Eine graphische Programmierumgebung für deklarative Programmiersprachen

von
Johannes Koj
Matrikelnummer 206228

Diplomarbeit im Fach Informatik

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinisch-Westfälischen Technischen Hochschule Aachen
im Oktober 2000

angefertigt am
Lehr- und Forschungsgerbiet Informatik II
bei
Prof. Dr. M. Hanus

Ich danke Herrn Prof. Dr. M. Hanus für die Überlassung des Themas sowie für die Bereitschaft, die vorliegende Arbeit zu begutachten und Herrn Prof. Dr. K. Indermark dafür, daß er sich als Zweitgutachter zur Verfügung gestellt hat.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Aachen, den 4.Oktober 2000

Inhaltsverzeichnis

1. Einleitung

Die Formulierung eines Algorithmus in einer beliebigen Programmiersprache ist nicht trivial. Jede Programmiersprache muß somit auch daran gemessen werden, wie einfach ein Algorithmus mit ihr implementiert werden kann. Die Auswahl einer bestimmten Sprache für die Lösung eines bestimmten Problems hat Auswirkungen auf die Entwicklungszeit, die Wartbarkeit und die Fehleranfälligkeit.

Die Ausdrucksmöglichkeiten klassischer imperativer Sprachen orientieren sich sehr stark an der Arbeitsweise eines von Neumann-Rechners. Ein imperatives Programm besteht im Wesentlichen aus einer Folge von Speichertransformationen. Es gibt zwar Konstrukte zur Kontrollfluß und Datenabstraktion, diese können die entscheidenden Nachteile jedoch nicht beheben: Das Ergebnis der Programmausführung hängt in hohem Maße von der Auswertungsreihenfolge ab. Es gibt eine Vielzahl von Fehlerquellen wie beispielsweise Seiteneffekte oder fehlende Initialisierung. Diese Effekte kann der Programmierer nur sehr schwer übersehen, er kann sich somit schlechter auf das eigentliche Problem konzentrieren.

Mit deklarativen Programmiersprachen wird das Ziel verfolgt, diese Nachteile imperativer Sprachen zu beheben. Sie zeichnen sich dadurch aus, dass sie völlig unabhängig vom von Neumann-Rechner sind. Sie werden zwar auf solchen Rechnern implementiert, der Programmierer muß jedoch nichts über die Arbeitsweise eines solchen Rechners wissen. Die Konzepte deklarativer Sprachen orientieren sich an bereits vorhandenen und gut verstandenen Konzepten. Seiteneffekte und ähnliche unerwünschte Effekte werden vermieden. Weiterhin kann die operationale Semantik in einfacher und klarer Weise definiert werden, wodurch die Verifikation erleichtert wird. Der Programmierer kann sich auf das eigentliche Problem konzentrieren und muß sich nicht mit dem als wilkürlich empfundenen von Neumann-Konzept auseinandersetzen, welches sich alleine aus elektotechnischen Randbedingungen ergibt.

Es gibt zwei verschieden Klassen von deklarativen Sprachen: Funktionale und logische Sprachen. Ein funktionales Programm besteht aus einer Menge von Funktionsdeklarationen. Ein solches Programm wird ausgeführt, indem ein Ausdruck relativ zu diesen ausgewertet wird. Die Vorteile funktionaler Sprachen sind referentielle Transparenz, Funktionen höherer Ordnung, geschachtelte Ausdrücke und meistens ein polymorphes Typsystem. Ein Logikprogramm besteht aus einer Menge von Fakten und Regeln. Wenn ein logisches Programm ausgeführt wird, wird versucht aus den Fakten mit Hilfe der Regeln neue Fakten abzuleiten. Vorteile logischer Programmiersprachen sind freie Variablen und Nichtdeterminismus.

Die funktional-logische Programmiersprache Curry verfolgt das Ziel, die Konzepte funktionaler und logischer Programmiersprachen zu vereinen. Zusätzlich werden auch noch nebenläufige Konzepte integriert.

Trotz der großen Vorteile deklarativer Programmiersprachen sind diese nicht seht populär. Dies liegt unter anderem daran, dass es erst wenige Programmierwekzeuge für sie gibt. Bei der Erstellung grosser Systeme kann auf die Unterstützung durch geeignete Werkzeuge jedoch kaum verzichtet werden. Für imperative Sprachen gibt es eine Reihe gut durchdachter und nützlicher Werkzeuge wie beispielsweise Debugger oder Generatoren. Für deklarative Sprachen ist dies jedoch nicht der Fall, und teilweise ist es auch noch unklar, wie geeignete Werkzeuge aussehen sollen. Um diese Situation zu verbessern, soll in dieser Arbeit eine graphische Programmierumgebung erstellt werden, die die Erstellung von Curry-Programmen unterstützt. Aufgrund der Ähnlichkeit zwischen Curry und der funktionalen Sprache Haskell kann sie auch für Haskell genutzt werden.

Die Programmierumgebung soll auf zwei verschiedenen Konzepten basieren: Zum Einen soll mit einem einfachen Debugger die Fehlersuche im Allgemeinen unterstützt werden. Zum Anderen soll es eine Vielzahl von Werkzeugen für spezielle Aufgaben geben. Neue Werkzeuge sollen auf einfache Art und Weise in die Programmierumgebung integriert werden können. Diese Werkzeuge dienen auch hauptsächlich der Auffindung von Fehlern, können jedoch auch anderen Zwecken dienen. Somit wird ein erstes Programmierwerkzeug für Curry geschaffen welches Schritt für Schritt innerhalb gewisser Grenzen erweitert werden kann.

2. Die Programmiersprache Curry

Deklarative Programmiersprachen lassen sich in funktionale, logische und nebenläufige Sprachen unterteilen. Jedes dieser Konzepte bietet gewisse Vorteile, so dass es sich anbietet, diese zu vereinen. Dieses tut die funktional-logische Sprache Curry, die die wichtigsten Konzepte der deklarativen Programmierung in einer eleganten Art und Weise integriert: Sie bietet die wichtigen funktionalen (Funktionen höherer Ordnung, partielle Applikation, geschachtelte Ausdrücke, polymorphes Typsystem) logischen (logische Variablen, partielle Datenstrukturen, eingebaute Suche) und nebenläufigen (Konjunktion von Constraints, Synchronistaion über logische Variablen) Konzepte an und ermöglicht die Kombination dieser. Dadurch werden zum Teil völlig neue Programmierstile ermöglicht.

Weiterhin werden die beiden wichtigen Auswertungsstrategien funktional-logischer Sprachen (Residuation und Narrowing) unterstützt. Curry könnte somit ein Standard für funktional-logische Sprachen werden.

Die Syntax von Curry orientiert sich stark an der der funktionalen Sprache Haskell, eine Implementierung von Curry ist mit gewissen Einschränkungen auch eine von Haskell.

2.1. Grundlagen

In diesem Abschnitt sollen einige grundlegende Begriffe erklärt werden. $\mathcal{V}$ bezeichne die Menge der Variablen (`x`,`y`, ...), $\mathcal{F}$ die Menge der Funktionssymbole (`f`,`g`, ...), $\mathcal{C}$ die Menge der Datenkonstruktorsymbole (`C`,`D`,...), $\mathcal{T}$ die Menge der Typvariablen (`a`,`b`, ...) und $\mathcal{TC}$ die Menge der Typkonstruktoren (`T`,`U`, ...).

Ein Ausdruck ist entweder eine Variable $\mathtt{x} \in \mathcal{X}$ oder eine Applikation $(\mathtt{f}\, e_1 \ldots e_n)$. Dabei ist `f` ein Funktionssymbol oder ein Datenkonstruktorsymbol $(\mathtt{f} \in \mathcal{F} \cup \mathcal{C})$ und $e_1 \ldots e_n$ beliebige andere Ausdrücke. Ein Datenterm ist ein Ausdruck, in dem keine Funktionssymbole vorkommen. Jedem Funktions- und Konstruktorsymbol `f` ist eine Stelligkeit $a_{\mathtt{f}}$ zugeordnet, dabei gilt $a_{\mathtt{f}} >= n$. Falls $a_{\mathtt{f}} > n$ gilt, spricht man auch von partieller Applikation. Die Menge der Ausdrücke wird auch als $\mathcal{E}(\mathcal{C} \cup \mathcal{F}, \mathcal{V})$ bezeichnet, die Menge der Datenterme mit $\mathcal{E}(\mathcal{C}, \mathcal{V})$. $root(e)$ bezeichnet die Wurzel eines Ausdrucks.

Ein Typausdruck ist entweder eine Typvariable $\mathtt{t} \in \mathcal{T}$ oder eine Applikation eines Typkonstruktors $\mathtt{T} \in \mathcal{TC}$ auf n andere Typausdrücke: $(\mathtt{T}\, t_1 \ldots t_n)$. Einem Typkonstruktor `T` wird ebenfalls eine Stelligkeit $a_{\mathtt{T}}$ zugeordnet, es gilt $a_{\mathtt{T}} = n$. Die Menge aller Typausdrücke wird mit $\mathcal{TE}(\mathcal{TC}, \mathcal{T})$ bezeichnet. Der rechtsassoziative Typkonstruktor `->` erzeugt den Funktionenraum.

Eine Substitution σ ist eine Abbildung $\sigma : \mathcal{V} \to \mathcal{E}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ $(\sigma : \mathcal{T} \to \mathcal{TE}(\mathcal{TC}, \mathcal{T}))$ von der Menge der (Typ-)Variablen in die Menge der (Typ-)Ausdrücke, bei der nur für endlich viele x $\sigma(x) \neq x$ gilt. Somit kann jede Substitution durch Angabe der Bilder dieser Variablen definiert werden: $\sigma = \{x_1 \to e_1, \ldots, x_n \to e_n\}$. Meistens wird eine Substitution mit ihrer homomorphen Fortsetzung auf (Typ-)Ausdrücke identifiziert.

2.2. Syntax

Im Folgenden werden kurz die wichtigsten syntaktischen Konstrukte der Programmiersprache Curry und ihre intuitive Bedeutung vorgestellt. Ein Curry-Programm besteht im Wesentlichen aus einer Menge von Datentyp- und Funktionsdeklarationen.

2.2.1. Datentypen

Mit Datentypen werden in Curry Wertebereiche festgelegt, auf denen Berechnungen durchgeführt werden sollen. Curry hat ein polymorphes Hindley/Milner-Typsystem.

Datentypdeklarationen beginnen mit dem Schlüsselwort `data`:

`data T a1 ...  an = C1` τ_{11} `...` τ_{1n_1} `| ... | Ck` $\tau_{k1} \ldots \tau_{k_n k}$

Mit dieser Deklaration wird ein neuer n-stelliger Typkonstruktor `T` mit Typparametern `a1`, …, `an` eingeführt. Auf der rechten Seite der Deklaration werden die Datenkonstruktoren `C1`, …, `Ck` definiert. Jedes `Ci` hat den Typ

τ_{i1} `-> ... ->` $\tau_{i n_i}$ `-> (T a1 ... an)`

(i=1…k). Dabei besteht jedes τ_{ij} aus beliebigen Typkonstruktoren und den Typvariablen `a1`, …, `an`. Durch Applikation des Typkonstruktors `T` auf n Typen $\tau_1, \ldots, \tau_n$ entsteht ein neuer Typ: (`T` $\tau_1 \ldots \tau_n$). Ein Element dessen Wertebereichs ergibt sich aus der Applikation eines Datenkonstruktors `Ci` auf beliebige Ausdrücke $e_1, \ldots, e_{ni}$ mit den Typen $\sigma(\tau_{i1}), \ldots, \sigma(\tau_{in_i})$ (dabei gilt $\sigma($`ai`$) = \tau_i$, i=1…n).

Bsp. 2.2.1 *Ein Datentyp für Binärbäume kann wie folgt deklariert werden:*

```
data Tree a = Node a (Tree a) (Tree a) | Leaf a
```

`(Tree Int)` *ist dann der Typ aller Binärbäume mit ganzzahligen Beschriftungen,* `(Node 1 (Leaf 2) (Leaf 3))` *ist vom Typ* `(Tree Int)` *und stellt einen Baumknoten mit zwei Blättern dar.* `Tree` *ist ein einstelliger Typkonstruktor mit Typparameter* `a`*,* `Node` *und* `Leaf` *sind Datenkonstruktoren.*

2.2.2. Funktionen

Funktionsdeklarationen sind das zentrale Element eines Curry-Programms. Eine Funktionsdeklaration besteht aus einer *Typdeklaration* und einer Menge von *Definitionsgleichungen*. Eine Funktionsdeklaration für eine n-stellige Funktion `f` hat folgende Form:

`f ::` τ_1 `-> ... ->` τ_n `->` τ

Dabei sind $\tau_1, \ldots, \tau_n, \tau$ Typausdrücke. $\tau_1, \ldots, \tau_n$ sind die Typen der Argumente von `f`, τ ist der Ergebnistyp. Obwohl Curry streng getypt ist, kann die Typdeklaration weggelassen werden, da das Typsystem die automatische Herleitung der Typen erlaubt.

Eine Gleichung für `f` wird wie folgt notiert:

`f` t_1 `...` t_n `|` c `=` e

Dabei sind $t_1, \ldots, t_n$ Datenterme, die als *formale Parameter* bezeichnet werden. e ist ein Ausdruck und wird als *rechte Seite* bezeichnet, entsprechend wird `f` $t_1 \ldots t_n$ auch als linke Seite bezeichnet. Innerhalb der linken Seite darf eine Variable maximal einmal auftreten. Die optionale *Bedingung* c ist ein Ausdruck vom Typ `Success` (siehe Abschnitt 2.3.4) oder `Bool`. Mehrer Gleichungen mit Bedingungen $c_1, \ldots, c_n$ und gleicher linker Seite können zusammengefasst werden:

$$
\begin{array}{ll}
\texttt{f}\ t_1\ \ldots\ t_n & |\ c_1\ \texttt{=}\ e_1 \\
 & \vdots \\
 & |\ c_n\ \texttt{=}\ e_n
\end{array}
$$

Wenn eine Gleichung angewendet wird, wird ein Funktionsaufruf (`f` $e_1 \ldots e_n$) durch die rechte Seite ersetzt. Die Bedingung einer Gleichung bestimmt, ob diese angewendet werden kann oder nicht.

Bsp. 2.2.2 *Die Fakultäts-Funktion kann wie folgt definiert werden:*

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

2.2.3. Lokale Deklarationen

Mit *lokalen Deklarationen* können Funktionen und Patterns definiert werden, die nur lokal sichtbar sind. Patterns sind Datenterme, in denen jede Variable maximal einmal vorkommt. Die in einem Pattern p vorkommendn Variablen werden durch Angabe einer Gleichung p=e gebunden.

Es gibt zwei Möglichkeiten, lokale Regeln und Patterns zu deklarieren: Mit *where*-Deklarationen können Bezeichner definiert werden, die in der gesamten rechten Seite der Regel sichtbar sind:

l `=` r
 `where` $l2$=$r2$
        p=e

Die Regel $l2$=$r2$ und alle Variablen, die im Datenterm p vorkommen, sind somit in der rechten Seite r und allen anderen lokalen Deklarationen sichtbar. Die Variablen in p werden auf die gleiche Art und Weise wie beim pattern matching (siehe Abschnitt 2.3.1) gebunden.

Mit let-Ausdrücken kann der Gültigkeitsbereich lokaler Deklarationen weiter eingeschränkt werden:

`let` l `=` r `in` e

Hier sind die entsprechenden Bezeichner nur im Ausdruck e sichtbar.

2.3. Operationale Semantik

2.3.1. Termersetzung

Die Ausführung eines Curry-Programms bedeutet das Auswerten eines Ausdrucks relativ zu einer Menge von Datentyp- und Funktionsdeklarationen. Die Auswertung eines Ausdrucks geschieht durch eine Folge von Reduktionsschritten: Ein Ausdruck e kann zu einem Ausdruck e' reduziert werden, falls die folgenden Bedingungen erfüllt sind:

1. Der Funktionsaufruf $(\texttt{f}\ e_1 \ldots e_n)$ ist ein Teilausdruck von e.
2. $l \mid c = r$ ist eine Regel für die Funktion `f` und es gibt eine Substitution σ mit $\sigma(l) = (\texttt{f}\ e_1 \ldots e_n)$.
3. Die Bedingung c ist erfüllt, das heißt sie lässt sich zu `True` bzw. `success` reduzieren.
4. e' entsteht durch Ersetzung von $(\texttt{f}\ e_1 \ldots e_n)$ in e durch $\sigma(r)$.

