))
{-
-}
assoc_fun (assoc_fun a x) (foldl1 assoc_fun (y : ys))
{-
  Rule: assocFun
-}
assoc_fun a (assoc_fun x (foldl1 assoc_fun (y : ys)))
```

```
{-
  Cond: assoc_fun a (assoc_fun x (foldl1 assoc_fun (y : ys))) `inductive`
        assoc_fun a (foldl1 assoc_fun (x : y : ys))
-}
assoc_fun a (foldl1 assoc_fun (x : y : ys))
```

A.2 Parallelisierung von Divide-and-Conquer

Der Beweis, der im folgenden gezeigt wird, entstammt [15]. Dabei wird die Gleichheit der rekursiven Funktion `dc2a`

```
dc2a :: (PS a -> PS b) -> (PS a -> PS a) -> (PS b -> PS b) ->
        Nat -> PS a -> PS b
dc2a bb dd cc   N   = bb
dc2a bb dd cc (S n) = cc . dmap (dc2a bb dd cc n) (S N) . dd
```

mit der *linear* rekursiven Funktion `it2a`

```
it2a :: (PS a -> PS b) -> (PS a -> PS a) -> (PS b -> PS b) ->
        Nat -> PS a -> PS b
it2a bb dd cc n = down (dmap cc) n . dmap bb n . up (dmap dd) n
```

gezeigt.[1] Der Beweis wird per Induktion über die natürliche Zahl `n` geführt.

A.2.1 Beweisdatei

```
{-Name-}    basic
{-Types-}   bb :: (PS a->PS b),
            dd :: (PS a->PS a),
            cc :: (PS b->PS b),
            xx :: PS a,
            PS b
{-Steps-}
        dc2a bb dd cc N xx
{-==-}  bb xx
{-==-}  bb (id xx)
{-==-}  (bb . id) xx
{-==-}  id ((bb . id) xx)
{-==-}  (id . bb . id) xx
{-==-}  (down (dmap cc) N . bb . id) xx
{-==-}  (down (dmap cc) N . dmap bb N . id) xx
{-==-}  (down (dmap cc) N . dmap bb N . up (dmap dd) N) xx
{-==-}  it2a bb dd cc N xx
```

[1] Die Beschreibung der verwendeten Datentypen und Funktionen findet sich unter A.2.4.

```
{-Name-}    inductive_case
{-Types-}   bb :: (PS a->PS b),
            dd :: (PS a->PS a),
            cc :: (PS b->PS b),
            n  :: Nat,
            xx :: PS a,
            PS b
{-Steps-}
        dc2a bb dd cc (S n) xx
{-==-}  (cc . dmap (hypo (dc2a bb dd cc n)) (S N) . dd) xx
{-==-}  (cc . dmap (hypo (it2a bb dd cc n)) (S N) . dd) xx
{-==-}  (cc . dmap (down (dmap cc) n . dmap bb n .
                    up (dmap dd) n) (S N) . dd) xx
{-==-}  (cc . (dmap (down (dmap cc) n) (S N) .
              dmap (dmap bb n . up (dmap dd) n) (S N)) . dd) xx
{-==-}  (cc . (dmap (down (dmap cc) n) (S N) . dmap (dmap bb n) (S N) .
              dmap (up (dmap dd) n) (S N)) . dd) xx
{-==-}  (cc . (down (dmap cc . (+(S N))) n . dmap (dmap bb n) (S N) .
                    dmap (up (dmap dd) n) (S N)) . dd) xx
{-==-}  (cc . (down (dmap cc . (+(S N))) n . dmap bb (n+(S N)) .
                    dmap (up (dmap dd) n) (S N)) . dd) xx
{-==-}  (cc . (down (dmap cc . (+(S N))) n . dmap bb (n+(S N)) .
                    up (dmap dd . (+(S N))) n) . dd) xx
{-==-}  ((cc . down (dmap cc .(+(S N))) n) . dmap bb (n+(S N)) .
                   (up (dmap dd . (+(S N))) n . dd)) xx
{-==-}  ((dmap cc N . down (dmap cc .(+(S N))) n) . dmap bb (n+(S N)) .
                   (up (dmap dd . (+(S N))) n . dd)) xx
{-==-}  ((dmap cc N . down (dmap cc .(+(S N))) n) . dmap bb (n+(S N)) .
                   (up (dmap dd . (+(S N))) n . dmap dd N)) xx
{-==-}  (down (dmap cc) (n+(S N)) . dmap bb (n+(S N)) .
                   (up (dmap dd . (+(S N))) n . dmap dd N)) xx
{-==-}  (down (dmap cc) (n+(S N)) . dmap bb (n+(S N)) .
                   up (dmap dd) (n+(S N))) xx
{-==-}  (down (dmap cc) (S n) . dmap bb (n+(S N)) .
                   up (dmap dd) (n+(S N))) xx
{-==-}  (down (dmap cc) (S n) . dmap bb (S n) . up (dmap dd) (n+(S N))) xx
{-==-}  (down (dmap cc) (S n) . dmap bb (S n) . up (dmap dd) (S n)) xx
{-==-}  it2a bb dd cc (S n) xx
```

A.2.2 Regeldatei

```
{-Name-}    dc2a_then
{-Types-}   bb :: (PS a->PS b),
            dd :: (PS a->PS a),
            cc :: (PS b->PS b),
            (PS a -> PS b)
{-Steps-}
        dc2a bb dd cc N
{-==-}  bb
```

```
{-Name-}    dc2a_else
{-Types-}   bb :: (PS a->PS b),
            dd :: (PS a->PS a),
            cc :: (PS b->PS b),
            n  :: Nat,
            (PS a -> PS b)
{-Steps-}
        dc2a bb dd cc (S n)
{-==-}  cc . dmap (dc2a bb dd cc n) (S N) . dd

{-Name-}    it2a_definition
{-Types-}   bb :: (PS a->PS b),
            dd :: (PS a->PS a),
            cc :: (PS b->PS b),
            n  :: Nat,
            (PS a -> PS b)
{-Steps-}
        it2a bb dd cc n
{-==-}  down (dmap cc) n . dmap bb n . up (dmap dd) n

{-Name-}  compose
{-Types-} f :: (b->c),
          g :: (a->b),
          x :: a,
          c
{-Steps-}
(f . g) x {-==-} f (g x)

{-Name-}  down_N
{-Types-} f :: (Nat -> a -> a),
          (a -> a)
{-Steps-}
        down f N
{-==-}  id

{-Name-}  up_N
{-Types-} f :: (Nat -> a -> a),
          (a -> a)
{-Steps-}
        up f N
{-==-}  id

{-Name-}  dmap_N
{-Types-} f :: (PS a -> PS b),
          (PS a -> PS b)
{-Steps-}
        dmap f N
{-==-}  f
```

```
{-Name-}  dmap_distribution
{-Types-} f :: (PS b -> PS c),
          g :: (PS a -> PS b),
          n :: Nat,
          (PS a -> PS c)
{-Steps-}
        dmap (f . g) n
{-==-}  dmap f n . dmap g n

{-Name-}  law2
{-Types-} f :: (PS a->PS a),
          n :: Nat,
          m :: Nat,
          (PS a->PS a)
{-Steps-}
        dmap (down (dmap f) n) m
{-==-}  down (dmap f . (+m)) n

{-Name-}  law3
{-Types-} f :: (PS a->PS a),
          n :: Nat,
          m :: Nat,
          (PS a->PS a)
{-Steps-}
        dmap (up (dmap f) n) m
{-==-}  up (dmap f . (+m)) n

{-Name-}  law4
{-Types-} f :: (Nat -> a -> a),
          n :: Nat,
          (a -> a)
{-Steps-}
        f N . down (f . (+(S N))) n
{-==-}  down f (n+(S N))

{-Name-}  law5
{-Types-} f :: (Nat -> a -> a),
          n :: Nat,
          (a -> a)
{-Steps-}
        up (f . (+(S N))) n . f N
{-==-}  up f (n+(S N))


{-Name-}  inc
{-Types-} n :: Nat,
          Nat
{-Steps-}
S n {-==-} (n+(S N))
```

```
{-Name-}  law6
{-Types-} f :: (PS a->PS b),
          n :: Nat,
          m :: Nat,
          (PS a->PS b)
{-Steps-}
        dmap (dmap f n) m
{-==-}  dmap f (n+m)