Das Finden einer passenden Regel und der Substitution σ und das Binden der Variablen durch Applikation von σ auf r wird auch *pattern matching* genannt. Der Ausdruck $(\texttt{f}\ e_1 \ldots e_n)$ wird dann auch als *Redex* bezeichnet. Im Allgemeinen kann ein Ausdruck mehrere Redexe haben, es muß also noch festgelegt werden, welcher Redex jeweils reduziert wird. In Haskell wird immer der linke äußerste Redex reduziert, in Curry wird zwar auch immer ein äußerster genommen, dies muß jedoch nicht unbedingt der linkeste sein. Ein Ausdruck wird so lange wie möglich reduziert. Ist eine Reduktion nicht mehr möglich, ergibt sich entweder als Ergebnis ein Datenterm, oder die Berechnung schlägt fehl.

Die Auswahl eines äußersten Redex ist ein wesentliches Merkmal der operationalen Semantik von Curry: Somit werden in bestimmten Fällen unendliche Auswertungen oder unnötige Auswertungen von Teilausdrücken vermieden.

Bsp. 2.3.1 *Gegeben sei folgendes Curry-Programm:*

```
f x = f x
g x = 1
```

Der Ausdruck `g (f x)` *wird in einem Schritt zu* `1` *ausgewertet, da immer der äußerste Redex ersetzt wird. Wäre dies nicht der Fall, würde die Auswertung nicht terminieren.*

Diese Art der Auswertung wird auch "bedarfsgesteuerte Auswertung" genannt, da nur die Ausdrücke ausgewertet werden, deren Werte für die Reduktion benötigt werden.

2.3.2. Nichtdeterminismus

Im Allgemeinen ist es möglich, dass es zu einem Redex mehrere passende Regeln gibt, die also die zweite Bedingung aus dem vorherigen Abschnitt erfüllen. Es stellt sich die Frage, wie ein Ausdruck in diesem Falle ausgewertet wird.

In Haskell wird einfach die erste passende Regel angewendet und die anderen werden ignoriert. Dies hat den Nachteil, das die Semantik eines Programms von der Reihenfolge abhängig ist, in der die einzelnen Regeln definiert werden. In Curry wird dieser Nachteil beseitigt, indem nichtdeterminstisch eine passende Regel ausgewählt wird. Somit kann es zu einem Ausdruck mehrere korrekte Ergebnisse geben, die eine Implementierung von Curry mittels Backtracking findet.

Wegen des Nichtdeterminismus reicht es nun nicht mehr aus, in einem Reduktionsschritt nur den Redex durch die entsprechende rechte Seite zu ersetzen. Vielmehr müssen nun alle Ausdrücke, die durch Bindung der selben Variable entstanden sind, ersetzt werden. Diese Ausdrücke stehen alle für das Gleiche und müssen somit auch gleichzeitig ausgewertet werden. Dieses Prinzip wird auch als "Sharing" bezeichnet. Während es bei Implementierungen von Haskell lediglich zur Steigerung der Effizienz benutzt wird, kann eine Implementierung von Curry also hierdrauf nicht verzichten. Ohne Sharing könnte es sonst passieren, dass zwei Ausdrücke, die durch Instantiierung der selben Variable entstanden sind, zu verschiedenen Werten ausgewertet werden.

Diese Art von Nichtdeterminismus wird auch als "don't know"-Nichtdeterminismus bezeichnet. Eine andere Form dieser Art von Nichtdeterminismus wird im nächsten Abschnitt besprochen.

2.3.3. Residuation und Narrowing

Da Curry auch Konzepte der Logikprogrammierung unterstützt, können Ausdrücke freie (oder auch ungebundene) Variablen enthalten. Diese müßen jedoch innerhalb von lokalen Deklarationen deklariert werden: Mit `x,y free` können beispielsweise die beiden freien Variablen `x` und `y` deklariert werden. Freie Variablen können auf zwei verschiedene Weisen gebunden werden: Durch Lösen von Constraints (siehe Abschnitt 2.3.4) und durch Narrowing (siehe unten).

Durch das mögliche Auftreten freier Variablen ergibt sich ein neues Problem: Wie wird ein Funktionsaufruf ausgewertet, der freie Variablen enthält? Hierfür gibt es nun zwei Lösungsmöglichkeiten:

1. *Residuation*: Der Funktionsaufruf wird zunächst nicht weiter ausgewertet. Die Auswertung suspendiert, bis alle freien Variablen durch andere Funktionsaufrufe oder das Lösen von Constraints gebunden sind. Residuation wurde zum Beispiel in den Sprachen Goffin und Oz verwendet.

2. *Narrowing*: Es werden (nichtdeterministisch) alle anwendbaren Regeln angewendet. Dabei werden die freien Variablen an die entsprechenden Patterns gebunden. Dies ist die zweite mögliche Ursache für "don't know"-Nichtdeterminismus. Narrowing ist z.B. für die Sprache Babel implementiert worden.

Curry unterstützt sowohl Residuation als auch Narrowing: Mit den Auswertungsannotationen `eval rigid` bzw. `eval flex` kann für jede Funktion bestimmt werden, ob sie mittels Residuation oder Narrowing ausgewertet werden soll. Ohne Annotation wird eine Funktion (mit gewissen Ausnahmen) mittels Residuation ausgewertet.

Bsp. 2.3.2 *Die Funktion* `f` *wird mittels Narrowing ausgewertet:*

```
f eval flex
f 1 = True
f 2 = True

test | f x = x where x free
```

Die Funktion `test` *kann zu* `1` *oder* `2` *ausgewertet werden, dabei wird die Variable* `x` *einmal an* `1` *und einmal an* `2` *gebunden. Ohne Auswertungsannotation würde die Auswertung von* `test` *suspendieren.*

2.3.4. Constraints

Constraints werden auf Erfüllbarkeit geprüft. Sie können als Bedingungen von Regeln auftauchen und deren Anwendbarkeit bestimmen. Weiterhin werden sie für die nebenläufigen Programmierung benutzt.

Constraints haben in Curry den Typ `Success`. Es gibt zwei elementare Constraints: `success` ist der immer erfüllbare Constraint. Daneben gibt es in Curry noch die lösende Gleichheit zwischen zwei Ausdrücken: e_1 `=:=` e_2. Dieser Constraint wird gelöst, in dem zunächst e_1 und e_2 so weit wie möglich ausgewertet werden. Nur falls hierbei keine unendlichen Berechnungen auftreten, ist der Constraint erfüllbar (strikte Gleichheit). In diesem Falle werden die Ergebnisse e_1' und e_2' unifiziert, das heißt es wird eine Substitution σ mit $\sigma(e_1') = \sigma(e_2')$ gesucht. Falls diese existiert, ist der Constraint erfüllbar, wird zu `success` reduziert und die freien Variablen aus e_1' und e_2' werden entsprechend σ gebunden, ansonsten ist er unerfüllbar.

Die lösende Gleichheit `=:=` unterscheidet sich von der testenden Gleichheit `==`. Die boolesche Funktion `==` vergleicht zwei Ausdrücke und liefert entweder `True` oder `False` zurück. Falls in einem Ausdruck freie Variablen auftreten, suspendiert `==`. Ein Constraint wird lediglich auf Erfüllbarkeit überprüft. Es wird kein Wert zurück geliefert. Dafür werden eventuell auftretende freie Variablen gebunden.

Elementare Constraints können mit den Operatoren `&` und `&>` zu neuen Constraints kombiniert werden: c_1`&` c_2 ist erfüllbar, falls c_1 und c_2 erfüllbar sind. Eine solche Konjunktion von Constraint wird nebenläufig gelöst: Falls das Lösen von c_1 wegen freien Variablen suspendiert, kann trotzdem c_2 weiter aufgelöst werden. Hierdurch können die entsprechenden Variablen in c_1 gebunden werden, so dass mit c_1 fortgefahren werden kann usw. Mit c_1`&>` c_2 werden zwei Constraints sequentiell kombiniert, das heißt es wird erst c_1 gelöst und dann im Falle der Erfüllbarkeit c_2.

Obwohl weitere Constraints zunächst nicht vorgesehen sind, können Implementierungen von Curry beliebige weitere Constraints anbieten. Dies können zum Beispiel arithmetische Constraints, endliche Bereiche oder Constraints zur nebenläufigen und verteilten Programmierung sein.

2.4. Nebenläufige Programmierung

Obwohl nebenläufige Programmierung eine Reihe neuer Probleme wie Synchronisation und Kommunikation mit sich bringt, erlaubt sie häufig einfachere und elegantere Lösungen. Viele Probleme haben nebenläufigen Charakter, somit ist es schwierig für diese eine sequentielle Lösung zu finden.

Nebenläufigkeit wird in Curry durch die Konjunktion von Constraints ausgedrückt. Der Ausdruck

```
let x,y free in x=:=f 1 & y=:=g 1
```

besteht aus zwei Threads: Der eine berechnet das Ergebnis der Applikation `(f 1)`, der andere das von `(g 1)`. Da die Konjunktion zweier Constraints nebenläufig ausgewertet wird, handelt es sich wirklich um zwei unabhängige Berechnungen. Zwei Threads können über freie Variablen miteinander kommunizieren:

```
let x,y free in x=:=f y & y=:=g 1
```

Der erste Thread kann zunächst nicht weiter ausgewertet werden, da `y` noch ungebunden ist. Sobald jedoch der Wert von `(g 1)` berechnet ist, wird `y` an diesen gebunden. Somit hat der zweite Thread eine Nachricht, nämlich das Ergebnis von `(g 1)`, an den ersten geschickt. Dieser kann nun weiter ausgewertet werden. Freie Variablen (und Residuation) dienen somit gleichzeitig der Kommunikation und der Synchronisation.

2.4.1. Committed Choice

Commited choice ist die zweite Art von Nichtdeterminismus und wird hauptsächlich zur Realisierung von n-zu-1 Kommunikation benutzt. Eine Aufruf einer Funktion mit der Auswertungsannotation `eval choice` wird wie folgt ausgewertet: Sobald eine Regel passt und ihre Bedingung erfüllt ist, wird sie angewendet und alle anderen Regeln werden ignoriert. Dabei dürfen keine freien Variablen gebunden werden.

Für die 1-zu-1 Kommunikation bietet es sich an, Ströme als Kommunikationsmedium zu wählen. Ströme sind Listen, die Schritt für Schritt instantiiert werden: Der Sender bindet das nächste Element der Liste um eine Nachricht zu verschicken, der Empfänger liest nach und nach alle Nachrichten aus dieser Liste. 2-zu-1 Kommunikation kann nun mit der nichtdeterministischen Funktion `merge` realisiert werden:

```
merge eval choice
merge [] ms2 = ms2
merge ms1 [] = ms1
merge (m:ms1) ms2 = m : (merge ms1 ms2)
merge ms1 (m:ms2) = m : (merge ms1 ms2)
```

Die Eingangsströme der beiden Sender werden durch `merge` zu einem Ausgangsstrom für den Empfänger gemischt. n-zu-1 Kommunikation kann durch mehrfache Anwendung von `merge` realisiert werden. Da die Reihenfolge der Nachrichten nicht vorhergesehen werden kann, ist die Verwendung einer nichtdeterministischen Funktion essentiell.

Diese Form von Nichtdeterminismus wird auch “don’t care”-Nichtdeterminismus genannt.

2.4.2. Ports

Die Verwendung von `merge` zur strombasierten n-zu-1 Kommunikation hat einen entscheidenden Nachteil: Die Anzahl der Operationen, die zum Versenden einer Nachricht benötigt werden, ist proportional zu n. Ports können diesen Nachteil beheben.

Obwohl eine Implementierung von Curry keine Ports anbieten muß, stellt die PAKCS-Distribution (siehe [He00b]) Port-Constraints zur Verfügung: Über einen Port vom Typ `(Port a)` können n Schreiber Nachrichten vom Typ `a` an einen Leser versenden. Ein Port muß zunächst mit der Funktion

```
openPort :: Port a -> [a] -> Success
```

geöffnet werden. Durch Auswertung des Constraints `(openPort p s)` wird `p` an einen neuen Port und `s` an dessen Ausgabestrom gebunden. Die Schreiber können nun mit der Funktion

```
send :: a -> Port a -> Success
```

eine Nachricht an den Port versenden. Der Leser empfängt diese, indem er vom Ausgabestrom `s` liest.

2.5. Objekte

In vielen Anwendungen werden Zustände benötigt, diese lassen sich in deklarativen Sprachen jedoch zunächst nur sehr umständlich realisieren. Mit den im letzten Abschnitt vorgestellten Mitteln zur nebenläufigen Programmierung ergibt sich jedoch eine neue elegantere Möglichkeit: Objekte.

Ein Objekt ist eine selbstrekursive Funktion mit folgendem Typ:

```
[Msg] -> State -> Success
```

Es empfängt über einen Strom (das erste Argument) Nachrichten von einem bestimmten Datentyp. Danach kann der Zustand (das zweite Argument), entsprechend der Nachricht geändert werden, und das Objekt ruft sich selbst mit dem Reststrom und dem eventuell neuen Zustand wieder auf. Der Zustand kann von einem anderen Thread mit einer entsprechenden Nachricht gesetzt werden. Abgefragt wird er durch das Versenden einer freien Variable, die vom Objekt entsprechend gebunden wird. Eine Applikation mit einem Objekt besteht somit aus zwei Threads, die über Ports miteinander kommunizieren. Der eine führt die eigentliche Berechnung durch, der andere hält den Zustand.

Bsp. 2.5.1 *In diesem Beispiel wird das Objekt* `stringObject` *definiert. Mit den Nachrichten* `Set` *und* `Get` *kann ein String gesetzt und abgefragt werden.*

```
data StringMsg = Set String | Get String | Terminate

stringObject :: [StringMsg] -> String -> Success
stringObject eval rigid
stringObject (Terminate:_) _ = success
stringObject ((Set s):msgs) _ = stringObject msgs s
```

```
stringObject ((Get s1):msgs) s2 = s1=:=s2 &> stringObject msgs s2

test = let p,s,res free in openPort p s &>
                            (stringObject s "" &
                             (send (Set "State") p &>
                              send (Get res) p &>
                              send Terminate p))
```

Die Auswertungsannotation `rigid` *darf nicht weggelassen werden, da Funktionen mit Ergebnistyp* `Success` *standardmäßig die Annotation* `flex` *haben. Die Funktion* `test` *öffnet zunächst einen Port und ruft dann* `stringObject` *auf. Dann wird der Zustand einmal gesetzt und abgefragt.*

2.6. Ein- und Ausgabe

Aufgrund der unvermeidlichen Seiteneffekte war es lange Zeit unklar, wie Ein-/Ausgabe in deklarativen Sprachen realisiert werden kann. Curry bietet genau wie Haskell monadische Ein- und Ausgabe an.

Eine Funktion, die Ein- oder Ausgabe ausführt, also Seiteneffekte hat, und einen Wert vom Typen `a` zurück gibt, muß den Typ `(IO a)` haben. Eine solche Funktion führt wenn sie ausgewertet wird eine Sequenz von IO-Aktionen aus. Für die Eingabe gibt es die Aktionen

```
getChar :: IO Char
getLine :: IO String
readFile :: String -> IO String.
```

Für die Ausgabe gibt es

```
putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()
writeFile :: String -> String -> IO ().
```

Wenn eine solche Aktion ausgewertet wird, findet die entsprechende Interaktion mit der Umwelt statt. Mit den beiden Operatoren

```
(>>) :: IO _ -> IO a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
```

können IO-Aktionen kombiniert werden: `(>>)` nimmt zwei Aktionen und gibt die Aktion zurück, die diese beiden Aktionen hintereinander ausführt. `(>>=)` hat eine Aktion und eine Funktion als Argumente. Es ergibt sich eine Aktion, die das Ergebnis der ersten Aktion an die Funktion weitergibt und dann die sich daraus ergebende Aktion ausführt.