{-Name-}  assoc5
{-Types-} a :: (e -> f),
          b :: (d -> e),
          c :: (c -> d),
          d :: (b -> c),
          e :: (a -> b),
          (a -> f)
{-Steps-}
        a . (b . c . d) . e
{-==-}  (a . b) . c . (d . e)
```

A.2.3 Deklarationsdatei

```
id :: a -> a
id x = x
```

A.2.4 Importdatei

```
data Nat  = N | S Nat deriving (Eq,Ord,Show)
data PS a = Sgt a | Com [PS a] deriving (Eq,Show)

instance Enum Nat where
 enumFrom = undefined

instance Num Nat where
 (+) = undefined
 (-) = undefined

com :: [PS a] -> PS a
com as = Com as

uncom :: PS a -> [PS a]
uncom (Com as) = as

comp :: [a->a] -> a -> a
comp   []   = id
comp (f:fs) = f . comp fs

up, down :: (Nat -> a -> a) -> Nat -> a -> a
down f n = comp [ f i | i<-[N .. n-1] ]
```

```
up f n   = comp [ f i | i<-[n-1,n-2..0] ]

dmap :: (PS a -> PS b) -> Nat -> (PS a -> PS b)
dmap f   N   = f
dmap f (S n) = com . map (dmap f n) . uncom

it2a :: (PS a -> PS b) -> (PS a -> PS a) -> (PS b -> PS b) ->
        Nat -> PS a -> PS b
it2a bb dd cc n = down (dmap cc) n . dmap bb n . up (dmap dd) n

dc2a :: (PS a -> PS b) -> (PS a -> PS a) -> (PS b -> PS b) ->
        Nat -> PS a -> PS b
dc2a bb dd cc   N   = bb
dc2a bb dd cc (S n) = cc . dmap (dc2a bb dd cc n) (S N) . dd
```

A.2.5 Ergebnis des Induktionsfalles

```
dc2a bb dd cc (S n) xx
{-
  Rule: dc2a_else
-}
(cc . dmap (dc2a bb dd cc n) (S N) . dd) xx
{-
  Cond: dc2a bb dd cc n `inductive` it2a bb dd cc n
-}
(cc . dmap (it2a bb dd cc n) (S N) . dd) xx
{-
  Rule: it2a_definition
-}
(cc . dmap (down (dmap cc) n . dmap bb n . up (dmap dd) n) (S N) . dd) xx
{-
  Rule: dmap_distribution
-}
(cc . (dmap (down (dmap cc) n) (S N) . dmap (dmap bb n .
       up (dmap dd) n) (S N)) . dd) xx
{-
  Rule: dmap_distribution
-}
(cc . (dmap (down (dmap cc) n) (S N) . dmap (dmap bb n) (S N) .
       dmap (up (dmap dd) n) (S N)) . dd) xx
{-
  Rule: law2
-}
(cc . (down (dmap cc . \l50_1 -> l50_1 + S N) n . dmap (dmap bb n) (S N) .
       dmap (up (dmap dd) n) (S N)) . dd) xx
{-
  Rule: law6
-}
(cc . (down (dmap cc . \l49_1 -> l49_1 + S N) n . dmap bb (n + S N) .
       dmap (up (dmap dd) n) (S N)) . dd) xx
```

```
{-
  Rule: law3
-}
(cc . (down (dmap cc . \147_1 -> 147_1 + S N) n . dmap bb (n + S N) .
       up (dmap dd . \148_1 -> 148_1 + S N) n) . dd) xx
{-
  Rule: assoc5
-}
((cc . down (dmap cc . \145_1 -> 145_1 + S N) n) . dmap bb (n + S N) .
       up (dmap dd . \146_1 -> 146_1 + S N) n . dd) xx
{-
  Rule: dmap_N
-}
((dmap cc N . down (dmap cc . \143_1 -> 143_1 + S N) n) . dmap bb (n + S N) .
       up (dmap dd . \144_1 -> 144_1 + S N) n . dd) xx
{-
  Rule: dmap_N
-}
((dmap cc N . down (dmap cc . \141_1 -> 141_1 + S N) n) . dmap bb (n + S N) .
       up (dmap dd . \142_1 -> 142_1 + S N) n . dmap dd N) xx
{-
  Rule: law4
-}
(down (dmap cc) (n + S N) . dmap bb (n + S N) .
       up (dmap dd . \140_1 -> 140_1 + S N) n . dmap dd N) xx
{-
  Rule: law5
-}
(down (dmap cc) (n + S N) . dmap bb (n + S N) . up (dmap dd) (n + S N)) xx
{-
  Rule: inc
-}
(down (dmap cc) (S n) . dmap bb (n + S N) . up (dmap dd) (n + S N)) xx
{-
  Rule: inc
-}
(down (dmap cc) (S n) . dmap bb (S n) . up (dmap dd) (n + S N)) xx
{-
  Rule: inc
-}
(down (dmap cc) (S n) . dmap bb (S n) . up (dmap dd) (S n)) xx
{-
  Rule: it2a_definition
-}
it2a bb dd cc (S n) xx
```

Anhang B

Grammatik

Der Aufbau der Grammatik orientiert sich an den Ausführungen des Haskell-1.3 Reports [30]. Die Namen der verwendeten Terminale und Nichtterminale korrespondieren zum größten Teil mit denen, die in der Beschreibung des Reports benutzt werden. Daher lassen sich beide Grammatiken einfach miteinander vergleichen.

Zur Vermeidung von Konflikten (*shift-reduce, reduce-reduce* [1]) mußten einige Anpassungen vorgenommen werden. Der Parser erzeugt mit der gegebenen Grammatik einen Parsebaum, der erst durch eine nachträgliche Bearbeitung in eine korrekte Repräsentation des eingegebenen Haskell-Programms umgeformt wird. Der Parser stellt also nur eine der Komponenten dar, die für die Abwicklung der Eingabe benötigt wird. Er nimmt aber eine zentrale Stellung ein.

```
> {
> module Parser where

> import Lexer
> import Exp
> import ParserUtility

> }

The function used for parsing (i.e. the parser) will get
the name 'parser'. The tokens produced by the scanner
have the form (Token, column, line).

> %name parser
> %tokentype { (Token,Int,Int) }

Definition of the terminal symbols used inside the grammar

> %token

indentifiers
------------

>       Varid   { (Varid  $$,_,_) }
>       Conid   { (Conid  $$,_,_) }
>       Qvarid  { (Qvarid $$,_,_) }
>       Qconid  { (Qconid $$,_,_) }
```

```
constructors/operators
----------------------

>       Varsym  { (Varsym  $$,_,_) }
>       Consym  { (Consym  $$,_,_) }
>       Qvarsym { (Qvarsym $$,_,_) }
>       Qconsym { (Qconsym $$,_,_) }

literal
-------

>       INT     { (TokenInt $$,_,_)    }
>       FLOAT   { (TokenFloat $$,_,_)  }
>       CHAR    { (TokenChar $$,_,_)   }
>       STRING  { (TokenString $$,_,_) }

special characters
------------------

>       '('     { (TokenOpenParen,_,_) }
>       ')'     { (TokenCloseParen,_,_) }
>       ','     { (TokenComma,_,_) }
>       ';'     { (TokenSemiColon,_,_) }
>       '['     { (TokenOpenSqBrack,_,_) }
>       ']'     { (TokenCloseSqBrack,_,_) }
>       '_'     { (TokenUnderBar,_,_) }
>       '`'     { (TokenBackQuote,_,_) }
>       '{'     { (TokenOpenCurBrack,_,_) }
>       '}'     { (TokenCloseCurBrack,_,_) }

>       "{"     { (TokenOpenCurBrackLayout _,_,_) }
>       "}"     { (TokenCloseCurBrackLayout,_,_)  }

reserved operators
------------------

>       ".."    { (TokenDotDot,_,_) }
>       "::"    { (TokenColonColon,_,_) }
>       '='     { (TokenEq,_,_) }
>       "\\"    { (TokenBackSlash,_,_) }
>       '|'     { (TokenGuard,_,_) }
>       "<-"    { (TokenLeftArrow,_,_) }
>       "->"    { (TokenRightArrow,_,_) }
>       '@'     { (TokenAs,_,_) }
>       '~'     { (TokenLazy,_,_) }
>       "=>"    { (TokenContext,_,_) }

reserved identifiers
--------------------

>       "case"          { (TokenCase,_,_) }
>       "class"         { (TokenClass,_,_) }
>       "data"          { (TokenData,_,_) }
```

```
>       "default"       { (TokenDefault,_,_) }
>       "deriving"      { (TokenDeriving,_,_) }
>       "do"            { (TokenDo,_,_) }
>       "else"          { (TokenElse,_,_) }
>       "if"            { (TokenIf,_,_) }
>       "import"        { (TokenImport,_,_) }
>       "in"            { (TokenIn,_,_) }
>       "infix"         { (TokenInfix,_,_) }
>       "infixl"        { (TokenInfixl,_,_) }
>       "infixr"        { (TokenInfixr,_,_) }
>       "instance"      { (TokenInstance,_,_) }
>       "let"           { (TokenLet,_,_) }
>       "module"        { (TokenModule,_,_) }
>       "newtype"       { (TokenNewType,_,_) }
>       "of"            { (TokenOf,_,_) }
>       "then"          { (TokenThen,_,_) }
>       "type"          { (TokenType,_,_) }
>       "where"         { (TokenWhere,_,_) }

> %%

either  we get a parse error
or      a list of declarations and
        a list which maps the name of an infix-operator
          to its precedence and associativity (left,no,right)

> main  :: { ErrorOr ([(String, AssocPrec)], [Decl]) }
> main  : impdecls ';' fixdecls ';' body       { oneErrorOr ((,) $3) $5 }
>       | impdecls ';' body                    { oneErrorOr ((,) []) $3 }
>       | fixdecls ';' body                    { oneErrorOr ((,) $1) $3 }
>       | body                                 { oneErrorOr ((,) []) $1 }

simple import (import Files are ignored)
-------------

> impdecls :: { Int }
> impdecls : impdecls ';' "import" Conid       { 0 }
>          | "import" Conid                    { 0 }

fixity
------

> fixdecls :: { [(String, AssocPrec)] }
> fixdecls : fixdecls ';' fix   { $3 ++ $1 }
>          | fix                { $1 }

> fix :: { [(String, AssocPrec)] }
> fix : "infix"  prec ops       { [(op, AssocPrec NonAssoc $2) | op <- $3] }
>     | "infixl" prec ops       { [(op, AssocPrec LeftAssoc $2) | op <- $3] }
>     | "infixr" prec ops       { [(op, AssocPrec RightAssoc $2) | op <- $3] }

> prec :: { Int }
> prec  : INT           { fst (head ((reads $1)::[(Int,String)])) }
>       |               { 9  }
```

```
> ops :: { [String] }
> ops : ops ',' op      { $3 : $1 }
>     | op              { [$1] }

> op :: { String }
> op : varop            { assocName $1 }
>    | conop            { assocName $1 }

toplevel declarations
---------------------

> body  :: { ErrorOr [Decl] }
> body  : topdecls                      { $1 }
>       |                               { Right [] }

> topdecls :: { ErrorOr [Decl] }
> topdecls : topdecls ';' topdecl       { p_list $1 $3 }
>          | topdecl                    { p_list (Right []) $1 }

> topdecl :: { ErrorOr Decl }
> topdecl : decl                        { $1 }

declarations
------------

the tokens "{" and "}" are inserted, if the layout mode is active

> decllist :: { ErrorOr [Decl] }
> decllist : '{' decls semicolon '}'    { $2 }

>          | "{" decls "}"              { $2 }
>          | "{" decls ';' "}"          { $2 }


> decls :: { ErrorOr [Decl] }
> decls : decls ';' decl        { p_list $1 $3 }
>       | decl                  { p_list (Right []) $1 }

> decl  :: { ErrorOr Decl }
> decl  : valdef                        { $1 }
>       | signdecl                      { $1 }


> signdecl :: { ErrorOr Decl }
> signdecl : vars "::" context_type     { Right (SignDecl $1 $3) }

> valdef :: { ErrorOr Decl }
> valdef : lhs '=' exp_pat where        { makeDecl $1 (true_guard $3) $4 }
>        | lhs gdrhs where              { makeDecl $1 $2 $3 }

> gdrhs :: { ErrorOr [Guard_Exp] }
> gdrhs : gdrhs '|' exp_pat_0 '=' exp_pat { guard_Exp $1 $3 $5 }
>       | '|' exp_pat_0 '=' exp_pat       { guard_Exp (Right []) $2 $4 }
```

```
> lhs   :: { ErrorOr LHS }
> lhs   : exp_pat_0
>         { either Left (Right . PatDecl) (snd $1) }

left-hand-side of function definitions

>       | var fexp_pat                  { patFunDecl $1 $2 Nothing }
>       | exp_pat_0 varop exp_pat_0     { patFunDecl $2 $1 (Just $3) }

expressions & patterns
----------------------

In order to avoid reduce-reduce conflicts the grammar rules
for expressions and patterns were combined. Otherwise the
qualifiers
            pat <- exp  (generator)
            exp         (filter)
would cause problems in the rule of the list comprehension.

> exp_pat :: { ExpPat }
> exp_pat
>       : exp_pat_0 "::" context_type   { expTyped $1 $3 }
>       | exp_pat_0                     { $1 }

> exp_pat_0 :: { ExpPat }
> exp_pat_0
>       : exp_pat_0 qop exp_pat_0       { expBin $2 $1 $3 }
>       | exp_pat_10                    { $1 }

> exp_pat_10 :: { ExpPat }
> exp_pat_10
>       : fexp_pat                      { isApplyOk $1 }

>       | "if" exp_pat "then" exp_pat "else" exp_pat    { expIf $2 $4 $6 }
>       | "\\" fexp_pat "->" exp_pat                    { expLambda $2 $4 }
>       | "case" exp_pat "of" altlist                   { expCase $2 $4 }
>       | "let" decllist "in" exp_pat                   { expLet  $2 $4 }

> fexp_pat :: { ExpPat }
> fexp_pat
>       : aexp_pat
>         { let a = Right (ExpList []) in exp_pat_list (a,a) $1 }
>       | fexp_pat aexp_pat
>         { exp_pat_list  $1 $2 }

> aexp_pat :: { ExpPat }
> aexp_pat
>       : qvar
>         { (Right $1, Left "\nPattern can not be a qualified name") }
>       | var                           { let ep=Right $1 in (ep,ep) }

>       | gcon                          { let ep=Right $1 in (ep,ep) }
>       | literal                       { let ep=Right $1 in (ep,ep) }
```

```
>       | '(' exp_pat_list ')'         { expTupel $2 }
>       | '[' exp_pat_list ']'         { $2 }

>       | '[' exp_pat '|' quallist ']'  { expListComprehension $2 $4 }

>       | '[' exp_pat_list ".." upbound ']'     { expArithSeq $2 $4 }

>       | '(' exp_pat_0 qop ')' { expSection 'R' $3 $2 }
>       | '(' qop exp_pat_0 ')' { expSection 'L' $2 $3 }

>       | '~' aexp_pat         { patLazy $2 }
>       | '_'                  { patWildcard }
>       | var '@' aexp_pat     { patAsPattern $1 $3 }

missing upper bound in an arith. sequence

> upbound :: { ExpPat }
> upbound
>       : exp_pat       { $1 }
>       |               { (Right ExpInfinite, Left "") }

nonterminals for the alternatives in a case-expression
the tokens "{" and "}" are inserted, if the layout mode
is active

> altlist :: { ErrorOr [CaseAlt] }
> altlist
>       : '{' alts semicolon '}'       { $2 }

>       | "{" alts     "}"             { $2 }
>       | "{" alts ';' "}"             { $2 }

> alts  :: { ErrorOr [CaseAlt] }
> alts  : alts ';' alt         { p_list $1 $3 }
>       | alt                  { p_list (Right []) $1 }

> alt   :: { ErrorOr CaseAlt }
> alt   : exp_pat "->" exp_pat where    { caseAlt $1 (true_guard $3) $4 }
>       | exp_pat gdpat where           { caseAlt $1 $2 $3 }

> gdpat :: { ErrorOr [Guard_Exp] }
> gdpat : gdpat '|' exp_pat_0 "->" exp_pat
>         { guard_Exp $1 $3 $5 }
>       | '|' exp_pat_0 "->" exp_pat
>         { guard_Exp (Right []) $2 $4 }

list of qualifiers used inside a list comprehension

> quallist :: { ErrorOr [ListCompQual] }
> quallist
>       : quallist ',' qual    { p_list $1 $3 }
>       | qual                 { p_list (Right []) $1 }
```

```
> qual  :: { ErrorOr ListCompQual }
> qual  : exp_pat "<-" exp_pat  { listCompGen $1 $3 }
>       | exp_pat               { either Left (Right . ListCompFilter) (fst $1)}
>       | "let" decllist        { either Left (Right . ListCompLet) $2 }