Die Einführung von IO-Aktionen hat einen entscheidenden Vorteil: Falls es zur Ein-/Ausgabe ganz normale Funktionen (z.B. `readInt :: Int`) gäbe, könnten diese in allen anderen Funktionen benutzt werden und Seiteneffekte erzeugen. So wird das Auftreten von Seiteneffekten auf wenige spezielle Funktionen beschränkt.

Aus offensichtlichen Gründen müßen Funktionen mit Ergebnistyp `(IO a)` deterministisch sein. Damit die Vorteile von nichtdeterministischen Funktionen trotzdem genutzt werden können, gibt es Konstrukte mit denen nichtdeterministische Berechnungen in deterministische eingebettet werden können (siehe [He00a]).

2.7. Verteilte Programmierung

Prozess- und rechnerübergreifende Programmierung wird durch externe Ports unterstützt, die von PAKCS angeboten werden. Dazu gibt es die folgenden beiden Funktionen:

```
openNamedPort :: String -> IO [a]
connectPort :: String -> IO (Port a)
```

Mit der IO-Aktion `openNamedPort n` wird ein neuer externer Port mit dem symbolischen Namen `n` eröffnet. Es wird der Ausgabestrom des Ports zurückgegeben, über den dann Nachrichten von anderen Curry-Prozessen empfangen werden können. Mit der Aktion `connectPort` kann sich jeder Curry-Prozess, der den entsprechenden symbolischen Namen kennt, mit dem Port verbinden und ganz normal mit `send` Nachrichten verschicken.

2.8. Listen

Listen sind eine Datenstrukutur, die in Curry sehr oft Verwendung finden. Aus diesem Grunde werden sie hier kurz erläutert. Eine Liste ist ein Behälter, der eine Variable Anzahl von Elementen gleichen Typs aufnehmen kann und sequentiellen Zugriff auf diese erlaubt:

```
data List a = (:) a (List a) | []
```

Die Typapplikation `(List a)` wird meistens durch `[a]` abgekürzt. Als Datenkonstruktoren gibt es den Infix-Konstruktor `:` und die leere Liste `[]`. Der Datenterm `(x:xs)` bezeichnet eine Liste mit dem Kopfelement `x` und der Restliste `xs`. Es gibt eine Reihe von vordefinierten Funktionen für Listen: Die Funktion

```
map :: (a->b) -> [a] -> [b]
```

wendet eine Funktion auf alle Elemente einer Liste an. Die Funktion `foldr` verknüpft alle Elemente einer Liste mit einer beliebigen Funktion:

```
foldr :: (a->b->b) -> b -> [a] -> b
```

Dabei ist das erste Argument die Verknüpfungsfunktion und das zweite der Wert, der bei der leeren Liste zurückgegeben werden soll. Mit

```
concat :: [[a]] -> [a]
```

wird eine Liste von Listen durch Konkatenation dieser in eine geglättete Liste verwandelt. Mit dem Infix-Operator `++` können zwei Listen konkateniert werden:

```
(++) :: [a] -> [a] -> [a]
```

Mit dem Infix-Operator

```
(!!) :: [a] -> Int -> a
```

kann auf ein beliebiges Element einer Liste zugegriffen werden.

2.9. Implementierungen

Es gibt eine grössere Anzahl an Implementierungen von Curry. Hier sei nur die PAKCS-Distribution erwähnt (siehe [He00b]), welche für diese Arbeit benutzt wurde. Sie bietet unter anderem Port-Constraints an und enthält nützliche Bibliotheken zur GUI- und Meta-Programmierung (siehe Kapitel 4,5). Es gibt ein interaktives System, mit dem man Curry-Programme übersetzen und ausführen kann. Der Compiler übersetzt ein Curry-Programm in ein Prolog-Programm, welches dann von einem Prolog-System ausgeführt werden kann. Da Curry noch eine recht junge Sprache ist, haben alle Implementierungen noch kleinere Defizite, erlauben jedoch trotzdem die Erstellung brauchbarer Applikationen. So fehlt zum Beispiel bei der PAKCS-Distribution das Modul-System.

3. Graphische Programmierumgebungen

3.1. Allgemeines

Graphische Programmierumgebungen integrieren verschiedene Werkzeuge , die bei der Programmerstellung nützlich sind, in eine einheitliche Oberfläche. Sie erhöhen die Produktivität und dienen der Vermeidung von Fehlern: Häufig benötigte Funktionen können sehr schnell ausgeführt werden, sich wiederholende Arbeitsabläufe können automatisert werden, Programmierfehler (z.B. Syntax-Fehler) werden teilweise früher erkannt, die Fehlersuche wird unterstützt. Ein weiterer Nutzen ergibt sich aus der einfacheren Kombination verschiedener Werkzeuge.

Typischerweise besteht eine solche Umgebung aus folgenden Werkzeugen:

- Editoren
- Compiler
- Debugger
- Code-Generatoren
- Projektverwalter
- (kontextsensitive) Hilfe
- Diverse andere sprachabhängige Werkzeuge

Ein Editor ermöglicht die (textuelle) Eingabe eines Programms, kann aber auch zusätzliche Funktionalität zur Fehlervermeidung haben. Compiler überprüfen die Korrektheit eines Programms und übersetzen es in eine andere Sprache. Sie sollen im Fehlerfall möglichst präzise Fehlermeldungen erzeugen. Debugger unterstützen das Auffinden und Beseitigen von Programmierfehlern. Code-Generatoren erzeugen Codegerüste für spezielle Anwendungsfälle (z.B. graphische Oberflächen). Die Projektverwaltung verwaltet alle zu einem Projekt gehörenden Dateien und Abhängigkeiten zwischen diesen. Die Hilfefunktion bietet Informationen zu bestimmten Themenbereichen, die die Programmiersprache oder die Programmierumgebung betreffen, an. Die kontextsensitive Hilfe versucht, anhand des Quellcodes eines Programms relevante Hilfethemen auszuwählen. Weiterhin gibt es eine Vielzahl von Wekzeugen für spezielle Klassen von Programmiersprachen (z.B. Klassenbrowser für objektorientierte Sprachen).

Die Funktionalität und die Arbeitsweise dieser Wekzeuge hängen natürlich teilweise stark von der unterstützten Programmiersprache ab. Dies gilt insbesondere für Debugger.

Im Folgenden werden verschiedene Werkzeuge für imperative, logische und funktionale Programmiersprachen vorgestellt.

3.2. Editoren

Der (Text-)*Editor* ist das zentrale Werkzeug zur Programmierung und dient der Eingabe des Quelltextes. Für die Erstellung großer Systeme ist es wünschenswert, dass der Quelltext mehrerer Module nebeneinander bearbeitet werden kann, das heißt es sollten mehrere Editoren gleichzeitig geöffnet werden können. Die meisten Editoren bieten eine Menge zusätzlicher Funktionen: Mit den Funktionen Auschneiden, Kopieren und Einfügen (cut/copy/paste) können vorher markierte Text-Blöcke markiert werden. Es kann nach bestimmten Zeichenketten gesucht werden, welche dann durch andere ersetzt werden können (search/replace). Alle Aktionen können wieder rückgängig gemacht werden (undo).

Neben diesen allgemeinen Funktionen eines Editors, gibt es auch noch andere, die ein frühzeitiges Erkennen von Syntax-Fehlern unterstützen: Syntax-Färbung und automatisches Einrücken. Durch Färbung bestimmter Teile des Quelltextes (zum Beispiel Token mit einer festgelegten Bedeutung wie <if>) wird die Lesbarkeit deutlich verbessert. Tippfehler können evtl. sofort erkannt werden (zum Beispiel bei Eingabe von "ig"). Automatisches Einrücken dient ebenfalls der besseren Lesbarkeit und kann auf fehlende Begrenzer (wie "END;" oder "}") aufmerksam machen. Somit können kleine Fehler schon zu einem sehr frühen Zeitpunkt behoben werden. Allerdings ist die Implementierung solcher syntaxgesteuerter Editoren recht aufwendig: Es wird ein spezieller Parser benötigt. Dieser muß zwar nicht unbedingt alle syntaktischen Konstrukte der Programmiersprache erkennen können, aus Gründen der Effizienz sollte es jedoch ein inkrementeller Parser sein, der nicht nach jeder Tastatureingabe den gesamten Quelltext parsen muß.

Neben Text-Editoren gibt es auch noch graphische Editoren: Der Programmierer erstellt gewisse graphische Objekte, die von Code-Generatoren in den entsprechenden Quelltext übersetzt werden. Meistens werden sie zur Erstellung von graphischen Benutzerschnittstellen eingesetzt.

3.3. Compiler

Ein *Compiler* übersetzt das mit einem Editor erstellte Quellprogramm in ein Zielprogramm. Dazu wird das Quellprogramm ausführlich analysiert. Die unvermeidlichen Fehlermeldungen sollten so präzise und ausführlich wie möglich sein.

Compiler können auch Optimierungen am Programm durchführen, oder versuchen Probleme, die erst zur Laufzeit offensichtlich werden, schon zur Übersetzungszeit aufzudecken (zum Beispiel nicht initialisierte Variablen) und somit die Programmentwicklung unterstützen. Durch getrennte Übersetzung oder inkrementelle Compiler kann der Recompilationsaufwand und somit die Entwicklungszeit erheblich reduziert werden.

3.4. Debugger

Da Fehler bei der Programmerstellung unvermeidlich sind, ist die Unterstützung von Fehlersuche und Fehlerbeseitigung durch geeignete Werkzeuge wichtig. Diese Werkzeuge werden *Debugger* genannt. Der Begriff "Debugger" ist allerdings nicht eindeutig bestimmt: Nach [Sha82] besteht ein Debugger aus je einem Werkzeug zur Diagnose und Korrektur, d.h. er muß sowohl Fehler erkennen als auch Fehler korrigieren können. Allerdings wurden bis zum heutigen Zeitpunkt hauptsächlich Werkzeuge zur Fehlererkennung entwickelt. Der Nutzen einer automatischen Fehlerkorrektur ist zudem fragwürdig: Unter Umständen versteht der Benutzer die Korrektur gar nicht und möglicherweise leider die Programmstruktur unter den Korrekturen. Weiterhin können lokale Korrekturen wieder neue Fehler hervorrufen. Aus diesen Gründen werden in der Literatur auch reine Diagnose-Werkzeuge als Debugger bezeichnet. Zudem werden manchmal auch Werkzeuge als Debugger bezeichnet, die die Fehlersuche überhaupt nicht unterstützen (zum Beispiel "Performance-Debugger"). Im Folgenden wird die Bezeichnung "Debugger" sehr allgemein gebraucht.

Es gibt eine Menge verschiedener Konzepte für Debugger. Die einfachsten beschränken sich lediglich auf die Visualisierung des Programmablaufes. Diese sogenannten *Tracer* bieten meistens die folgenden Funktionen an:

- Schrittweise Ausführung
- Anzeigen von Speicherinhalten
- Unterbrechungspunkte

Die Arbeitsweise eines Tracers für eine spezifische Programmiersprache orientiert sich natürlich stark an deren operationeller Semantik. Der Nutzen ist teilweise jedoch begrenzt: Der Programmablauf kann sehr komplex sein, operationelle Semantiken deklarativer Sprachen sind oft schwer verständlich. Insgesamt erhält der Entwickler relativ wenig Unterstützung, die Fehlerquelle muß er immer noch selbst auffinden, was durch die Visualisierung jedoch vereinfacht wird.

Algorithmisches Debugging ist eine Technik, die eine automatische Lokalisierung des Fehlers im Quelltext ermöglicht. Nachdem der Debugger das Programm analysiert hat, wird der Programmierer nach dem korrekten Ergebnis bestimmter Funktionsaufrufe gefragt. Eine solche Frage könnte z.B. so aussehen:

> Ist `[1,2,3]` das korrekte Ergebnis zu dem Funktionsaufruf `append [1,2] [3]`?

Mit den so gesammelten Informationen kann der Fehler dann lokalisiert werden: Falls eine Funktion `f` das falsche Ergebnis liefert, alle Funktionen, die `f` benutzt, jedoch das richtige Ergebnis liefern, muß der Fehler in der Definition von `f` liegen. Diese Technik wurde zuerst 1982 von Ehud Y. Shapiro zum Auffinden logischer Fehler in Prolog-Programmen vorgestellt. Sie wurde jedoch auch auf imperative und funktionale Sprachen übertragen.

Debugger müßen nicht ausschließlich die Lokalisierung von Fehlern unterstützen. Einige Werkzeuge helfen auch bei der Entdeckung von Fehlern indem sie beispielsweise eine (minimale) Überdeckung aller Programmpfade berechnen.

3.4.1. Debugger für imperative Programmiersprachen

Für nahezu alle imperativen Programmiersprachen stehen Tracer zur Verfügung. Sie unterstützen schrittweise Ausführung von Programmen, Unterbrechungspunkte und das Anzeigen von Speicherinhalten. Als Beispiel wird hier der Debugger `jdb` für die Sprache Java betrachtet, der Teil des Java Development Kits ist. Nach dem Aufruf des Debugers mit `jdb` *Progname* stehen folgende Kommandos zur Verfügung:

1. Mit dem Kommando `run` wird die Ausführung des Programms gestartet.
   (Dies ist erst dann sinnvoll, wenn bereits Unterbrechungspunkte definiert wurden.)

2. `stop at` *line* bewirkt, dass die Ausführung bei Erreichen von Programmzeile *line* unterbrochen wird.

3. Mit `cont` wird die Ausführung des Programmes nach Erreichen eines mit `stop` definierten Unterbrechungspunktes fortgeführt.

4. `print` v zeigt den aktuellen Wert der Variablen v an.
   (Falls v keinen skalaren Typ hat muß `dump` benutzt werden.)

5. Mit dem Kommando `step` kann das Programm schrittweise ausgeführt werden.

Weiterhin gibt es noch Kommandos, die spezielle Eigenschaften von Java unterstützen:

1. `thread` *threadindex* wird der Thread mit dem Index *threadindex* zum aktiven Thread.

2. `catch` *exception* bewirkt, dass die Ausführung des Programms durch Auslösung der Ausnahme *exception* unterbrochen wird.

3. `gc` startet den garbage collector.

Dies sind die Funktionen, die ein Tracer für imperative Sprachen üblicherweise bietet. In graphischen Programmierumgebungen sollten sie schnell und einfach aufgerufen werden können, und ihre Ergebnisse sollten übersichtlich dargestellt werden.

Manche Debugger erlauben auch die dynamische Änderung von Programmvariablen zur Laufzeit. Falls beispielsweise entdeckt wird, dass eine Funktion den falschen Wert zurück gibt, kann dieser in den Korrekten geändert werden um somit den Rest des Programms auf Korrektheit zu überprüfen. Der GNU Debugger unterstützt dies mit folgendem Kommando:

`set` v=*newvalue*

Obwohl auch algorithmische Debugger für imperative Sprachen entwickelt wurden, besitzen diese keine große Bedeutung. Dies liegt hauptsächlich daran, dass hier Tracer bereits gute Ergebnisse liefern. Im Gegensatz zu deklarativen Sprachen gibt es keinen so großen Unterschied zwischen der deklarativen Semantik, die die Grundlage für die Programmerstellung ist, und der operationellen Semantik, die von Tracern visualisiert wird.

3.4.2. Debugger für logische Programmiersprachen

Auch für logische Programmiersprachen wurden Tracer entwickelt, die den Beweisverlauf visualisieren. Sie basieren meistens auf dem *Box-Modell* (siehe Abb. 3.1, [Han87]). Für jedes Literal, das bewiesen werden soll, wird eine Box mit zwei Eingängen und zwei Ausgängen erzeugt. Die Box wird beim ersten Beweisversuch über den Eingang call betreten, bei jedem weiteren Versuch wird der Eingang redo benutzt. Nach einem erfolgreichen Beweisversuch wird sie über den Ausgang exit verlassen, falls kein Beweis für das Literal existiert, wird der Ausgang fail genommen. Bei jedem Betreten oder Verlassen einer Box macht der Tracer eine entsprechende Ausgabe.