> exp_pat_list :: { ExpPat }
> exp_pat_list
>       : exp_pat_list ',' exp_pat
>         { exp_pat_list  $1 $3 }
>       | exp_pat
>         { let a = Right (ExpList []) in exp_pat_list (a,a) $1 }

where-part of a declaration or case-expression

> where :: { ErrorOr [Decl] }
> where :                       { Right [] }
>       | "where" decllist      { $2 }

> semicolon :: { Bool }
> semicolon
>       :       { False }
>       | ';'   { True  }

--------------------------------

> vars  :: { [Exp] }
> vars  : vars ',' var          { $3:$1 }
>       | var                   { [$1]  }

> var   :: { Exp }
> var   : Varid                 { ExpVar $1 ExpVarid  }
>       | '(' Varsym ')'        { ExpVar $2 ExpVarsym }

> qvar  :: { Exp }
> qvar  : Qvarid                { ExpVar $1 ExpQvarid  }
>       | '(' Qvarsym ')'       { ExpVar $2 ExpQvarsym }

> gcon  :: { Exp }
> gcon  : '(' ')'               { ExpCons ConsUnit }
>       | '[' ']'               { ExpCons ConsList }
>       | '(' commas ')'        { ExpCons (ConsTupel $2) }
>       | qcon                  { $1 }

> commas :: { Int }
> commas
>       : commas ','            { $1+1 }
>       | ','                   { 2 }

> qcon  :: { Exp }
> qcon  : Conid                 { ExpCons (Cons $1 ExpConid)   }
>       | Qconid                { ExpCons (Cons $1 ExpQconid)  }
>       | '(' Consym ')'        { ExpCons (Cons $2 ExpConsym)  }
>       | '(' Qconsym ')'       { ExpCons (Cons $2 ExpQconsym) }
```

```
> literal :: { Exp }
> literal
>       : INT           { ExpLiteral (LitInt    $1) }
>       | FLOAT         { ExpLiteral (LitFloat  $1) }
>       | CHAR          { ExpLiteral (LitChar   $1) }
>       | STRING        { ExpLiteral (LitString $1) }


context and types
-----------------

The context-part is parsed with the nonterminal 'type'.
Therefore the correct structure of

        context ::
        context : class1
                | '(' classlist ')'

        class1  ::
        class1  : Qconid Varid

        classlist ::
        classlist : classlist ',' class1
                  | class1

must be proved afterwards. Reduce-reduce conflicts are
responsible for this adaptation.

> context_type :: { Type }
> context_type  : type            { $1 }
>               | type "=>" type  { Context $1 $3 }

> type  :: { Type }
> type  : btype                 { $1 }
>       | btype "->" type       { TypeArrow $1 $3 }

> btype :: { Type }
> btype : atype                 { $1 }
>       | btype atype           { expType $1 $2 }

> atype :: { Type }
> atype : Qconid                { TypCons $1 }
>       | Conid                 { TypCons $1 }
>       | Varid                 { TypVar  $1 }
>       | '(' ')'               { UnitType   }

>       | '[' ']'               { ListCons }
>       | '[' type ']'          { ListType $2 }

>       | '(' typelist ')'      { let l = length $2
>                                 in    if l == 1
>                                       then head $2
>                                       else TupelType l (reverse $2)
>                               }
```

```
>       | '(' commas ')'        { TupelType 0 [] }

>       | '(' "->" ')'          { FuncCons }

> typelist :: { [Type] }
> typelist : typelist ',' type          { $3:$1 }
>          | type                       {  [$1] }
```

```
constructor / operator rules
-----------------------------

> qop   :: { Exp }
> qop   : qvarop                { $1 }
>       | qconop                { $1 }

> varop :: { Exp }
> varop : Varsym                { ExpVar $1 ExpVarsym }
>       | '`' Varid '`'         { ExpVar $2 ExpVarid  }

> qvarop :: { Exp }
> qvarop : Qvarsym              { ExpVar $1 ExpQvarsym }
>        | '`' Qvarid '`'       { ExpVar $2 ExpQvarid  }
>        | varop                { $1 }

> conop :: { Exp }
> conop : Consym                { ExpCons (Cons $1 ExpConsym) }
>       | '`' Conid '`'         { ExpCons (Cons $2 ExpConid)  }

> qconop :: { Exp }
> qconop : Qconsym              { ExpCons (Cons $1 ExpQconsym) }
>        | '`' Qconid '`'       { ExpCons (Cons $2 ExpQconid)  }
>        | conop                { $1 }
```

```
grammar end
-----------

> {

This function is called if a parse error (happyFail) occurs.

> happyError :: Int -> [(Token,Int,Int)]
>               -> ErrorOr ([(String, AssocPrec)], [Decl])
> happyError i tks
>  = case tks of
>       ((_,col,line):_) -> Left ("\n\nparse error near line "  ++
>                                          show line           ++
>                                  " / column " ++ show col     ++
>                                  "\n\nTokenStack:\n"          ++
>                                  showMyTokens (take 10 tks) "\n"
>                                 )
>       _                -> Left ("parse error near end of file\n")
```

```
To invoke the parser with the ability of treating the layout mode, use

> layParser:: String -> ErrorOr ([(String, AssocPrec)], [Decl])
> layParser = parser . layoutlexer

If the layout mode should/need not to be considered, use

> simpleParser:: String -> ErrorOr ([(String, AssocPrec)], [Decl])
> simpleParser cs = parser (lexer cs 1 1)

With 1 and 1 the column and line of the first input-character is fixed.

used for layout rule (off-side rule)
--------------------
The output of happy-0.9a can be modified so that the full layout
style is supported. At four locations -- correspondending to the
missing "}"-token in the token stream produced by the scanner --
the call of happyFail (the missing "}" causes a parse error) must
be replaced by

 insertTokenCloseCurBrackLayout <next parser state>
                                <integer used to represent "}">

> insertTokenCloseCurBrackLayout shift lit _ tk hst ln tks hstl stk
>  = case stk of {
>       (HappyTerminal (TokenSemiColon,_,_) :
>       _ :
>       HappyTerminal (TokenOpenCurBrackLayout layouts,_,_) :
>       _ ) -> happyShift shift lit (TokenCloseCurBrackLayout,0,0)
>                          hst ln (laySkip (tk:tks) layouts) hstl stk;

>       (_ :
>       HappyTerminal (TokenOpenCurBrackLayout layouts,_,_) :
>       _ ) -> happyShift shift lit (TokenCloseCurBrackLayout,0,0)
>                          hst ln (laySkip (tk:tks) layouts) hstl stk
>       }

> }
```

Anhang C

Auszüge aus dem Quellprogramm

Es werden die in Kapitel 5 beschriebenen Funktionen dargestellt.

C.1 expCompare

```
type ProofType
 = [DoTest] -> Log -> [DerefVar] -> Info -> UModes -> EnvId
            -> BFCT -> Declarations -> [RULE]
            -> Result

===============
| unify/terms |
===============

> expCompare :: ProofType

call 'utility' to do the work

> expCompare dotest@(Utility _ : _)
>  = utility dotest


skip the obsolete backtrack point

> expCompare (OnReplace {} : dotest)
>  = expCompare dotest


> expCompare (OnApply exp1 exp2 drefVs : dotest)
>  | isHypo1 || isHypo2
>  = if isHypo1 && isHypo2

if both expressions are tagged with 'hypo' then accept
the equivalence of exp1 and exp2
```

```
>    then useHypo x1 x2 drefVs dotest

otherwise normalize both expressions

>    else normalize 1 (OnSuccess x1 x2 drefVs : dotest) []
>  | True
>  = normalize 1 (OnSuccess exp1 exp2 drefVs : dotest) []

>  where
>    isHypo (ResExp e (ResApply _) (ResExp _ (ResVarStr "hypo" _) _:args))
>       = (True, case args of

if args consists of only one element then pass the environment e to arg

>                 [arg] -> resEnv arg e
>                 _:as  -> ResExp e (ResApply (FunArity (length as))) args)
>    isHypo exp
>       = (False, exp)

>    (isHypo1, x1) = isHypo exp1
>    (isHypo2, x2) = isHypo exp2


If the two expressions are simple enough then compare them directly.
Otherwise decompose the expressions (with applies) if both represent
a function application.

> expCompare doTest@(OnSuccess (ResExp _ sp1 args) (ResExp _ sp2 _) _ : dotest)
>  | sp1 == sp2
>    = if null args then expCompare dotest else applies doTest
>  | isResApply sp1 && isResApply sp2
>    = applies doTest

-----------
| VarFree |
-----------

> expCompare (OnSuccess exp1 exp2 drefVs : dotest)
>  | isVar1eqVar2 exp1 exp2

exp1 and exp2 represent the same free Variable

>    = expCompare dotest

exp1 and/or exp2 is a free variable. Bind the variable
to the corresponding expression if it does not occur
inside this expression (occur check succeeds).

>  | isFreeVar exp1
>    = notOccur exp1 exp2 (dref dotest) dotest
>  | isFreeVar exp2
>    = notOccur exp2 exp1 (dref dotest) dotest
```

```
>  where

>      isVar1eqVar2 (ResVar _ VarFree id1) (ResVar _ VarFree id2) = id1==id2
>      isVar1eqVar2 _  _                                          = False
```

determine the backtrack point

```
>      dref do_test
>       = case skipUntilOnReplace do_test of
>             d@(OnReplace { u_ID } : _) -> [(fst u_ID, d)]
>             _                          -> [( -1 , [] )]
```

```
----------
| ResVar | VarDefined
----------
```

If the two expressions represent the same toplevel function
then they are equal. If it is the same local function then the
free variables inside it must be compared to assure equivalence.

```
> expCompare (OnSuccess
>       (ResVar e1 VarDefined id1)
>       (ResVar e2 VarDefined id2)
>       drefVs : dotest)
>  | id1==id2
>  = if isTopLevel id1
>    then expCompare dotest
>    else lookatFreeVars dotest

>  where
>    lambda e i = ResVar e VarLambda i

>    lookatFreeVars dotest
>               log bkpts info mode iD bfc declsEnv
>     = maybe
>        (error "\ni#ResVar: Vardefined")
>        (\(DeclEntry _ _ _ f _)
>         -> expCompare ([OnApply (lambda e1 i) (lambda e2 i) drefVs | i<-f] ++
>                        dotest)
>                        log bkpts info mode iD bfc declsEnv)
>        (declsLookUp declsEnv id1)
```

```
-------------
| Succeeded |
-------------
```

there is no more problem to prove

```
> expCompare []
>  = \log bkpts info mode iD bfc _ _ ->
>      UnifySuccess [] log bkpts info mode iD bfc
```

```
one proof-step is successfully finished

> expCompare dotest@(Succeeded _:_)
>  = \log bkpts info mode iD bfc _ _ ->
>        UnifySuccess dotest log bkpts info mode iD bfc

----------
| Failed |
----------

both expressions differ therefore a new way in the verification
process is chosen by the backtracking component

> expCompare (succ@(OnSuccess exp1 exp2 drefVs) : dotest)
>  = \log bkpts info mode iD -> let
>      logFail = append log
>                 ("\nReplace at loc:   "  ++
>                  show (fst iD:[fst d|d<-drefVs]) ++ "\n\t\t  " ++
>                  show mode ++ "\n\t" ++
>                  showResExpR exp1 "\n\t<->\n\t" ++ showResExpR exp2 "\n")
>      in failed dotest logFail bkpts info mode iD

unexpected behaviour: internal error

> expCompare dotest
>  = error ("\ni#Unify#missing Case#0\n" ++ showDOTEST dotest)
```

C.2 replaceLocals

```
> replaceLocals :: Bool -> [DoTest] -> ProofType

> replaceLocals
>        stripOff  --  can one of the expression be a lambda-abstraction?
>        doTest@(succ@(OnSuccess u_exp1 u_exp2 u_drefVs):dotest)
>        insCond log bkpts info mode iD bfc@(bind,_,_,_) declsEnv rules
>  = let

normalize u_exp1 and u_exp2 by using the function 'getValues'

>        (x,(l,d,i)) = getValues stripOff bind declsEnv patternMatch
>                         (u_exp1,u_exp2) Nothing (mode, u_drefVs, iD)
>        nlog x1 x2 = append log ("\nreplaceLocals:\t  "++showExp1Exp2 x1 x2)
>     in

>      if testStruct mode && tStruct u_exp1 + tStruct u_exp2 > 2
>      then

tests, wether u_exp1 and u_exp2 are structural equivalent

>       struct u_exp1 u_exp2
>      else
>       maybe
>        (error "\ni#replaceLocals\n")
```

go on proving the equivalence of the normalized expressions (exp1,exp2)

```
>       (\(exp1,exp2) -> (\sc ->
>           if case l of { UModes _ U_Cond -> True; _ -> False }
>           then case exp2 of
>                ResExp _ (ResPatternFail c) _ -> CondFail  c   d
>                _ -> CondSucc exp2 d
>           else call mode (sc:(if setReplace l u_exp1
>                          then OnReplace (succ:insCond) sc l info iD bfc:dotest
>                          else dotest))
>                     (nlog exp1 exp2) bkpts info l (incEnv i) bfc declsEnv rules
>        ) (OnSuccess exp1 exp2 d)) (repl top1 top2 x)
>  where
```

for "case x of { T -> 1 } == case x of { T -> 1 }"
the following alternatve of 'setReplace' is needed

```
>   setReplace _ (ResExp _ ResCaseAlt _)
>    = False
```

decide if a backtrack point must be set or not (depending of the mode)

```
>   setReplace (UModes _ mode) _
>    = case mode of
>       U_UseRules      -> True
>       U_NoRules       -> True
>       U_PatternMatch _-> True
>       U_Struct        -> True
>       U_Eval          -> error "\ni#replaceLocals:setReplace"
>       _               -> False
```

is there a necessity to test, if two expressions are structural equivalent

```
>   testStruct (UModes _ U_UseRules) = True
>   testStruct (UModes _ U_NoRules)  = True
>   testStruct _                     = False
```

```
determine the 'head' of an application:  f a b  --> f
                                         x      --> x
```

```
>   top1 = topSymbol u_exp1
>   top2 = topSymbol u_exp2
```

if the two declarations are the same, then the bodies need not be
compared, at most the free variables inside it must be considered

```
>   repl (ResVar _ VarDefined iD1) (ResVar _ _ iD2) _ | iD1==iD2
>    = Just (u_exp1,u_exp2)
>   repl _ _ x = x
```

mark declaration with identifier iD as used

```
>   mark (ResVar _ _ iD) exp
>    = setEnv exp (expEnvInsert (getExpEnv exp) iD
>                        (ResExp expEnvEmpty (ResMarked MLoc iD) []))
>   mark _ _
>    = error "\ni#mark\n"
```

tests, if u_exp1 and u_exp2 are structural equivalent, i.e. they
differ only -- more or less -- in local names

```
>   struct u_exp1 u_exp2
>    = let (x,(l,d,i)) =
>                getValues False bind declsEnv patternMatch (u_exp1,u_exp2)
>                               Nothing (UModes 1 U_Eval, [], iD)
>      in
>       maybe
>        (error "\ni#struct\n")
>        (\(exp1',exp2') ->
>          let  exp1 = mark top1 exp1'
>               exp2 = mark top2 exp2'
>               rexp1= mark top1 u_exp1
>               rexp2= mark top2 u_exp2
>               succ = OnSuccess exp1 exp2 d
>               util = Utility (Util_Rules mode rules)
>               log' = append log
>                      ("\n=======\nStruct\t" ++ showExp1Exp2 u_exp1 u_exp2 ++
>                       "\n=======\n")
>          in
>          expCompare
>           (succ : OnReplace [] util mode info iD bfc :
>            util : OnReplace insCond succ mode info iD bfc :
>            dotest)

>           log' [] [] (UModes maxNum U_Struct) (incEnv i) bfc declsEnv
```

take rexp1=rexp2 as a temporary rule

```
>           (makeStructRules rexp1 rexp2)) x
```

C.3 replaceTopLevel

```
> replaceTopLevel :: [DoTest] -> ProofType

> replaceTopLevel
>       doTest@(succ@(OnSuccess u_exp1 u_exp2 u_drefVs) : dotest)
>       insCond log bkpts info
>       mode@(UModes depth m) iD bfc@(bind,_,_,_) declsEnv rules

>  = let
```

replace toplevel declarations by using the function 'getValues'

```
>       (x,(_,d,i)) = getValues False bind declsEnv patternMatch (u_exp1,u_exp2)
>                                Nothing
>                                (UModes 0 (U_ReplaceGlobals mode), u_drefVs, iD)
>       nlog x1 x2 = append log ("\nreplaceTopLevel:\t  "++showExp1Exp2 x1 x2)
>    in
>     maybe
```

if no toplevel declaration can be replace then evaluate the expressions