Bsp. 3.4.1 *Als Beispiel wird folgendes Prolog-Programm betrachtet:*

```
append([],E,[E]).
append([X|Xs],E,[X|Xse]) :- append(Xs,E,Xse).

last(Le,E) :- append(L,E,Le).
```

Die Visualisierung des Beweisverlaufs für das Literal `last([1,2],X).` *sieht dann wie folgt aus:*

```
Call: ( 5) last([1,2],G700) ? creep
Call: ( 6) append(L436,G700,[1,2]) ? creep
Call: ( 7) append(G1100,G700,[2]) ? creep
Exit: ( 7) append([],2,[2]) ? creep
Exit: ( 6) append([1],2,[1,2]) ? creep
Exit: ( 5) last([1,2],2) ? creep

X = 2 ;
Redo: ( 7) append(G1100,G700,[2]) ? creep
Call: ( 8) append(G1112,G700,[]) ? creep
Fail: ( 8) append(G1112,G700,[]) ? creep
Redo: ( 7) append(G1100,G700,[2]) ? creep
Fail: ( 7) append(G1100,G700,[2]) ? creep
Redo: ( 6) append(L436,G700,[1,2]) ? creep
Fail: ( 6) append(L436,G700,[1,2]) ? creep
Redo: ( 5) last([1,2],G700) ? creep
Fail: ( 5) last([1,2],G700) ? creep
```

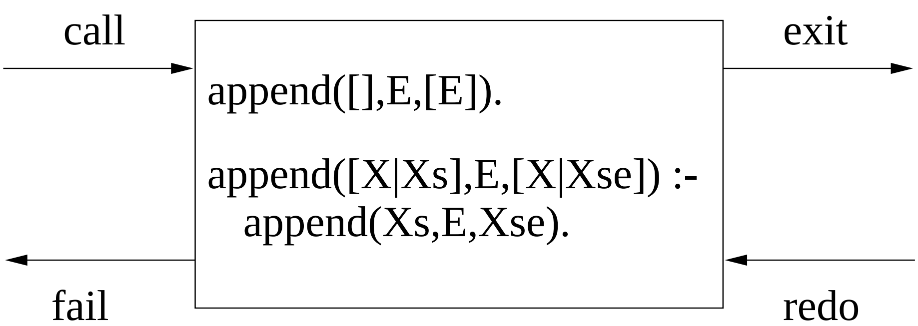

Abbildung 3.1.: Das Box-Modell

Jeder Box wird eine neue Nummer zugewiesen. Variablenbindungen werden sofort sichtbar. Am Ende eines Beweises werden die Bindungen der Anfragevariablen ausgegeben. Nach Eingabe von ; *wird nach weiteren Lösungen gesucht. Mit dem Kommando* `creep` *wird der nächste Schritt ausgeführt.*

Die PAKCS-Distribution enthält einen Debugger für Curry, der auf dem Box-Modell basiert.

Der praktische Nutzen solcher Debugger ist jedoch recht begrenzt: Der Programmierer muß sich mit der operationellen Semantik auseinandersetzen. Der Programmablauf großer Programme ist meistens sehr komplex und ist auch wegen des Backtrackings schwer durchschaubar. Algorithmische Debugger, auf die im nächsten Abschnitt eingegangen wird, sollen diese Probleme beheben.

3.4.3. Debugger für funktionale Programmiersprachen

Visualisierung von Graph-Reduktion

Ein möglicher Ansatz für einen Tracer für eine funktionale Sprache ist die Visualisierung der Termersetzung. Dazu wird ein Term als Baum dargestellt.

Bsp. 3.4.2 *Gegeben sei folgendes Haskell-Programm:*

```
double x = x+x
```

Die Auswertung des Terms

```
double (1+2)
```

kann beispielsweise wie folgt als Folge von Baümen visualisiert werden:

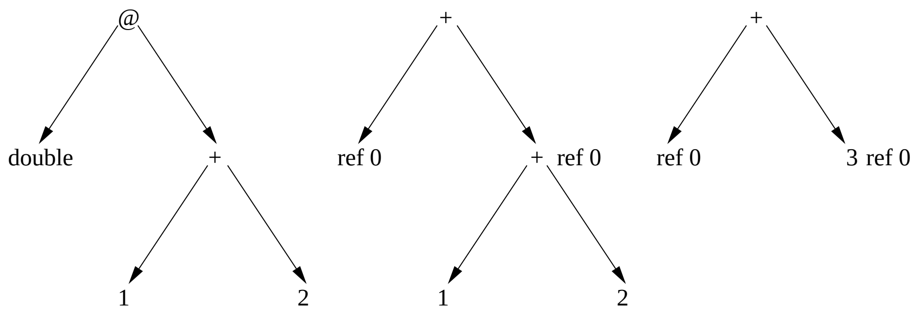

Abbildung 3.2.: Baum-Reduktion

Jeder Teilterm wird nur an einer Stelle als Baum dargestellt, an allen anderen Stellen wird lediglich eine Referenz eingetragen. Der Baum stellt somit also eigentlich einen Graphen dar. Eine Darstellung als Graph ist jedoch ungünstig, da sie weniger übersichtlich ist. Diese Technik wurde in [FR95] vorgeschlagen, wo dann auch von Graph-Baümen geredet wird.

Bei der Verwendung von Graph-Baümen gibt es jedoch immer noch zwei Probleme: Die Baüme und die Anzahl der Reduktionsschritte sind für nicht-triviale Beispiele viel zu groß.

Zur Lösung des ersten Problems werden zwei Lösungen vorgeschlagen:

1. Die Größe der Bäume wird nicht reduziert, dafür wird in einem kleinen Fenster eine "Landkarte" des Baums angezeigt, mit der dann in einem zweiten großen Fenster navigiert werden kann.
2. Es werden gewisse Filter auf die Bäume angewendet. Dazu wird eine Äquivalenzrelation auf den Knoten definiert und äquivalente Knoten werden zu einem Knoten verschmolzen. Diese Äquivalenzrelation kann vom Benutzer über eine geeignete Meta-Sprache definiert werden.

Um die Anzahl der angezeigten Reduktionsschritte zu vermindern, werden ähnliche Konzepte wie die für imperative Sprachen (Unterbrechungspunkte) bereits vorgestellten vorgeschlagen: Der erste und letzte Baum werden stets angezeigt, d.h. initial wird die Auswertung nicht unterbrochen. Mit einer Meta-Sprache werden Bedingungen für das Anzeigen weiterer Bäume definiert. Somit kann beispielsweise festgelegt werden, dass die Auswertung stets bei Anwendung einer bestimmten Funktion stoppt.

Dieses Konzept für die Visualisierung von Graph-Reduktion wurde von Sandra P. Foubister und Colin Runciman für eine Teilsprache von Haskell implementiert. Das KIEL-System (Kiel Interactive Evaluation Laboratory) basiert auf der gleichen Idee: Es wird die Auswertung von Standard ML-Programmen dargestellt. Zusätzlich kann die Auswerungsstrategie (leftmost-innermost,leftmost-outermost) ausgewählt werden.

Redex-Trails

Ein anderer Ansatz zum Tracen funktionaler Programme sind Redex-Trails (siehe [SR97]. Die grundlegende Idee hierbei ist es, während der Ausführung des Programms gewisse Informationen aufzusammeln, und diese hinterher dem Benutzer in einer geeigneten Form zur Verfügung zu stellen.

Die entscheidende Frage bei der Auffindung von Fehlern ist meistens "Wie ist diese falsche Ausgabe entstanden?". Deshalb macht es Sinn, die Folge der Berechnungsschritte von hinten zu betrachten. Ein Redex-Trail ist eine Datenstruktur, die zu jedem während der Berechnung neu entstandenem Teilausdruck den Redex, durch den er entstanden ist (Eltern-Redex) enthält. Dadurch kann die Entstehung des Fehlers vom fehlerhaften Ausruck an sukzessive zurückverfolgt werden.

Bsp. 3.4.3 *Es wird folgendes Haskell-Programm betrachtet, welches die Gültigkeit eines Datums überprüft:*

```
daysIn Jan = 31
daysIn Feb = 28
     ⋮
dayIn Dec = 31

valid (Date d m) = daysIn m >= d
date = Date 31 Feb
main = valid date
```

Der Redex-Trail für den Funtionsaufruf `main` *sieht so aus:*

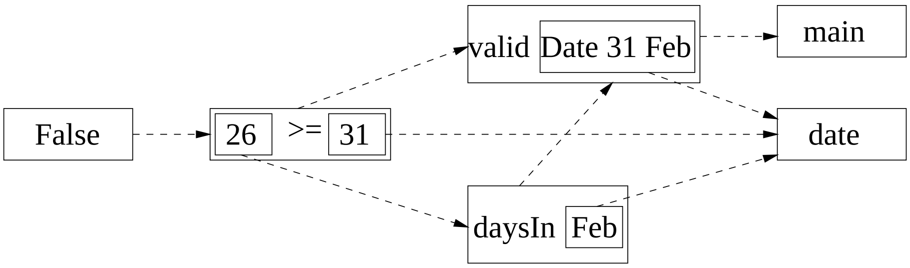

Abbildung 3.3.: Redex-Trail

Der Ausdruck `False` *entstand unmittelbar durch Reduktion von* `28 >= 31`*. Dessen Eltern-Redex ist wiederum* `valid (Date 31 Feb)`*, das Argument* `31` *hat* `date` *als Eltern-Redex und so weiter.*

Im Allgemeinen besteht ein Redex-Trail aus sehr vielen Knoten, die nicht alle gleichzeitig angezeigt werden können. Deshalb schlagen Jan Sparud und Colin Runciman vor, zunächst nur den Ausdruck anzuzeigen, bei dem die Auswertung aufhört. Der Benutzer kann dann durch Anklicken eines Ausdrucks das Anzeigen des entsprechenden Eltern-Redex bewirken.

Die einzelnen Ausdrücke können ebenfalls sehr groß werden. Deshalb werden Teilausdrücke ab einer bestimmten Tiefe durch Platzhalter ersetzt. Durch Anklicken dieser wird der der Ausdruck dann entsprechend expandiert. Damit die Beziehungen zwichen den einzelnen Ausdrücken noch gut erkennbar bleiben, wird jeder in einer einzigen Zeile dargestellt. Der Einsatz von Farben soll die Übersichtlichkeit weiter erhöhen.

Ein solcher Tracer kann durch eine Programmtransformation implementiert werden, die bewirkt, dass während der Programausführung der Redex-Trail aufgebaut wird. Erste Implementierungen haben gezeigt, dass dies jedoch recht teuer ist: Die Übersetzungszeit verdoppelt sich, die Laufzeit des Programms verschlechtert sich um den Faktor 6 bis 36.

Algorithmisches Debugging

Tracing ist eine Debugging-Technik, die für deklarative Programmiersprachen relativ schlecht geeignet ist. Bei imperativen Sprachen läßt sich die Reihenfolge der Auswertung, die von Tracern visualisiert wird, direkt am Quelltext erkennen. Einem Programmierer fällt es somit leicht, die Auswertung zu verfolgen. Bei deklarativen Sprachen (insbesondere bei funktionalen Sprachen mit fauler Auswertung) ist dies jedoch nicht so. Hier muß der Programmierer sich mit der komplizierten operationellen Semantik auseinandersetzen, ein wichtiger Vorteil deklarativer Sprachen entfällt. Algorithmische Debugger (oft auch als deklarative Debugger bezeichnet) versuchen, diesen Nachteil zu beheben, in dem sie lediglich auf die syntaktische Struktur eines Programmes Bezug nehmen. Im Folgenden wird algorithmisches Debugging für die funktionale Sprache Haskell vorgestellt, die Konzepte lassen sich jedoch leicht auf logische und imperative Sprachen übertragen.

Deklaratives Debugging besteht aus zwei Schritten: Zunächst wird zu dem Ausdruck, der fehlerhaft ausgewertet wird, ein Abhängigkeitsbaum erstellt, der die syntaktischen Abhängigkeiten des Programms reflektiert. Dieser wird dann zur Lokalisierung der Fehler

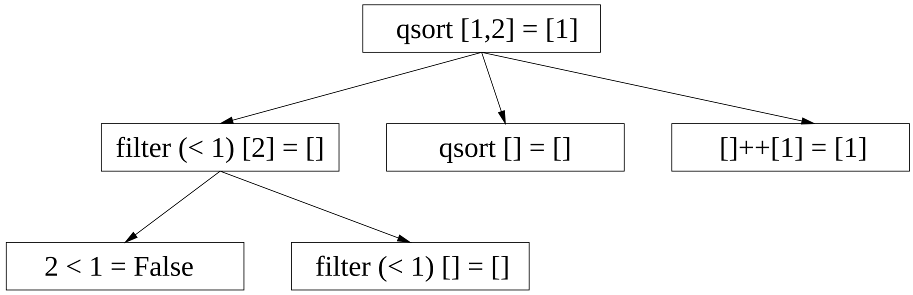

Abbildung 3.4.: Abhängigkeitsbaum

traversiert. Jeder Knoten des Abhängigkeitsbaums besteht aus einer Funktionsapplikation und deren Ergebnis, welches während der Ausführung des Programms berechnet wurde. Die Wurzel besteht also aus dem Ausdruck, der debuggt werden soll und dem (fehlerhaften) Ergebnis. Die Kinder eines Knotens bestehen aus den entsprechenden Knoten zu den Applikationen, die zur Berechnung des Ergebnisses des Vaterknotens notwendig sind.

Zur Lokalisierung von Fehlern wird nun ein Orakel benötigt: Die Traversierung des Baums startet an der Wurzel. Das Orakel wird gefragt, ob das vom Programm berechnete Ergebnis der Applikation in der Wurzel korrekt ist. Falls dies der Fall ist, wird die Fehlersuche abgebrochen, da sich das Programm korrekt verhält. Ansonsten wird mit den Kindern beginnend von links in gleicher Weise fortgefahren. Falls das Programm Fehler enthält, gibt es fehlerhafte Knoten (Knoten, deren berechnetes Ergebnis falsch ist), die keine fehlerhaften Kinder haben. Der Fehler muß dann in der Definition der Funktion liegen, die in diesem Knoten appliziert wird.