```
>       replaceAll
```

after replacing toplevel declarations, prove the equivalence of the
transformed expressions (exp1,exp2)

```
>       (\(exp1,exp2)
>         -> (\sc-> expCompare
>               (sc:OnReplace (succ:insCond) sc mode info iD bfc:dotest)
>               (nlog exp1 exp2) bkpts info mode (incEnv i) bfc declsEnv rules)
>            (OnSuccess exp1 exp2 d)
>       ) x
>  where

>    replaceAll = let
```

evaluate the expressions u_exp1 u_exp2, but do not use more than 'depth' steps

```
>       (x,(l,d,i)) = getValues False bind declsEnv patternMatch (u_exp1,u_exp2)
>                                Nothing (UModes depth U_Eval, u_drefVs, iD)
>       nlog x1 x2 = append log ("\nreplaceAll:\t  "++showExp1Exp2 x1 x2)
```

gather information to insert a condition, if the equivalence of u_exp1
and u_exp2 can not be shown

```
>       setCond sc = OnReplace (succ:insCond) sc l info iD bfc :
>                    Utility (Util_SwitchMode mode) : dotest
>       setDepth (UModes depth _) = (UModes depth m)
>     in
>       maybe
```

evaluation had no effect, therefore try to insert a condition

```
>       (failed (setCond succ) log bkpts info mode iD bfc declsEnv rules)
```

go on proving the equivalence of the result of the evaluation (exp1,exp2)

```
>       (\(exp1,exp2)
>         -> (\sc-> expCompare
>               (sc : Utility (Util_SwitchMode mode) : setCond sc)
>               (nlog exp1 exp2) bkpts info (setDepth l)
>               (incEnv i) bfc declsEnv rules)
>            (OnSuccess exp1 exp2 d)
>       ) x
```

C.4 searchRules

```
> searchRules :: [DoTest] -> Log -> [DerefVar] -> Declarations -> [RULE]
>              -> Result

> searchRules (rule@(OnReplace {u_Success=OnSuccess {u_exp1,u_exp2},
>                   u_mode=u_mode@(UModes depth _)}) :
>                   dotest) log bkpts declsEnv rules
```

Determine among the 'rules' those, for which there is a possibility to solve the conflict of u_exp1 and u_exp2. The test, whether these rules are suitable, is carried out by 'nextRule'.

```
>  = nextRule (rule { u_mode = UModes depth (U_OtherRules useRules) } :
>             Utility (Util_SwitchMode u_mode) :
>             dotest) log bkpts declsEnv rules
>  where
```

Only if u_exp1 or u_exp2 can match one side of a rule, look for rules. (2==3, it is not necessary to invest work in searching suitable rules)

```
>   useRules =  if isRulePossible u_exp1 || isRulePossible u_exp2
```

to decide, if a rule is suitable, compare the symbols at the head of an application: (exp) g 2 <-> (side) f x   only match if   g==f

```
>              then filterRules (topSymbol u_exp1) (topSymbol u_exp2)
>                               (bothSides rules)
>              else []
```

the rules are not directed: use a==b and b==a

```
>   bothSides (lr@(_,r):rs)
>    = lr : (RightSide, r) : bothSides rs
>   bothSides [] = []

>   filterRules x1 x2
>    = filter
>       (\(s,(_,_,lhs,rhs,_)) ->
>          let (y1,y2) = if s==LeftSide then (lhs,rhs) else (rhs,lhs) in
>          matchRule x1 (topSymbol y1) && matchRule x2 (topSymbol y2))
```

check, if there is a chance to match an expression with one side of a rule

```
>   matchRule :: ResExp -> ResExp -> Bool
>   matchRule _ (ResVar _ VarLambda _)  = True
>   matchRule (ResVar _ VarFree _)  _   = True
>   matchRule (ResVar _ VarDefined iD1)
>             (ResVar _ VarDefined iD2) = iD1==iD2
>   matchRule _ (ResVar _ VarDefined iD)= not (isTopLevel iD)
>   matchRule _ (ResExp _ ResLambda _)  = True
>   matchRule (ResExp _ a1 _)
>             (ResExp _ a2 _)           = a1==a2
>   matchRule _ _                       = False
```

```
unexpected behaviour: internal error

> searchRules d _ _ _ _
>  = error $ "\ni#searchRules" ++showDOTEST d
```

C.5 nextRule

```
> nextRule ::  [DoTest] -> Log -> [DerefVar] -> Declarations -> [RULE]
>               -> Result

> nextRule (rep@(OnReplace
>        { u_Success = OnSuccess {u_exp1, u_exp2, u_drefVs},
>          u_mode = UModes depth
>                   (U_OtherRules ((side,(name,vars,lhs,rhs,condS)):o_rules)),
>          u_info,
>          u_bfc = (d_bind, d_filter, d_Conds, d_types),
>          u_ID
>        }  ) : dotest) log bkpts declsEnv

>  | l_a == 0 || l_a /= r_a

exps:             u_exp1   u_exp2
sides of rule:    lhs      rhs

we must show:     u_exp1 == lhs
                  u_exp2 == rhs

>  = unify 0 nIdFree (doTest exp_lhs exp_rhs usedRule)

>  | l_a == r_a && l_a > 0
>  = let

>        apply f = ResExp expEnvEmpty (ResApply (FunArity l_a)) (f:args)

rule of form:           f x1 ... xn == g y1 ... ym
and we want to show
                        f a1 ... ax == g b1 ... by      with n-x == m-y > 0

then the missing arguments will be inserted to test the rule

                        (f a1 ... ax) m1 ... m(n-x) ==
                        (g b1 ... by) m1 ... m(n-x)

>        (nIdTmp, args)
>         = foldr (\_ (mId, args) ->
>                  (incID mId,ResExp expEnvEmpty (ResMarked MRule mId) []:args) )
>                 (incEnv nIdFree,[])
>                 [1..l_a]
>    in
>     unify 1 nIdTmp (doTest (apply exp_lhs) (apply exp_rhs) usedRule)
```

```
>  | l_a == r_a && l_a < 0
>  = let

exps:             f1 x1 ... xn          f2 u1 ... uk
sides of rule:    g1 y1 ... ym          g2 v1 ... vl
                           (m-n)==(l-k) < 0

we must show:     f1 x1 ... xm == g1 y1 ... ym  (take (n-m) arguments of f1)
                  f2 u1 ... ul == g2 v1 ... vl  (take (k-l) arguments of f2)
                        x(m+1) == u(l+1)
                               ...
                            xn == uk

>       takeArgs e 0 args = resEnv (head args) e
>       takeArgs e a args = ResExp e (ResApply (FunArity a)) (take (a+1) args)

>       (l_e, l_args) = let ResExp e _ args = exp_lhs in (e,args)
>       (r_e, r_args) = let ResExp e _ args = exp_rhs in (e,args)

>    in
>     unify 2 nIdFree
>       (doTest (takeArgs l_e rl_a l_args) (takeArgs r_e rr_a r_args)
>               (foldr
>                (\(l,r) -> (OnApply (resEnv l l_e) (resEnv r r_e) u_drefVs:) )
>                usedRule
>                (zip (drop (rl_a+1) l_args) (drop (rr_a+1) r_args))))

>   where

function to compute the arity of an expression

>    arity (ResExp _ (ResApply f1) _)   = applyArity f1
>    arity _                            = 0

compare "exp1" with lhs of the rule and "exp2" with rhs and go on with "dotest"

>    doTest exp1 exp2 dotest
>     = OnApply lhs' exp1 [] :
>       OnApply rhs' exp2 [] :
>       dotest

determine which side of the rule is used

>    (exp_lhs, exp_rhs)
>       = if side==LeftSide then (u_exp1, u_exp2) else (u_exp2, u_exp1)

create free variables for the variables used in the rule (given in "vars")

>    (nIdFree, env, nTypes) = createFreeVars vars (incEnv (incEnv u_ID))

set the environment with the new inserted free variables
for the left and right side of the rule and for the used conditions

>    lhs' = resEnv lhs env
```

```
>    rhs' = resEnv rhs env
>    conds = [resEnv exp env | exp <- condS]

mark the variables used in the conditions as "important", i.e. they
mustn't be removed during a filter step of the already bound variables

>    u_filter = filterVars d_filter
>       (case concat [let (fv,_,_)=freeVarsIn d_bind declsEnv exp
>                        in fv | exp<-conds] of
>               fvs -> if null fvs then [] else [fvs])
>       (0,0)

arity of left-/right-hand-side rule
 lhs     rhs
 ------------
 f x == g 2 x
 arity of lhs: 1
 arity of rhs: 2

>    rl_a = arity lhs
>    rr_a = arity rhs

difference of arities
 exp_lhs    exp_rhs
 ------------------
    f    ==   g 2
 l_a : 1    r_a : 1

>    l_a = rl_a - arity exp_lhs
>    r_a = rr_a - arity exp_rhs

>    usedRule = Utility (Util_UseRule 0 (fst u_ID) name conds) : nextRule
>    nextRule = rep { u_mode = UModes depth (U_OtherRules o_rules) } : dotest

>    bkpt = (fst u_ID, nextRule)

test, whether the rule is applicable

>    unify i nId dotest = expCompare
>       dotest (logMes i) (bkpt:bkpts)
>       u_info (UModes maxNum U_TestRule) (incEnv nId)
>       (d_bind, u_filter, d_Conds, nTypes++d_types)
>       declsEnv

>    logMes i = append log
>       ("\nTry Rule#" ++ show i ++ "  " ++name++ " " ++ show side ++ "\n\t" ++
>        showExp1Exp2 lhs rhs)

all rules supplied by 'searchRules' couldn't be used to show the equivalence
of both expressions.

> nextRule (top:Utility (Util_SwitchMode u_mode):dotest) log bkpts declsEnv
>  = case u_mode of
>       UModes _ U_Struct
```

```
we can not show that the two expressions are structural equivalent

>         -> failure (skipUntilOnReplace dotest) log1 bkpts declsEnv

try to replace toplevel declarations

>       _ -> replace 3 (top{ u_mode } : dotest) log2 bkpts declsEnv
>  where
>    log1 = append log ("\nNot struct. equiv.")
>    log2 = append log ("\nNo rule is applicable")


unexpected behaviour: internal error

> nextRule _ _ _ _
>  = error "\ni#nextRule"
```

C.6 failure

```
> failure :: [DoTest] -> Log -> [DerefVar] -> Declarations -> [RULE]
>            -> Result

No more backtrack point to use => the verification fails

> failure [] log []
>  = \_ _ -> UnifyFailed log


The structural equivalence of the two expressions cannot be shown. Therefore
restore the original set of rules with 'rules' and search for another solution.

> failure (OnReplace {u_Success=Utility (Util_Rules _ rules)}:dotest) log drefs
>  = \decls _ -> failure (skipUntilOnReplace dotest) log drefs decls rules


the necessary backtracking operations depend on the mode 'm_mode'

> failure dotest log bkpts
>  = case snd bkpt of
>     doTest@(OnReplace { u_Success, u_mode=UModes depth m_mode } : _)
>      -> case m_mode of
>       U_UseRules         -> searchRules doTest log nbkpts
>       U_Struct           -> searchRules doTest log nbkpts
>       U_OtherRules rules -> nextRule doTest log nbkpts

replace toplevel declarations

>       U_NoRules          -> replace 3 doTest log nbkpts
```

determine the reason for the failure of the pattern match

```
>       U_PatternMatch _
>        -> if depth==0
>           then \_ _ -> PatternFail PatEvalToComplex
>           else if isConsLit exp
>                then \_ _ -> PatternFail PatConsDiffer
>                else if canReplace exp
>                     then replace 2 doTest log bkpts
>                     else whyNotReplace exp
>        where
>               exp = u_exp2 u_Success

>               canReplace (ResVar _ _ _)                = True
>               canReplace (ResExp _ (ResApply _) (f:_)) = canReplace f
>               canReplace (ResExp _ ResCase _)          = True
>               canReplace _                             = False

>               whyNotReplace (ResExp _ (ResPatternFail cause) _)
>                = \_ _ -> PatternFail cause
>               whyNotReplace _
>                = \_ _ -> PatternFail PatSymbol
```

evaluation failed, try to insert a condition which will solve the problem

```
>       U_Eval
>        -> \declsEnv -> (maybe
>               (failure (goOnTest dotest) (append log "\nCondFail") nbkpts)
>               (condSuccess doTest message nbkpts)
>               (setCond declsEnv doTest)) declsEnv
>        where
>           message = "\n#RULE: >Insert Condition<\n" ++ log
```

unexpected behaviour: internal error

```
>       _-> error ("\ni#Missing mode: " ++ show u_mode)
```

try next backtrack point

```
>     []-> failure [] log nbkpts
>  where
>    goOnTest []        = []
>    goOnTest (_:x)     = skipUntilOnReplace x
```

'bkpts' is sorted by the age of the backtrack points, i.e. the youngest
is at the top of the list
determine the next backtrack point 'bkpt' and the remaining ones 'nbkpts'

```
>    bkpt : nbkpts
>     = case dotest of
>        OnReplace { u_ID } : _
>           -> insertDerefVar (fst u_ID, dotest) bkpts
>        [] -> bkpts
>        _  -> error $ "\ni#onReplaceFail#1\n" ++ log
```

C.7 getValues

```
> getValues ::  Bool ->
>               Bindings ->
>               Declarations ->
>               ProofType ->
>               (ResExp,ResExp) ->
>               Maybe (ResExp,ResExp) ->
>               STATUS (Maybe (ResExp,ResExp))

> getValues stripOff pbind declsEnv patternMatch exps rexps
>  = if stripOff
```

```
look for lambda-abstractions and eliminate them -- if possible -- by
inserting pseudo-arguments
```

```
>    then checkForLambdas exps  `thenSTATUS`
>         maybe (getVs exps) getVs
>    else
>         getValues exps rexps

>  where

>  getVs :: (ResExp,ResExp) -> STATUS (Maybe (ResExp,ResExp))
>  getVs exps = getValues exps (Just exps)
```

```
each expression is simplified on its own, therefore set the number
of available evaluation steps to the minimum of the numbers, which can
be used for the expressions
```

```
>  minimum _ (UModes _ (U_ReplaceGlobals l))    = l
>  minimum (UModes _ (U_ReplaceGlobals l)) _    = l
>  minimum (UModes d1 m) (UModes d2 _)          = UModes (min d1 d2) m

>  getValues
>       :: (ResExp,ResExp)
>          -> Maybe (ResExp,ResExp)
>          -> STATUS (Maybe (ResExp,ResExp))
```

```
Simplify both expressions in 'exps'.
If there is no change then return the result of the last
simplification-step given in 'retExps' (perhaps 'Nothing').
```

```
>  getValues exps retExps
>   = getValues_ exps                   `thenSTATUS`
>     maybe (returnSTATUS retExps)
>           (\x -> checkForLambdas x    `thenSTATUS`
>                  maybe (retJ x) (\x -> getValues x (Just x)))
```

```
determine, whether one of the expressions changes
If the simplification has no effect on both expressions then return 'Nothing'.

> getValues_ :: (ResExp,ResExp) -> STATUS (Maybe (ResExp,ResExp))
> getValues_ (exp1,exp2)
>  =   getLevel                `thenSTATUS` \l  ->
>      getValue_ exp1          `thenSTATUS` \x1 ->
>      setLevel l              `thenSTATUS` \l1 ->
>      getValue_ exp2          `thenSTATUS` \x2 ->
>      getLevel                `thenSTATUS` \l2 ->
>      setLevel (minimum l1 l2) `thenSTATUS` \_  ->
>      maybe   (maybe retN (retJ . (,) exp1) x2)
>              (\x -> maybe (retJ (x,exp2)) (retJ . (,) x) x2)
>              x1
>   where
>    getValue_ :: ResExp -> STATUS (Maybe ResExp)
>    getValue_ exp = getValue exp       `thenSTATUS` \x ->
>                    decGlobalLevel     `thenSTATUS` \_ ->
>                    maybe retN recGetValue x

>    recGetValue :: ResExp -> STATUS (Maybe ResExp)
>    recGetValue exp
>     = getValue exp    `thenSTATUS` \x ->
>       maybe (retJ exp) recGetValue x

Is the total number of evaluation-steps reached?

> depthViolation (UModes 0 _)  = True
> depthViolation _             = False

Is a permission necessary to replace the name of recursive declarations?

> depthPermission (UModes _ m) = case m of
>      U_Eval           -> True
>      U_PatternMatch _ -> True
>      _                -> False

Is it allowed to replace toplevel declarations?

> replaceTopLevel (UModes _ (U_ReplaceGlobals _))      = True
> replaceTopLevel _                                    = False

   ============
   | getValue |
   ============

Simplify one expression.