Bsp. 3.4.4 *Gegeben sei folgendes Haskell-Programm, welches Listen sortieren soll. Dem Ausdruck* `qsort [1,2]` *wird der Abhängigkeitsbaum in Abb.3.4 zugeordnet.*

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)

filter _ [] = []
filter p (x:xs) | p x = x : (filter p xs)
                | otherwise = filter p xs

qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs) ++
               x : (qsort (filter (< x) xs))
```

Da die Wurzel der einzige fehlerhafte Knoten ist, muß der Fehler in der Definition der Funktion `qsort` *liegen: Das Prädikat* `(< x)` *im zweiten Aufruf von* `filter` *ist falsch und muß durch* `(>= x)` *ersetzt werden.*

Die Rolle des benötigten Orakels wird meistens vom Benutzer selbst übernommen, d.h. in einer Debugging-Sitzung wird dieser nach der Korrektheit der berechneten Ergebnisse gefragt und erhält dann als Ergebnis die fehlerhafte Funktion. Dies muß jedoch nicht so sein: beispielsweise kann auch eine andere (evtl. ineffiziente) Implementierung des gleichen Problems als Orakel dienen.

Algorithmisches Debugging ist ein Verfahren, dass sich gerade für deklarative Sprachen sehr gut eignet. Leider gibt es auch Fehlerklassen, bei denen es versagt: Falls das fehlerhafte Programm kein Ergebnis liefert, weil es an einen Punkt gelangt an dem keine Regel anwendbar ist oder in eine Endlos-Schleife gerät, kann ein algorithmischer Debugger den Fehler nicht lokalisieren. Das System *Buddha* (siehe [Pop98]), welches von Bernard Pope entwickelt wurde, implementiert einen algorithmischen Debugger für Haskell.

3.4.4. Performance-Debugging

Um ein prinzipiell anderes Debugging-Prinzip vorzustellen, wird hier kurz der Oz-Explorer betrachtet. Oz ist eine nebenläufige, constraintbasierte Programmiersprache, die eingekapselte Suche unterstützt. Die Entwicklung von Oz-Programmen besteht meist aus zwei Schritten: Zunächst wird eine prinzipiell funktionierende Lösung erstellt. Dies ist relativ einfach, so dass eine Wekzeugunterstützung nicht unbedingt notwendig ist. Meistens ist die Effizienz dieser ersten Lösung jedoch sehr schlecht, so dass sie im zweiten Schritt noch optimiert werden muß. Dies ist der bedeutend schwierigere Schritt, der durch den Oz-Explorer unterstützt wird (Diese Art von Debugging wird auch Performance-Debugging genannt).

Der Programmierer erhält die Möglichkeit, den gesamten Suchbaum interaktiv zu durchsuchen. Somit bekommt er ein besseres Verständnis des Programmablaufes: Er kann erkennen wie viele Lösungen es gibt, wie diese Lösungen berechnet werden oder wie lange für eine Lösung gesucht werden muß. Dadurch ist er in der Lage, die Gründe für die schlechte Effizienz zu erkennen.

3.5. Projektverwaltung

Die meisten Programmierumgebungen bieten zusätzliche Unterstützung für die Verwaltung großer Projekte. Diese bestehen aus einer Menge von Quelltextdateien und anderen Dateien (z.B. Bild-Dateien oder Dokumentationen), die einheitlich behandelt werden sollen. In einer Projektdatei werden die entsprechenden Dateinamen und Beziehungen zwischen diesen abgelegt. Diese Informationen können dann von anderen Werkzeuge für verschiedene Zwecke genutzt werden. Weiterhin können jedem Projekt gewisse Eigenschaften wie z.B. Pfade in denen nach Bibliotheken gesucht werden soll, Kommandozeilen-Parameter oder Debugging-Optionen zugewiesen werden.

Die meisten Projekte werden von mehreren Entwicklern gleichzeitig bearbeitet, deshalb haben die meisten Programmierumgebungen eine Versionskontrolle. Diese protokolliert alle Änderungen am Projekt mit und erkennt Konflikte, die durch gleichzeiteige Bearbeitung des gleichen Quelltextes entstehen können.

3.6. Code-Generatoren

Für bestimmte Anwendungsfälle gibt es Code-Generatoren, die auf Basis einer einfacheren meist graphischen Programmdefinition den entsprechenden Quelltext für die unterstützte Sprache generieren. Die einfachsten Generatoren erzeugen einfach ein oft benutztes

Code-Gerüst, beispielsweise das Gerüst einer Klassen-Definition einer objektorientierten Sprache.

Für die Erstellung von graphischen Benutzerschnittstellen (GUI) gibt es eine Reihe von Generatoren: In sogenanten *visuellen Programmiersprachen* kann die Benutzerschnittstelle mit einem graphischen Editor festgelegt werden. Dazu müßen lediglich die entsprechenden Elemente (z.B. Fenster, Knöpfe oder Menüs) ausgewählt und positioniert werden. Einzelnen Teilen des GUI können bestimte Code-Fragmente als Event-Handler zugewiesen werden. Schließlich erzeugt der Code-Generator den Quelltext in der entsprechenden Sprache.

Während die meisten Code-Generatoren nur für bestimmte Anwendungsfälle gedacht sind, gibt es auch Sprachen, die vollständig ohne textuelle Programmdefinition auskommen. In *Visual Haskell* (siehe [Ree94]) kann die folgende Haskell-Funktion

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

wie folgt definiert werden:

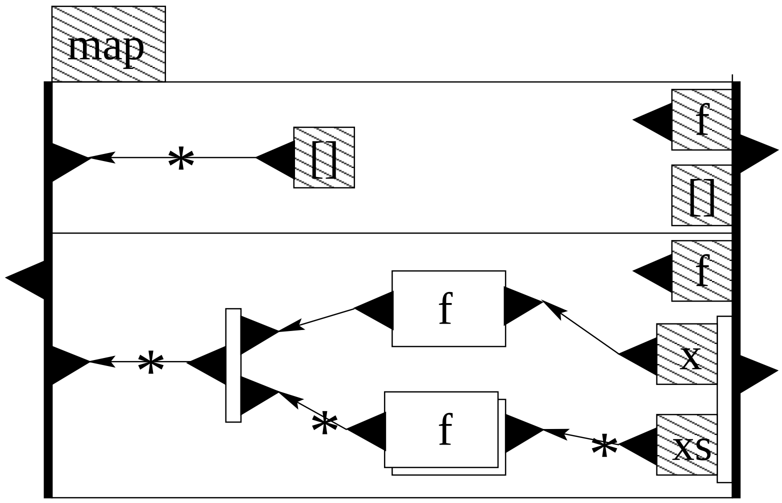

Abbildung 3.5.: Definition von `map` in Visual-Haskell

Auch ohne genaueres Verständnis ist klar, dass diese Notation zumindest zur Programmierung ungeeignet ist.

4. Graphische Benutzerschnittstellen

4.1. Allgemeines

Für fast jede Programmiersprache gibt es Bibliotheken, die das Erstellen von graphischen Benutzerschnittstellen (GUI) unterstützen. Allerdings gibt es bei diesen (genau wie bei den Programmiersprachen selbst) große Unterschiede. Die Struktur einer GUI-Definition hängt natürlich stark von den Ausdrucksmöglichkeiten der Sprache ab.

Die GUI-Bibliotheken imperativer Sprachen haben meist folgende Nachteile:

1. Die Definition eines GUI besteht aus einer (meist sehr langen) Sequenz von Befehlen, an der man die tatsächliche Struktur des GUI nicht erkennen kann.

2. Um Teile des GUI referenzieren zu können, müßen diesen Namen zugeordnet werden. Dabei kann es leicht zu Tippfehlern kommen, die nicht zur Compilezeit erkannt werden können.

Dadurch wird die Wartbarkeit und Erweiterbarkeit eines Programms mit graphischer Benutzerschnittstelle erschwert.

4.2. Die Tk-Bibliothek

Im Folgenden wird die Tk-Bibliothek für Curry vorgestellt, die Teil der PAKCS-Distribution ist. Sie baut auf Tcl/Tk auf und wurde mit folgenden Zielen entwickelt:

1. Da noch keine GUI-Bibliothek für Curry vorhanden war, war es an der Zeit dieses Defizit zu beseitigen.

2. Es sollten möglichst viele Konzepte von Curry ausgenutzt werden, um die oben beschriebenen Nachteile der Bibliotheken imperativer Sprachen zu vermeiden.

3. Um sich auf das Wesentliche konzentrieren zu können und möglichst schnell zum Ziel zu kommen, sollte die Bibliothek auf eine schon vorhandene Bibliothek aufbauen.

Diese Ziele wurden zwar erreicht, allerdings gibt es auch noch einige Unzulänglichkeiten: Es wird deutlich werden, dass die Tk-Bibliothek im Moment noch etwas unflexibel ist und die Funktionalität noch nicht vollständig ausreicht.

4.2.1. Tcl/Tk

Tcl (Tool Command Language) ist eine ebenso einfache wie beliebte Skriptsprache, die 1988 von J.K. Ousterhout entwickelt wurde. Ihre Beliebtheit ist nicht zuletzt durch die Tk-Erweiterung begründet, die die Programmierung graphischer Benutzerschnittstellen erlaubt. Tcl/Tk wird nicht nur als Skriptsprache zur Systemintegration benutzt. Häufig werden auch komplette Applikationen in Tcl/Tk implementiert. Dies hat eine Reihe von Vorteilen: Die Einfachheit ermöglicht ein einfaches Erlernen und sorgt für relativ kurze Entwicklungszeiten. Tk abstrahiert von vielen Details des zugrundeliegenden Fenster-Systems. Es gibt Tcl/Tk-Implementierungen für diverse Betriebssysteme, insbesondere für verschiedene Unix-Varianten und Windows.

Tcl/Tk ist eine interpretierte Sprache, dies ist der entscheidende Grund, sie als Grundlage für die Curry-Bibliothek zu nehmen. Eine Curry-Applikation kann mit dem Tcl/Tk-Interpreter *wish* (windowing shell) durch das Versenden von Strings über Ports kommunizieren. Dazu gibt es die vordefinierte Funktion

```
openProcessPort :: String -> IO (Port ppMsg).
```

Durch den Funktionsaufruf `(openProcessPort cmd)` wird das Shell-Kommando `cmd` (zum Beispiel `wish` zum Starten des Tcl/Tk-Interpreters) ausgeführt und ein Port zurückgegeben, über den man mit dem entsprechenden Prozess komunizieren kann. Der entsprechende Unix-Prozess hat dann je einen Ein- und Ausgabestrom. Mit drei verschiedene Arten von Nachrichten, die man über den Port versenden kann, kann dann kommuniziert mit diesem werden:

```
data ppMsg = ppPut String | ppGet String | ppClose
```

Das Versenden der Nachricht `(ppPut s)` bewirkt das Hinzufügen des Strings `s` zum Eingabestrom des externen Prozesses, beim Versenden von `(ppGet s)` wird `s` mit dem nächsten Element des Ausgabestroms unifiziert, mit `ppClose` wird die Kommunikation beendet.

Die Aufgaben der Curry-Bibliothek sind nun die Folgenden:

1. Übersetzen der GUI-Definition in entsprechende Tcl/Tk-Befehle
   (Da GUI-Definitionen in Curry eine andere Struktur haben sollen, ist dies nicht trivial).
2. Versenden der Befehle zum Tcl/Tk-Interpreter.
3. Empfangen von Nachrichten (Events) vom Tcl/Tk-Interpreter.
4. Aufrufen der entsprechenden Event-Handler.

4.2.2. Benutzung der Bibliothek

Eine graphische Benutzerschnittstelle besteht aus einer Reihe verschiedener Elemente (zum Beispiel Knöpfe, Menüs, Text-Fenster,...), die *Widgets* genannt werden.

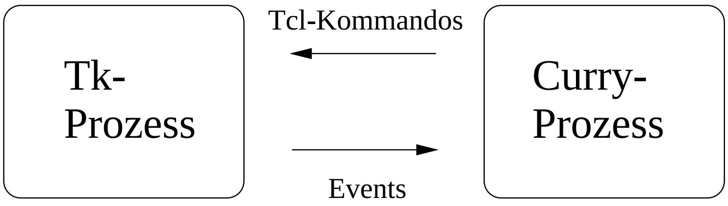

Abbildung 4.1.: Kommunikation mit dem Tcl/Tk-Interpreter

Zur Definition dieser Widgets wird der Datentyp `TkWidget` zur Verfügung gestellt:

```
data TkWidget a = TkButton (Port ppMsg -> a) [TkConfItem a]
                | TkListBox                  [TkConfItem a]
                | TkTextEdit                 [TkConfItem a]
                ⋮
                | TkRow [TkConfCollection] [TkWidget a]
                | TkCol [TkConfCollection] [TkWidget a]
```

Sinn und Zweck des Typparameters `a` werden noch deutlich werden. Mit den beiden Widgets `TkRow` und `TkCol` können andere Widgets (gegeben durch das zweite Argument) in Zeilen beziehungsweise Spalten angeordnet werden, mit dem ersten Argument dieser Widgets können noch zusatzliche Optionen (zum Beispiel zentrierte Ausrichtung) festgelegt werden. Die anderen Widgets entsprechen den allgemein üblichen Elementen graphischer Benutzerschnittstellen. Sie können über eine Konfigurationsliste (`[TkConfItem a]`) konfiguriert werden. Eine Ausnahme bilden Knöpfe (`TkButton`): Wenn ein Knopf gedrückt wird, soll eine Aktion ausgeführt werden, deshalb muß jeder Knopf einen Event-Handler (vom Typ `(Port ppMsg -> a)`) haben, der ausgeführt wird, falls der Knopf gedrückt wurde. Auch andere Widgets können Event-Handler haben, diese müssen dann in der Konfigurationsliste angegeben werden. Wann ein Event ausgelöst wird, hängt von der Art des Widgets ab. Meistens passiert dies bei einem einfachen Mouse-Click auf das Widget.

Event-Handler haben den Typ `(Port ppMsg -> a)`. Ihr Ergebnistyp, der entweder `IO ()` oder `Success` sein darf, ist der Typ der Aktion, die beim Eintreten eines Events ausgeführt wird. Jeder Event-Handler bezieht sich auf ein bestimmtes GUI. Zu jedem GUI gibt es einen Port vom Typ `(Port ppMsg)` über den es mit bestimmten Funktionen vom Typ `IO ()` oder `Success` verändert werden kann. Da diese Ports zur Übersetzungszeit noch nicht bekannt sind, muß jeder Event-Handler ein entsprechendes Argument vom Typ `(Port ppMsg)` haben. Zur Laufzeit wird er dann beim Eintreten des Events auf den entsprechenden Port appliziert.

Zur Konfiguration der Widgets gibt es den Datentyp `TkConfItem`:

```
data TkConfItem a = TkActive Bool
                  | TkCmd (Port ppMsg -> a)
                  | TkText String
                  | TkRef TkRefType
                  ⋮
                  | TkWidth Int
                  | TkHeight Int
```

Nicht alle Optionen sind für alle Widgets sinnvoll, unsinnige und nicht unterstützte Optionen werden einfach ignoriert. Insbesondere kann einem Knopf kein zweiter Event-Handler zugewiesen werden. Da die Bedeutung der meisten Optionen intuitiv klar sein sollte, wird hier nur auf die Option `TkRef` eingegangen. Damit ein Event-Handler die Konfiguration eines Widgets abfragen und abändern kann, muß es eine Möglichkeit geben, einzelne Teile des GUI zu referenzieren. Mit der Option `TkRef` kann man einem Widget eine Referenz zuordnen. Diese kann dann in speziellen Funktionen zum Setzen und Abfragen der Konfiguration dieses Widgets benutzt werden. Ein wichtiger Punkt ist, dass die Tk-Bibiliothek nur den Typ `TkRefType` und nicht die zugehörigen Konstruktoren exportiert. Der Programmierer kann also nur freie Variablen als Referenzen angeben. Diese werden beim Übersetzen des Widgets in die entsprechenden Tcl-Befehle an einen internen Bezeichner gebunden. Somit ist einer der oben erwähnten Nachteile behoben: Tippfehler, die in anderen Bibliotheken zu Laufzeitfehlern führen, werden so bereits zur Übersetzungszeit erkannt.

Benutzt werden Referenzen zum Beispiel mit den Funktionen

```
tkGetValue :: TkRefType -> Port ppMsg -> IO String
tkSetValue :: TkRefType -> String -> Port ppMsg -> IO ()
```

innerhalb von Event-Handlern mit Ergebnistyp `IO ()` und

```
tkCGetValue :: TkRefType -> Port ppMsg -> String -> Success
tkCSetValue :: TkRefType -> String -> Port ppMsg -> Success
```

innerhalb von Event-Handlern mit Ergebnistyp `Success`. Mit ihnen kann man den Wert eines Widgets (dessen Bedeutung unterschiedlich ist) abfragen beziehungsweise setzen. Dieser ist stets eine Zeichenkette, die aber eventuell noch interpretiert werden muß. So ist beispielsweise der Wert eines Listbox-Widgets die Nummer des gerade selektierten Eintrages. Um diese zu erhalten, muß der Wert, den man mit `tkGetValue` erhält, von `String` nach `Int` konvertiert werden. Die Konfigurationsoptionen einzelner Widgets können mit den Funktionen

```
tkConfig :: TkRefType -> TkConfItem _ -> Port ppMsg -> IO ()
tkCConfig :: TkRefType -> TkConfItem _ -> Port ppMsg -> Success
```

zur Laufzeit geändert werden. Leider können sie aus technischen Gründen nicht dazu verwendet werden, Event-Handler zu modifizieren. Dies ist auch der Grund, warum der Typparameter von `TkConfItem` keine Rolle spielt.

Eine so definierte graphische Benutzerschnittstelle kann achließlich mit der Funktion

```
runWidget :: String -> TkWidget (IO ()) -> IO ()
```

ausgeführt werden.

Bsp. 4.2.1 *Es wird die folgende einfache GUI-Definition betrachtet:*

```
w = TkRow [] [
       TkListBox [TkRef listbox, TkCmd listboxH,
                  TkList ["Zero","One","Two"]],
       TkCol [] [
         TkLabel [TkText "Editor"],
         TkTextEdit [TkRef textedit,TkWidth 30]
       ]
    ]
  where listbox,textedit free
        listboxH gp = tkGetValue listbox gp >>= \s->
                      tkSetValue textedit s gp
```

Das Widget `w` *besteht aus einer Listbox und einem Editor mit Überschrift. Die beiden freien Variablen* `listbox` *und* `textedit` *dienen als Referenzen. Der Listbox ist der Event-Handler* `listboxH` *zugeordnet. Dieser liest den Wert der Listbox (den Index des gerade selektierten Eintrages) aus und schreibt ihn in den Editor. Nach dem Aufruf von*

```
runWidget "GUI Example" w
```

erscheint die Benutzerschnittstelle wie sie in Abb. 4.2 gezeigt wird auf dem Bildschirm. In diesem Beispiel wird ein Event-Handler mit Ergebnistyp `IO ()` *benutzt, auf Event-Handler mit Ergebnistyp* `Success` *wird im nächsten Abschnitt eingegangen.*

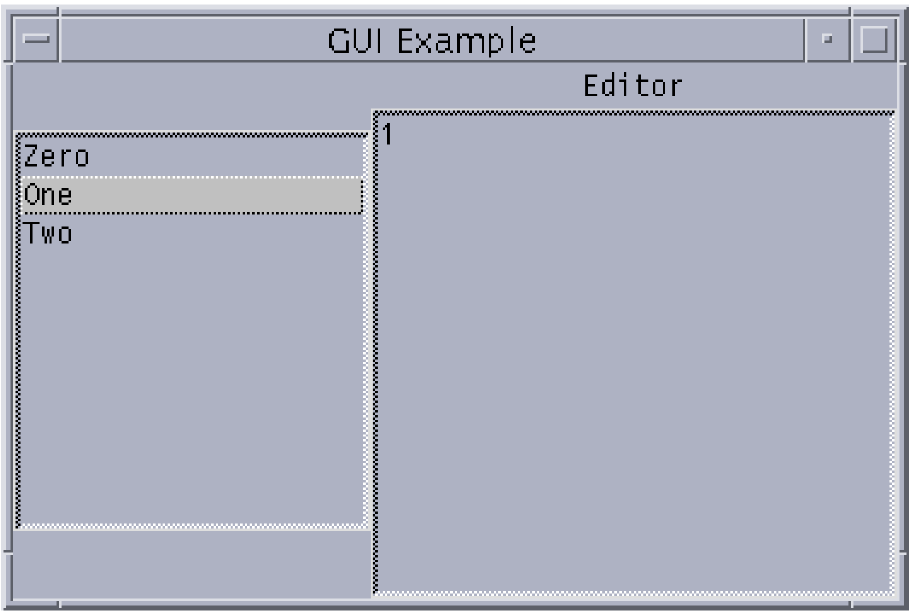

Abbildung 4.2.: Beispiel einer graphischen Benutzerschnittstelle

4.2.3. GUIs mit Zustand

Das vorgestellte Beispiel hat einen entscheidenden Nachteil: Es gibt keine Möglichkeit, einen Zustand zu halten. Zustände werden jedoch recht häufig benötigt (zum Beispiel bei der Implementierung von cut/copy/paste) und meistens mit Objekten implementiert. Hierzu muß der GUI-Thread nebenläufig zum Objekt-Thread ablaufen können, was durch Widgets vom Typ (`TkWidget Success`) ermöglicht wird. Mit der Funktion

```
runWidgetOnPort :: TkWidget Success -> Port ppMsg -> Success
```

kann eine GUI-Definition innerhalb eines Constraints und somit nebenläufig zu einem Objekt-Thread ausgeführt werden. Dazu muß zunächst mit der Funktion

```
openWish :: String -> IO (Port ppMsg)
```

der Tcl/Tk-Interpreter gestartet und ein entsprechender Port erschaffen werden. Vor dem Starten der beiden Threads muß noch ein (interner) Port `p` für die Kommunikation zwischen GUI und Objekt erschaffen werden. Insgesamt sieht der entsprechende Quellcode dann prinzipiell wie folgt aus:

```
startGUI gp | let p,s free in openPort p s &>
                              (serveState s initialState &
                               runWidgetOnPort (w p) gp) = done
start = openWish "GUI Example" >>= startGUI
```

Somit wird modulare Programmierung unterstützt, die Benutzerschnittstelle ist klar von der Programmlogik getrennt.

4.2.4. Probleme

Es wurde kurz die logisch-funktionale GUI-Bibliothek aus der PAKCS-Distribution vorgestellt. Diese nutzt sowohl die funktionalen als auch die logischen Eigenschaften von Curry aus und ermöglicht somit ein sehr komfortables Programmieren von graphischen Benutzerschnittstellen. Allerdings hat sie auch noch einige Defizite:

1. Die Event-Handler einzelner Widgets können nicht zur Laufzeit geändert werden.
2. Die Layout-Struktur der Oberfläche kann zur Laufzeit nicht modifiziert werden.
3. Mehrere Fenster können nicht gleichzeitig aktiv sein, es sind nur modale Dialog-Boxen möglich.

Für bestimmte Anwendungen werden diese Möglichkeiten jedoch benötigt. Um den Grund hierfür und mögliche Verbesserungen zu verstehen, ist eine genauere Kenntniss der Architektur dieser Bibliothek notwendig.

Wenn ein Widget ausgeführt wird, wird zunächst der Tcl/Tk-Interpreter gestartet. Danach wird innerhalb des Curry-Prozesses ein Scheduler gestartet. Dieser hat alle Event-Handler und den Port zum Tcl/Tk-Interpreter als Argumente und hat die Aufgabe, die Events, die über diesen Port gesendet werden, zu empfangen und den entsprechenden Event-Handler aufzurufen. Das Problem hierbei ist, dass er nur diesen einen Port abhören kann und somit keine Nachrichten vom Curry-Prozeß erhalten kann. Dies ist jedoch aus zwei Gründen wünschenswert: Erstens würde es die Modifikation von Event-Handlern erlauben. Zweitens muß an irgendeiner Stelle ein Index zur Vermeidung von Namenskonflikten gehalten werden, wenn die Layout-Struktur zur Laufzeit modifiziert werden soll. Hierfür ist der Scheduler gut geeignet, falls er auch vom Curry-Prozeß Nachrichten erhalten kann.

Die Ursache für dieses Problem liegt darin, dass der Scheduler nicht gleichzeitig den Tcl/Tk-Port und einenen anderen (internen) Port abhören kann. Eine Lösung wäre es, den Strom vom Tcl/Tk-Port mit einem anderen Strom von Nachrichten vom Curry-Prozeß zu einem Strom für den Scheduler zu mischen.

Die Programmiersicherheit wird zwar durch die Benutzung freier Variablen erhöht, es gibt jedoch immer noch ein Problem, wenn der Programmierer versehentlich eine Referenz mehrfach benutzt. In diesem Falle stürzt das Programm dann einfach ab und die Fehlersuche wird sehr schwierig. Dieses Problem kann jedoch nicht behoben werden: Eine freie Variable kann nur einmal gebunden werden, und es gibt in Curry keine Möglichkeit festzustellen, ob eine Variable bereits gebunden ist oder nicht.

5. Meta-Programmierung

Meta-Programme sind solche Programme, die andere Programme analysieren oder manipulieren. Dies können beispielsweise Optimierer, (partielle) Auswerter oder Analysatoren sein. Ihre Ziele sind unter anderem Verkürzung der Laufzeiten, Auffinden von Programmierfehlern (z.B. nicht initialisierte Variablen), Implementierung verschiedener Auswertungsstrategien (z.B. verzögerte Auswertung) oder Unterstützung der Fehlersuche (Debugging). Ohne entsprechende Bibliotheken oder Sprachkonstrukte sind Meta-Programme nur mit viel Aufwand zu implementieren. In einigen Sprachen gibt es jedoch die entsprechende Unterstützung: So ist es z.B. kein großes Problem, einen Prolog-Interpreter in Prolog zu implementieren. Die PAKCS-Distribution bietet in Form der Flat-Bibliotehk ebenfalls Unterstützung für Meta-Programmierung.

5.1. FlatCurry

FlatCurry ist eine Zwischensprache für Curry. Sie soll einerseits die Benutzung verschiedener Back-Ends mit einem einzigen Front-End ermöglichen und andererseits die Entwicklung von Werkzeugen (wie z.B. partielle Auswerter oder Programm-Analysatoren) für Curry erleichtern. In FlatCurry gibt es keine lokale Funktions- oder Patterndeklarationen mehr. Zudem wird die Behandlung von pattern matching wesentlich erleichtert. Jedes Curry-Programm kann durch zwei Programmtransformationen in ein FlatCurry-Programm übersetzt werden: Lokale Deklaration werden mittels lambda lifting eliminiert (siehe [He00a]), pattern matching wird in Case- und Or-Ausdrücke übersetzt (siehe Abschnitt 5.2). Die Implementierung von Back-Ends oder Werkzeugen auf der Basis von FlatCurry wird somit relativ einfach. Lokale Deklarationen brauchen nicht berücksichtigt zu werden, Case- und Or-Ausdrücke erleichtern die Implementierung von pattern matching. Allerding hat die Benutzung von FlatCurry auch Nachteile, da die Struktur eines Programmes durch lambda lifting stark geändert wird. Die genaue Syntax von FlatCurry kann in [He00b] nachgelesen werden.

Auch PAKCS basiert auf FlatCurry: Der Parser prüft die Korrektheit eines Programms und übersetzt es dann in ein FlatCurry-Programm. Der curry2prolog-Compiler liest dieses ein und übersetzt es in ein Prolog-Programm.

5.2. Die Flat-Bibliothek

Die Flat-Bibliothek ist Teil der PAKCS-Distribution und unterstützt Meta-Programmierung. Die Funktion

```
readFlatCurry :: String -> IO Prog
```

übersetzt das durch ihr Argument gegebene Curry-Programm in ein FlatCurry-Progamm und erzeugt von diesem eine interne Darstellung vom Typ `Prog`. Im Folgenden wird der algebraische Datentyp `Prog` kurz vorgestellt. Datenterme von diesem Typ repräsentieren FlatCurry-Programme.

Ein Programm besteht aus einem Namen und jeweils einer Liste von Importen, Typdeklarationen, Operatordeklarationen, und Funktionsdeklaration einer Übersetzungstabelle, die jedem Bezeichner einen internen Bezeichner zuordnet:

```
data Prog = Prog String [String] [TypeDecl] [OpDecl]
                 [FuncDecl] [Translation]
```

Eine Typdeklaration besteht aus einem Bezeichner für den Typkonstruktor, einer Liste von Typvariablen (die Argumente) und einer Liste von (Daten-)Konstruktordeklarationen. Typvariablen werden zur Vereinfachung durch einen Integer-Wert dargestellt:

```
type TVarIndex = Int
data TypeDecl = Type String [TVarIndex] [ConsDecl]
```

Eine Konstruktordeklaration ist durch einen Bezeichner, die Stelligkeit des Konstruktors und eine Liste von Typausdrücken gegeben:

```
data ConsDecl = Cons String Int [TypeExpr]
```

Ein Typausdruck ist entweder eine Typvariable, ein funktionaler Typ oder eine Applikation eines Typkonstruktors auf andere Typausdrücke:

```
data TypeExpr = TVar TVarIndex
              | FuncType TypeExpr TypeExpr
              | TCons String [TypeExpr]
```

Mit Operatordeklarationen werden Infix-Operatoren unterstützt, für die jeweils die Assoziativität und die Priorität deklariert werden müssen:

```
data Fixity = InfixOp | InfixlOp | InfixrOp
data OpDecl = Op String Fixity Int
```

Eine Funktionsdeklaration besteht aus einem Bezeichner, einem Integer-Wert für die Stelligkeit, einem Typausdruck und einer Regel.

```
data FuncDecl = Func String Int TypeExpr Rule
```

Da das pattern matching in `Case-` und `Or-`Ausdrücke übersetzt wurde, hat jede Funktionsdeklaration genau eine Regel. Externe Funktionen sind Funktionen, die in einer anderen Sprache implementiert wurden. Auch diese haben eine Regel, diese besteht dann lediglich aus dem externen Bezeichner. Die Regeln interner Funktionen bestehen aus einer Liste von Variablen und einem Ausdruck:

```
data Rule = Rule [VarIndex] Expr
          | External String
```

In der Übersetzungstabelle wird den externen Bezeichnern (die der Benutzer sieht) ein interner zugeordnet:

```
data Translation = Trans String String
```

Hierdurch werden Konflikte zwischen Bezeichnern verschiedener Module vermieden.

Ausdrücke werden durch den Datentyp `Expr` dargestellt:

```
data Expr = Var VarIndex
          | Lit Literal
          | Comb CombType String [Expr]
          | Apply Expr Expr
          | Constr [VarIndex] Expr
          | Or Expr Expr
          | Case CaseType Expr [CaseExpr]
          | GuardedExpr [VarIndex] Expr Expr
          | Choice Expr

type VarIndex = Int
data CombType = FuncCall | ConsCall | PartCall
data CaseType = Rigid | Flex

data CaseExpr = Pattern Literal [VarIndex] Expr
data Literal = Intc Int | Floatc Float |
               Charc Char | Ident String
```

Bsp. 5.2.1 *Die Bedeutung der verschiedenen Ausdrücke wird an der Übersetzung einiger Beispiele verdeutlicht:*

1. *Variablen werden durch Integer-Werte dargestellt:*

   ```
   x → Var 0
   ```

2. *Funktionsaufrufe und Datenterme werden in* `Comb`*-Ausdrücke überetzt:*

   ```
   f 1 → Comb FuncCall "f" [Lit (Intc 1)]
   [1] → Comb ConsCall ":" [Lit (Intc 1),Comb ConsCall "[]" []]
   (+) 1 → Comb PartCall "+" [Lit (Intc 1)]
   ```

3. *Sukzessive Applikation wird mit* `Apply` *ausgedrückt:*

   ```
   ((+) 1) 2 → Apply (Comb PartCall "+" [Lit (Intc 1)]) (Lit (Intc 2))
   ```

4. *Deklarationen logischer Variablen mit* `let` *werden mit dem Konstruktor* `Constr` *dargestellt:*

   ```
   let x free in success → Constr [0] (Comb FuncCall "success" [])
   ```

   Andere let-Deklarationen werden durch lambda lifting eliminiert

5. *Der Konstruktor* `GuardedExpr` *wird unter anderem für* `where`*-Deklaration benutzt:*

   ```
   ...| success = x where x free
                  ↓
   GuardedExpr [0] (Comb FuncCall "success" []) (Var 0)
   ```

   Boolesche Guards werden in if-Ausdrücke übersetzt

6. *Pattern matching wird in* `Case`- *und* `Or`-*Ausdrücke übersetzt:*

```
      and False _ = False
      and _ False = False
      and True True = True
                ↓
    Func "and" 2 (FuncType (TCons "Bool" [])
                           (FuncType (TCons "Bool" [])
                                     (TCons "Bool" [])))
         (Rule [1,2] (Or (Case Rigid (Var 1) [
                           Pattern (Lit (Ident "False"))
                                   []
                                   (Comb ConsCall "False" []),
                           Pattern (Lit (Ident "True"))
                                   []
                                   (Case Rigid (Var 2) [
                                     Pattern (Lit (Ident "True"))
                                             []
                                             (Comb ConsCall "True" []),
                                     Pattern (Lit (Ident "False"))
                                             []
                                             (Comb ConsCall "False" [])
                                   ])
                         ])
                         (Case Rigid (Var 2) [
                           Pattern (Lit (Ident "False"))
                                   []
                                   (Comb ConsCall "False" [])
                         ])))
```

Diese Definition der `and`-*Funktion hat in Curry eine andere Bedeutung als in Haskell: Sie ist aufgrund der überlappenden Regeln nichtdeterministisch, was auch durch den* `Or`-*Ausdruck deutlich wird.*

7. *commited choice wird wie folgt übersetzt:*