> getValue :: ResExp -> STATUS (Maybe ResExp)

> getValue exp
>  = getLevel  `thenSTATUS` \level ->

'level' shows the mode and the number of available simplification steps
```

```
>     (case exp of
>       ResVar _ VarFree f_id
>        -> maybe
>            (case level of
>               (UModes _ (U_PatternMatch _))
```

Pattern match fails because of a free variable. Keep this variable in mind
to try the match again when this variable is bound.

```
>                -> retJ (ResExp expEnvEmpty (ResPatternFail (PatFree f_id)) [])
```

a free variable cannot be simplified any more, return Nothing

```
>               _-> retN)
```

If the 'VarFree'-variable is already bound, take the corresponding value.

```
>            (\(loc, exp) -> appendDerefVars loc `thenSTATUS` \_ -> retJ exp)
>            (bindLookUp pbind f_id)
```

Get the value of a variable bound by a lambda abstraction.
This correspondens to the beta-reduction.

```
>       ResVar env VarLambda lId
>        -> maybe retN retJ (expEnvLookUp env lId)
```

Replace the name of a local/toplevel declaration if it is possible, i.e.
  - during evaluation the number of steps may not be exhausted
    or
  - (in normal mode)
    this name may not have been replaced in an earlier step, i.e.
         the name is not marked as used.

```
>       ResVar env VarDefined dId
>        -> let
>               f (DeclEntry _ _ exp free rec)
>                = if depthPermission level
>                  then if depthViolation level
>                       then retN
>                       else decLevel    `thenSTATUS` \_ ->
>                            retExp nEnv
>                  else if (not (isTopLevel dId) || replaceTopLevel level) &&
>                          (not rec || notMarked)
>                       then if rec
>                            then retExp marked
>                            else retExp nEnv
>                       else retN
>                where
>                 nEnv   = newExpEnv env free

>                 marked = expEnvInsert nEnv dId
>                              (ResExp expEnvEmpty (ResMarked MLoc dId) [])
```

```
>                 retEnv env
>                  = retJ (if null free && not rec then exp else setEnv exp env)

>                 notMarked = maybe True (const False) (expEnvLookUp env dId)

>           in  maybe retN f (declsLookUp declsEnv dId)
```

combine the representation of function applications
(f x y) (g z) u   -->   f x y (g z) u
(\x->f x) (h a) v -->   f (h a) v

```
>       exp@(ResExp e (ResApply a) (f:args))
>        -> case f of
>        ResExp fe (ResApply fa) fargs ->
>               let arity = FunArity (applyArity a + applyArity fa)
>                   nargs = if isExpEnvEmpty fe then fargs
>                                                else map (\x->resEnv x fe) fargs
>                   exp_  = ResExp e (ResApply arity) (nargs++args)
>               in
>                   getValue exp_      `thenSTATUS`
>                   maybe (retJ exp_) retJ

>        ResExp _ ResLambda _ ->
>               maybe   retN
>                       (\x -> getValue x `thenSTATUS` maybe (retJ x) retJ)
>                       (bindLambdaVars exp)

>        _ -> getValue (resEnv f e) `thenSTATUS`
>             maybe retN (retJ . ResExp e (ResApply a) . (:args))
```

Try to evaluate a case-expression. Therefore use the function given
with 'patternMatch'.

```
>       ResExp env ResCase (exp:alts) | not (depthViolation level)
>        -> let
```

How many steps can be used to evaluate the case-expression?

```
>               d = case level of
>                       (UModes d (U_PatternMatch _))   -> d
>                       _                               -> maxNumPat

>               match = result (map (\(ResExp _ ResCaseAlt [pat,gu,rhs]) ->
>                         patternMatch [OnApply pat (resEnv exp env) [],
>                                Utility (Util_PatternMatch (gu,rhs))]
>                               "" [] [] (UModes d (U_PatternMatch env))
>                               (1,1) (pbind,([],[]),[],[]) declsEnv [])
>                               alts)
```

```
Determine, if the pattern match was successful or not. If it was not
successfull then return the reason for the mismatch.

>               result (PatternFail PatConsDiffer:r)
>                = result r
>               result (PatternFail x:_)
>                = (False, retJ (ResExp expEnvEmpty (ResPatternFail x) []))
>               result (PatternSuccess id' rhs:_)
>                = (True, retJ rhs)
>               result []
>                = (False, retJ
>                   (ResExp expEnvEmpty (ResPatternFail PatNotExhaustive) []))
>               result y
>                = error "\ni#patternMatch:\n"

>            in
>               case level of
>                 UModes _ (U_PatternMatch _)   -> snd match
>                 UModes _ U_Cond               -> snd match

>                 _     -> if fst match
>                          then decLevel `thenSTATUS` \_ ->snd match
>                          else retN

there are no changes possible

>       _-> retN
>     )
```

Literaturverzeichnis

[1] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] John Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Commun. ACM 21*, pages 613–641, 1978.

[3] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[4] Richard Bird. *Introduction to functional programming using Haskell*. Prentice-Hall, 1998.

[5] Manfred Broy. *Informatik – Eine grundlegende Einführung (Teil I)*. Springer, 1992.

[6] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[7] William Clocksin and Christopher Mellish. *Programming in Prolog*. Springer, 1990.

[8] Marcelo P. Fiore. A Coinduction Principle for Recursive Data Types based on Bisimulation. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pages 110–119. IEEE Computer Society Press, 1993.

[9] Andrew Gordon. A tutorial on co-induction and functional programming. In *Glasgow functional programming workshop*, pages 78–95. Springer Workshops in Computing, 1994.

[10] Robert Griesemer. Detection of side-effects in function procedures. Technical Report 94-032, International Computer Science Institute, The University of California at Berkeley, September 1994.

[11] Cordelia Hall, Kevin Hammond, Simon L. Peyton-Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[12] Chris Hankin. *Lambda calculi – A guide for computer scientists*. Clarendon Press, 1994.

[13] Michael Hanus. Lazy unification with simplification. In *Proc. 5th European Symposium on Programming*, pages 272–286. Springer LNCS 788, 1994.

[14] Helmut Herold. *lex und yacc – Lexikalische und syntaktische Analyse*. Addison-Wesley, 1995.

[15] Christoph A. Herrmann and Christian Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, March 1997.

[16] Dieter Hofbauer and Ralf-Detlef Kutsche. *Grundlagen des maschinellen Beweisens*. Vieweg, 1991.

[17] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[18] R.J.M. Hughes and J. O'Donnell. Expressing and reasoning about non-deterministic functional programs. In *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 308–328, London, UK, 1990. Springer-Verlag. British Computer Society Workshops in Computing Series.

[19] Michael Hußmann, Peter Schefe, and Andreas Fittschen. *Das LISP-Buch*. McGraw-Hill, 1988.

[20] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*. Springer LNCS 201, 1985.

[21] Robin Jones, Clive Maynard, and Ian Stewart. *The art of Lisp programming*. Springer, 1990.

[22] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, 1988.

[23] Stanley B. Lippman. *C++ – Einführung und Leitfaden*. Addison-Wesley, 1995.

[24] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[25] R. Milner and M. Tofte. Coinduction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, September 1991.

[26] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[27] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990.

[28] John T. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.

[29] Lawrence C. Paulson. *ML for the working programmer*. Cambridge Univ. Press, 1991.

[30] John Peterson and Kevin Hammond (editors). Report on the Programming Language Haskell, A Non-Strict Purely Functional Language (Version 1.3). Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996.

[31] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[32] Simon L. Peyton-Jones. Compiling Haskell by Program Transformation: A report from the trenches. In *Programming Languages and Systems – ESOP'96*, pages 18–44. Springer LNCS 1058, 1996.

[33] Simon L. Peyton-Jones and David Lester. *Implementing Functional Languages – A Tutorial.* Prentice-Hall, 1992.

[34] G. Revesz. *Lambda Calculus, Combinators and Functional Programming.* Cambridge tracts in theoretical computer science. Cambridge Univ. Press, 1988.

[35] D.A. Spuler. Compiler detection of function call side effects. *Informatica*, 18(2), 1994.

[36] Simon Thompson. *Haskell – The Craft of Funktional Programming.* Addison-Wesley, 1996.

[37] Raymond Turner. *Constructive Foundations for Functional Languages.* McGraw-Hill, 1991.

[38] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[39] Philip Wadler. The essence of functional programming. In *Proc. 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14. ACM Press, 1992.

[40] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming.* Springer LNCS 925, 1995.

[41] Ake Wikström. *Functional programming using standard ML.* Prentice-Hall, 1987.