```
      choose eval choice
      choose x _ = x
      choose _ y = y
               ↓
  Func "choose" 2 (FuncType (TVar 0)
                            (FuncType (TVar 0) (TVar 0)))
        (Rule [1,2] (Choice
                       (Or (GuardedExpr []
                                        (Comb FuncCall "success" [])
                                        (Var 1))
                           (GuardedExpr []
                                        (Comb FuncCall "success" [])
                                        (Var 2)))))
```

Die rechten Seiten werden von `choice` *eingeschlossen und müßen eventuell noch in bewachte Ausdrücke umgewandelt werden.*

6. Eine Programmierumgebung für Curry

Obwohl deklarative Programmiersprachen das Programmieren leichter und sicherer machen, ist ihre Popularität nicht sehr groß. Dies liegt unter anderem daran, dass es nur sehr wenige Werkzeuge gibt, die den Entwickler bei seiner Arbeit unterstützen. Eine graphische Programmierumgebung für die deklarative Programmiersprache Curry soll diese Situation verbessern. Wegen der großen Ähnlichkeiten zwischen Curry und Haskell kann sie auch für die Erstellung von Haskell-Programmen benutzt werden.

6.1. Anforderungen

Es soll ein Werkzeug erstellt werden, welches den Progammierer bei der Erstellung von Curry-Programmen unterstützt. Die Behebung von Syntax- und Typfehlern ist relativ einfach, da der Compiler mehr oder weniger gute Hinweise auf die Fehlerquelle gibt. Logische Fehler sind dagegen schwerer zu beheben, hierfür gibt es jedoch erst sehr wenige Werkzeuge. Deshalb ist die Programmierumgebung für diese spätere Phase der Programmerstellung konzipiert.

Die Umgebung basiert auf zwei verschiedenen Konzepten: Zum Einen soll ein einfacher Tracer implementiert werden. Der Programmierer erhält hierdurch einen besseren Überblick über die Berechnungen und kann den Fehler so besser lokalisieren. Zum Anderen soll ein Rahmen für die Integration beliebiger Programmanalysen geschaffen werden. Es können auf einfache Art und Weise neue Analyse-Module integriert werden, die das Auffinden von Fehlern einer bestimmten Fehlerklasse ermöglichen. Diese Module müssen lediglich eine Funktion von einem bestimmten Typ implementieren, deren Ergenis entweder ein Ergebnisstring oder ein Graph ist. Ein Beispiel für eine Analyse ist zum Beispiel ein Modul, das unvollständie Funktionen, die nicht an allen Stellen definiert sind, auffindet. Falls ein Programm einfach mit der Meldung "no more solutions" abbricht, kann der Fehler somit sehr leicht und schnell lokalisiert werden. Schließlich sollen einige einfache Analyse-Module implementiert und in die Umgebung integriert werden.

Für die Implementierung soll die Sprache Curry verwendet werden.

6.2. Benutzerschnittstelle

Die Benutzerschnittstelle besteht aus sechs Teilen. Über die Menüleiste können die üblichen Editor-Funktionen aufgerufen werden: Laden und Speichern von Programmen, Kopieren, Ausschneiden, Einfügen und Suchen. Außerdem können Compiler und Debugger aufgerufen werden. Im Editorfenster wird der Quelltext des Programms angezeigt und

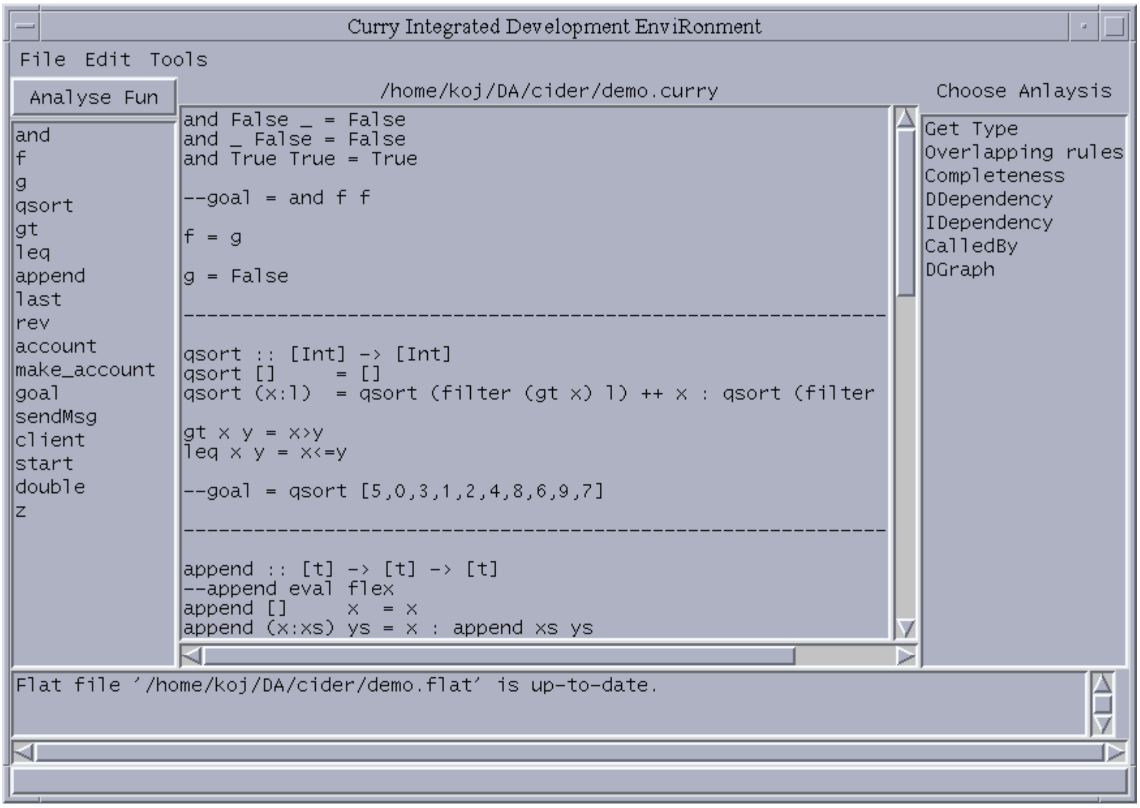

Abbildung 6.1.: Die Benutzerschnittstelle

kann editiert werden. Rechts davon werden alle verfügbaren Analysen in der Analyseliste angezeigt. Durch Anklicken kann die aktuelle Analyse bestimmt werden. Links vom Editorfenster werden alle deklarierten Funktionen in der Funktionsliste angezeigt. Die Funktion, welche analysiert werden soll, kann ebenfalls durch Anklicken ausgewählt werden. Sie wird mit der aktuellen Analyse analysiert, wenn der Analyseknopf oberhalb der Funktionsliste betätigt wird. Es können keine lokalen Funktionen analysiert werden. Da die Analysen auf FlatCurry basieren, könnten die Analyseergebnisse lokaler Funktionen wegen des lambda liftings sehr unverständlich sein. Im Informationsfenster unter dem Editorfenster werden die Ergebnisse der Analyse und die Meldungen des Compilers ausgegeben. In der Statuszeile werden gewisse Statusinformationen ausgegeben. Außerdem wird sie zur Eingabe eines Suchstrings benutzt. Abbildung 6.1 zeigt die Benutzerschnittstelle nach dem Öffnen der Datei `demo.curry`.

6.3. Implementierung

6.3.1. Überblick

Die Programmierumgebung besteht aus folgenden Komponenten:

- Die graphische Benutzerschnittstelle (GUI)
- Dem Zustands-Server (GUIServer)

- Dem Server zum Ausführen von IO-Aktionen (IOServer)
- Dem Konfigurations-Modul (Conf)
- Den Programmanalysen (Analysis)
- Dem Meta-Evaluator (Evaluator)
- Dem Debugger (Debugger)

Zusätzlich werden die Bibliotheken Tk (siehe Kapitel 4), Flat (siehe Kapitel 5) und DaVinci (siehe Kapitel 9) benutzt. Die Beziehungen zwischen den einzelnen Komponenten verdeutlicht Abbildung 6.2.

Das Modul GUI stellt die graphische Benutzerschnittstelle dar und reagiert auf die Aktionen des Benutzers. Es benutzt dazu die Tk-Bibliothek aus der PAKCS-Distribution. Der GUI-Server hält einen Zustand. Dieser enthält unter anderem das aktuelle FlatCurry-Programm und die Analyse-Resultate. Die Benutzerschnittstelle kommuniziert mit dem Server über Ports um Informationen auszutauschen. Falls beispielsweise der Benutzer eine neue Analyse auswählt, wird der Server hierüber informiert. Umgekehrt sendet der Server zum Beispiel den neuen Quelltext zur Benutzerschnittstelle, wenn ein neues Programm geöffnet wurde. Aus bestimmten Gründen kann der Server selbst keine IO-Aktionen ausführen. Deshalb wird ein IOServer in einem neuen Unix-Prozess gestartet, der diese Aktionen ausführt. Der GUI-Server kommuniziert mit diesem über externe Ports. Bei einem Aufruf des Debuggers oder zur Ausgabe von Graphen muß ein neues Fenster geöffnet werden. Dies kann ebenfalls nur der IO-Server tun. Im Conf-Modul gibt es zu jeder Analyse einen Eintrag. Dieser legt den Namen der Analyse, der in der Benutzerschnittstelle angezeigt werden soll, und die zugehörige Analyse-Funktion fest. Diese Funktion muß im entsprechenden Analyse-Modul definiert sein. Die Analysen müssen auf FlatCurry basieren und werden vom Server aufgerufen. Der Debugger benutzt einen Meta-Evaluator, der ebenfalls auf FlatCurry aufbaut.

6.3.2. Das GUI-Modul

Das Menü der Benutzerschnittstelle besteht aus drei Untermenüs: *File*, *Edit* und *Tools*.

Im *File*-Menü gibt es die Unterpunkte *New*, *Open*, *Save*, *Save As* und *Exit*. Durch Anwählen des Punktes *New* kann die Erstellung eines neuen Programms begonnen werden. Dazu wird der Inhalt des Editorfensters, des Informationsfensters und der Funktionsliste geleert. Der aktuelle Dateiname wird auf "" gesetzt. Der entsprechende Event-Handler sieht wie folgt aus:

```
newH gp = let editwin free
          in send (GetEditWindow editwin) guiServer &>
             send New guiServer &>
             tkCSetValue editwin "" gp &>
             tkCSetValue infowindow "" gp &>
             tkCConfig topfunwindow (TkList []) gp
```

Um eine eventuelle Erweiterung auf mehrere Editorfenster zu erleichtern, wird im Serverzustand das aktive Editorfenster gehalten. Deshalb muß dieses erst mit der Nach-

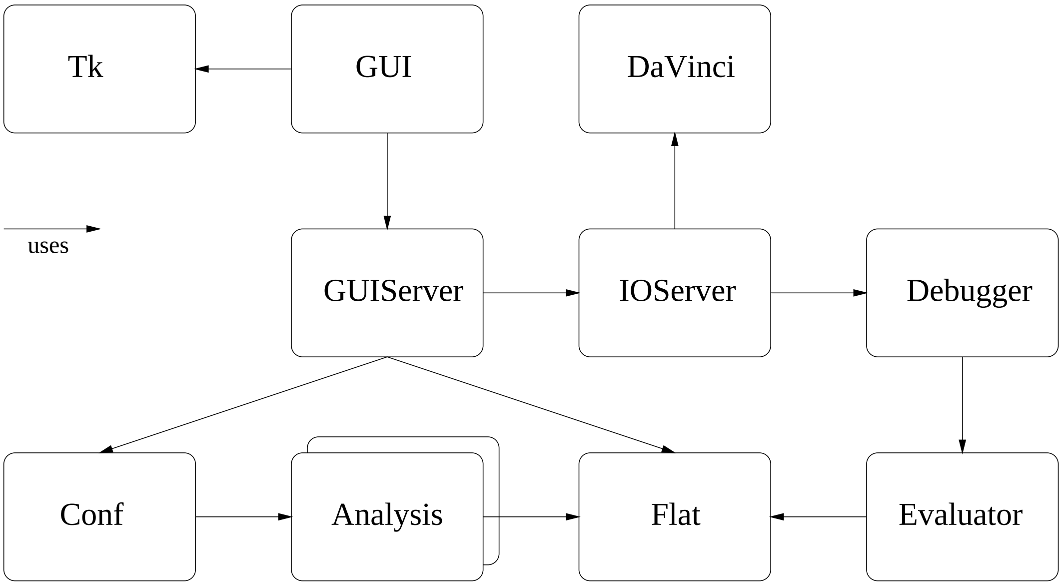

Abbildung 6.2.: Beziehungen zwischen den einzelnen Komponenten

richt (`GetEditWindow editwin`) abgefragt werden. Mit dem Server kann über den Port `guiServer` kommuniziert werden. Mit der Nachricht `New` wird der Dateiname der im aktuellen Editorfenster angezeigten Datei auf `""` gesetzt. Mit den Tk-Kommandos `tkCSetValue` und `tkCConfig` können die Werte der Widgets entsprechend verändert werden. Das Informationsfenster hat die Referenz `infowindow`, die Funktionsliste die Referenz `topfunwindow`.

Wenn der Menüpunkt *Open* angewählt wird, wird ein Dialog zur Dateiauswahl angezeigt. Hierfür gibt es das Tk-Kommando

```
tkCGetOpenFile :: String -> [(String,String)] ->
                  Port ppMsg -> Success.
```

Das erste Argument ist eine freie Variable, die nach der Dateiauswahl durch den Benutzer an den ausgewählten Dateinamen gebunden wird. Mit dem zweiten Argument können Dateifilter festgelegt werden. Ein solcher Filter besteht aus einer Bezeichnung (zum Beispiel `"literate curry files"`) und dem entsprechenden Suffix (zum Beispiel `".lcurry"`). Der Benutzer kann dann zwischen den verschiedenen Filtern wählen und bekommt jeweils nur die Dateien mit dem zugehörigen Suffix angezeigt. Steht der Dateiname der zu öffnenden Datei fest, kann die Datei durch das Versenden der Nachricht (`Open name contents parserout fnames`) an den Server geöffnet werden. Dadurch wird das Programm `name` nach FlatCurry übersetzt und eingelesen. Die Variable `contents` wird an den Quelltext des Programms, `parserout` an die Ausgabe des Parsers und `fnames` an die Liste der Namen der im Programm definierten Funktionen gebunden. Der Wert von `contents` wird in das Editorfenster geschrieben, die Ausgabe des Parsers in das Informationsfenster und die Funkionsnamen in die Funktionsliste.

Wenn das Programm gespeichert werden soll, wird lediglich die Nachricht (`Save contents`) an den Server geschickt. Falls zusätzlich noch der Dateiname festgelegt werden soll (Menüpunkt *Save As*), muß vorher noch ein ähnlicher Dialog wie beim Öffnen gestartet werden. In diesem Falle wird die Datei mit der Nachricht (`SaveAs filename contents`) gespeichert.

Im Edit-Menü gibt es die Unterpunkte *Copy* (Kopieren), *Cut* (Ausschneiden), *Paste* (Einfügen) und *Search* (Suchen). Die Funktionen Kopieren, Ausschneiden und Einfügen werden mit den entsprechenden Kommandos aus der Tk-Library implementiert. Die Funktionen Kopieren und Ausschneiden können mit den Tk-Kommandos `tkCCopy` und `tkCPaste` implementiert werden. Durch den Constraint `(tkCCopy editwin cs gp)` beziehungsweise `(tkCCut editwin cs gp)` wird die Variable `cs` an den gerade im Editorfenster selektierten String gebunden. Der selektierte String wird dann durch Versenden der Nachricht `(SetCopyString cs)` an den Server gespeichert. Wenn der Menüpunkt *Paste* ausgewählt wird kann er durch Versenden von `(GetCopyString cs)` wieder gelesen werden und durch den Aufruf `(tkCPaste editwin cs gp)` an der aktuellen Position im Editorfenster eingefügt werden. Wird der Unterpunkt *Search* ausgewählt, wird die Statuszeile aktiviert und rot gefärbt. Außerdem muß der Statuszeilenmodus auf `"search"` gesetzt werden. Dies erledigt der folgende Event-Handler:

```
searchH gp = send (SetStatusMode "search") stateServer &>
             tkCConfig statuswindow (TkActive True) gp &>
             tkCConfig statuswindow (TkBackground "red") gp
```

Da die Statuszeile später eventuell für verschiedene Zwecke benutzt werden soll, und man Event-Handler zur Laufzeit nicht austauschen kann, wird ein Statuszeilenmodus (ein einfacher String) im Server gehalten. Somit weiss der Event-Handler der Statuszeile, wie er reagieren muß:

```
statusH gp = let mode free
             in send (GetStatusMode mode) stateServer &>
                if mode=="search" then search gp else success
```

Dieser Handler wird immer dann aufgerufen, wenn in der Statuszeile ein neuer Buchstabe hinzugefügt wurde. Im Moment gibt es nur den Modus `"search"`. Die Funktion `search` sucht den Suchstring aus der Statuszeile im Editorfenster. Falls er gefunden wird, wird er rot markiert. Endet der Suchstring mit `\n` (*newline*), wird die Suche beendet: Der Statuszeilenmodus wird auf `""` gesetzt und die Statuszeile wird wieder deaktiviert.

Im Tools-Menü gibt es die beiden Unterpunkte *Compile* und *Debug*. Durch Anklicken von *Compile* wird das Programm gespeichert und neu übersetzt. Dies wird durch Versenden der Nachtichten `Save` und `Open` erledigt. Danach muß noch die Funktionsliste aktualisiert werden. Der Debugger kann durch das Verschicken der Nachricht `StartDebugger` gestartet werden.

Wenn in der Analyseliste eine neue Analyse `ana` ausgewählt wird, muß dies dem Server durch Versenden der Nachricht `(SetAna ana)` mitgeteilt werden. Wird eine neue Funktion ausgewählt, soll im Editorfenster zu der zugehörigen Deklaration gesprungen werden. Da die FlatCurry-Repräsentation eines Programms keine Information über Zeilen- oder Spaltennummern enthält, muß die Funktionsdeklaration im Editorfenster erst noch gesucht werden. Dies geschieht mit dem Funktionsaufruf

```
tkCSearchI win pattern (r,c) index gp
```

Es wird im Editorfenster von Zeile `r` und Spalte `c` an nach dem String `pattern` gesucht. Die Variable `index` wird an den Ergebnisstring gebunden, der noch interpretiert werden muß. Handelt es sich um Literate-Curry, muß nach dem String `("> "++fname++" ")` gesucht werden, ansonsten nach `(fname++" ")`. Es muß wiederholt gesucht werden, bis der Funktionsname in Spalte 0 gefunden wird. Leider können so keine Deklarationen von Infix-Operatoren gefunden werden. Hierzu müsste zusätzlich die Möglichkeit von Tk, nach regulären Ausdrücken zu suchen, ausgenutzt werden. Wurde die Deklaration gefunden, wird mit dem Aufruf

```
tkCSeeI editwin index gp
```

im Editorfenster zu dem entsprechenden Index `index` gesprungen.

Wenn der Analyse-Knopf betätigt wird, kann das Analyseresultat durch Versenden der Nachricht `(GetAnaRes fun result color)`. Die Variable `result` wird an den Ergebnis-String gebunden, welcher dann in das Informationsfenster geschrieben wird. `color` wird an eine dem Ergebnis entsprechende Farbe gebunden, mit der der Funktionsname in der Funktionsliste gefärbt wird.

6.3.3. Das GUI-Server-Modul

Der GUI-Server empfängt die Nachrichten von der Benutzerschnittstelle, beantwortet sie durch Binden der freien Variablen und aktualisiert seinen Zustand. Der Zustand besteht aus den folgenden Komponenten:

- Das aktuelle Editorfenster
- Der Statuszeilenmodus
- Die FlatCurry-Repräsentation des aktuellen Programms
- Der Dateiname des aktuellen Programms
- Eine Liste mit den Ergebnissen der aktuellen Programmanalyse
- Der Port des IO-Servers

Die Ergebnisliste ist eine Liste von Tupeln, die jedem Funktionsnamen ein Ergebnis zuweist.

Der Server kann selbst keine IO-Aktionen ausführen, da sein Ergebnistyp `Success` ist. Da dies jedoch notwendig ist (beispielsweise zum Öffnen von Dateien), wird ein IO-Server in einem neuen Unix-Prozess gestartet. Der GUI-Server kann somit das Ausführen von IO-Aktionen durch den IO-Server veranlassen, indem er entsprechende Nachrichten über einen externen Port verschickt.

Wenn eine neue Analyse ausgewählt wurde, erhält der Server die Nachricht `(SetAna ana)`. In diesem Falle muß die Liste mit den Analyseergebnissen aktualisiert werden. Dazu muß im Conf-Modul die zu `ana` gehörende Analysefunktion `analyse` gesucht werden. Die neue Liste ergibt sich dann aus der Applikation dieser Funktion auf das aktuelle FlatCurry-Programm:

```
data AnaRes = Message String String | Graph DvGraph String
```

```
analyse :: Prog -> [(String,AnaRes)]
```

Eine solche Analysefunktion hat als einziges Argument ein FlatCurry-Programm. Ihr Ergebnis ist eine Liste von Tupeln, welche jedem Funktionsnamen ein Ergebnis zuordnet. Dieses Ergebnis besteht entweder aus einem Ergebnisstring, der im Informationsfenster angezeigt werden soll, oder einem Graph, der mit der DaVinci-Bibliothek angezeigt werden soll. Zusätzlich besteht es noch aus einer Farbe, mit der der Eintrag der Funktion in der Funktionsliste gefärbt werden soll. Erhält der Server nun die Nachricht `(GetAnaRes name res color)` verhält er sich wie folgt: Er sucht in der Ergebnisliste das Ergebnis für die Funktion `name`. Handelt es sich bei diesem um einen String, der im Informationsfenster angezeigt werden soll, wird die Variable `res` entsprechend gebunden, ansonsten wird sie an den leeren String gebunden. Handelt es sich bei dem Ergebnis um einen Graphen, muß dieser angezeigt werden. Hierzu muß eine IO-Aktion ausgeführt werden, dazu wird einfach die Nachricht `(DisplayGraph g)` zum IO-Server gesendet. Die Variable `color` wird in beiden Fällen an die entsprechende Farbe gebunden.

Es scheint zunächst so, als würden alle Funktionen sofort bei Empfang der Nachricht `(SetAna ana)` analysiert. Dieses Vehalten ist jedoch nicht wünschenswert, da die Analyse einer einzelnen Funktion unter Umständen sehr aufwendig sein kann. Es ist jedoch auch nicht der Fall: Aufgrund der bedarfsgesteuerten Auswertung werden die Funktionen erst analysiert, wenn das Ergebnis angezeigt werden soll. Zudem wird jede Funktion nur einmal analysiert. Soll das Ergebnis ein zweites Mal angezeigt werden, liegt das Ergebnis sofort vor, es findet keine erneute Berechnung statt. Um dieses Verhalten zu erreichen, müsste in imperativen Programmiersprachen einiger Aufwand betrieben werden. Die Auswertungsstrategie von Curry ist an dieser Stelle also sehr nützlich.

6.3.4. Das IO-Server-Modul

Der IO-Server hat den Ergebnistyp `IO ()`:

```
serveIO :: [IOServerMsg] -> IO ()
```

Aus diesem Grunde muß er in einem eigenen Unix-Prozess ablaufen. Die Nachrichten, die er vom GUI-Server empfängt, entsprechen im Wesentlichen denen, die der GUI-Server empfängt und das Ausführen von IO-Aktionen erfordern. Beim Empfangen einer solchen Nachricht werden die entsprechenden IO-Aktionen ausgeführt und die Rückgabevariablen werden an die Ergebnisse gebunden. Das Binden von Variablen wird mit der Funktion `unifyIO` in die Sequenz von IO-Aktionen eingebettet:

```
unifyIO :: a -> a -> IO ()
unfiyIO x y | x=:=y = done
```

Wenn die Nachricht `(StartDebuggerIO prog expr)` empfangen wird, wird der Debuger durch den Aufruf `(startDebugger prog expr)` gestartet. Da diese IO-Aktion erst beendet ist, wenn der Debugger geschlossen wird, ist der IO-Server während dieser Zeit blockiert. Es können also nicht zwei Debugger gleichzeitig gestartet werden.

6.3.5. Das Conf-Modul

Über das Conf-Modul kann die Programmierumgebung konfiguriert werden. In der Liste

```
anaList :: [(String,Prog->[(String,AnaRes)])]
```

wird zu jeder Analyse, die in die Umgebung integriert werden soll, der Name, der in der Analyseliste angezeigt werden soll, und die Analysefunktion definiert. Der GUI-Server liest diese Informationen aus, um die Analyseergebnisse zu berechnen. Außerdem wird in diesem Modul der Bezeichner `ioServerPort` festgelegt, über den der Port des IO-Servers identifiziert und geöffnet werden kann.

6.4. Programmanalysen

Die folgenden Programmanalysen wurden bereits implementiert und in die Programmierumgebung integriert:

- Typ-Analyse
- Überlappungs-Analyse
- Vollständigkeits-Analyse
- Aufruf-Analyse
- Abhängigkeits-Analyse

Die ersten drei Analysen erlauben die Lokalisierung bestimmter Fehler, mit den letzten kann die Fortpflanzung dieser analysiert werden. Im Folgenden werden die einzelnen Analysen etwas genauer betrachtet. Damit der Typ `AnaRes` zur Verfügung steht, muß jedes Analyse-Modul neben dem `Flat`-Modul auch noch das nicht weiter erwähnte `Types`-Modul importieren.

6.4.1. Typ-Analyse

In Curry muß es nicht zu jeder Funktion eine Typdeklaration geben. Oft ist es praktisch, die Typdeklaration einfach weg zu lassen. Bei der Fehlersuche ist es dann jedoch manchmal wichtig, den Typ einer bestimmten Funktion zu kennen. Mit der Typ-Analyse kann sich deshalb der Programmierer den Typ jeder Funktion anzeigen lassen, auch wenn dieser nicht deklariert wurde.

Die Implementierung dieser Analyse ist denkbar einfach, da die entsprechenden Informationen in der FlatCurry-Repräsentation eines Programms bereits vorhanden sind und lediglich extrahiert werden müssen. Ein FlatCurry-Programm besteht unter anderem aus einer Liste von Funktionsdeklarationen vom Typ `FuncDecl`:

```
data FuncDecl = Func String Int TypeExpr Rule
```

Eine Funktion `analyseType`, welche zu einer Funktion ein entspechendes Ergebnistupel liefert, kann also einfach wie folgt implementiert werden:

```
analyseType :: [Translation] -> FuncDecl -> (String,AnaRes)
analyseType trans (Func name _ typ _) =
  (revTrans trans name,Message (ppTypes typ) "black")
```

```
    where revTrans [] _ = ""
          revTrans ((Trans n1 n2):ts) n | n==n2 = n1
                                        | otherwise = revTrans ts n
```

Der Typausdruck `typ` wird mit der Funktion `ppTypes` in einen String umgewandelt, der im Informationsfenster der Programmierumgebung ausgegeben werden soll. Da es wenig Sinn macht, einem Typen eine bestimmte Farbe zuzuordnen, wird jeder Funktion die Farbe schwarz zugeordnet. Da in FlatCurry interne Funktionsnamen verwendet werden, müssen diese noch mit der Funktion `revTrans` und der Übersetzungstabelle `trans` zurückübersetzt werden. Die Funktion `analyse`, mit der der GUI-Server die Ergebnisliste bestimmt, kann nun einfach wie folgt realisiert werden:

```
analyse :: Prog -> [(String,AnaRes)]
analyse (Prog _ _ _ _ funs trans) = map (analyseType trans) funs
```

Die Funktion `map` wendet eine beliebige Funktion auf alle Elemente einer Liste an.

6.4.2. Überlappungs-Analyse

Überlappende linke Regelseiten können die Ursache für ineffiziente Programme sein. Sie können eine große Zahl oder-paralleler Berechnungen verursachen. Ein Beispiel hierfür ist die folgende Funktionsdeklaration:

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

Überlappende Regeln sind also nicht immer erwünscht, zudem werden Überlappungen vom Programmierer manchmal einfach übersehen. Insbesondere die Programmierer, die von Haskell auf Curry umsteigen, haben höschst wahrscheinlich dieses Problem, da in Haskell ja immer nur die erste mögliche Regel angewendet wird.

In einem FlatCurry-Programm lassen sich Funktionsdeklarationen mit überlappenden Regeln recht einfach auffinden. Sie können am Auftreten von `Or`-Ausdrücken in der rechten Seite erkannt werden. Eine Funktion `analyseOverlapping` kann somit mit der Funktion `orInExpr` definiert werden:

```
analyseOverlapping trans (Func name _ _ (Rule _ e)) | orInExpr e =
  (revTrans trans name,Message "Overlapping" "red")
                                                    | otherwise =
  (revTrans trans name,Message "Not Overlapping" "blue")

analyseOverlapping trans (Func _ _ _ (External _)) =
  (revTrans trans name,Message "Not Overlapping" "blue")
```

Falls die rechte Seite `e` also einen `Or`-Ausdruck enthält, soll die Meldung `"Overlapping"` im Informationsfenster ausgegeben werden und der Eintrag der Funktion rot gefärbt werden. Ansonsten wird `"Not Overlapping"` ausgegeben und der Eintrag blau gefärbt. Da über externe Funktionen keine weitere Information vorhanden ist, werden diese als nicht überlappend vorausgesetzt. Die Definition der Funktion `orInExpr` ist nun wie folgt:

```
orInExpr :: Expr -> Bool
orInExpr (Var _) = False
orInExpr (Lit _) = False
orInExpr (Comb _ _ es) = foldr (||) False (map orInExpr es)
orInExpr (Apply e1 e2) = orInExpr e1 || orInExpr e2
orInExpr (Constr _ e) = orInExpr e
orInExpr (Or _ _) = True
orInExpr (Case _ e cs) = orInExpr e ||
                         foldr (||) False (map orInCase cs)
  where orInCase (Pattern _ _ e) = orInExpr e
orInExpr (GuardedExpr _ e1 e2) = orInExpr e1 || orInExpr e2
orInExpr (Choice _) = False
```

Die disjunktive Verknüpfung zweier boolescher Werte wird durch den Infix-Operator `(||)` durchgeführt. Atomare Ausdrücke enthalten kein `Or`, `Or`-Ausdrücke dagegen enthalten ganz sicher einen `Or`-Ausdruck. Alle anderen Ausdrücke enthalten genau dann einen `Or`-Ausdruck, wenn dies bei einem ihrer Teil-Ausdrücke der Fall ist. Da dies jedoch bei `Choice`- Ausdrücken nicht relevant ist, weil in diesem Falle nichtdeterministische Berechnungen gekapselt werden, wird bei `Choice`-Ausdrücken stets `False` zurückgeliefert.

Die erforderliche Funktion `analyse` kann nun einfach wie bei der Typ-Analyse mittels `map` definiert werden.

Diese Analyse ist bereits Teil der PAKCS-Distribution, in der sie jedoch nur gleichzeitig auf alle Funktionen angewendet werden kann. Dieses Beispiel zeigt also, dass bestehende Analysen einfach in die Programmierumgebung integriert werden können um von den Vorteilen dieser zu profitieren.

6.4.3. Vollstandigkeits-Analyse

Eine Funktion ist vollständig, falls es für jede mögliche Kombination von Argumenten mindestens eine passende Regel gibt, falls also das pattern matching niemals fehl schlägt. Die Definition unvollständiger Funktionen kann durchaus sinnvoll sein, die Funktion `head`, die das erste Element einer Liste bestimmt, kann beispielsweise nicht für die leere Liste definiert werden. Andererseits passiert es oft, dass bei der Definition einer Funktion eine Regel vergessen wird. Dies hat dann zur Folge, dass die Ausführung des Programms einfach mit der Meldung "no more solutions" beendet wird. Das Auffinden des Fehlers, also der unvollständigen Funktion, ist in diesem Falle sehr aufwendig und soll durch die Vollständigkeits-Analyse unterstützt werden. Diese bestimmt zu jeder Funktion, ob sie vollständig ist oder nicht.

Da die Implementierung einer korrekten Vollständigkeits-Analyse relativ aufwendig ist, wird hier nur eine Approximation vorgestellt. Die Funktion `analyseCompleteness` analysiert eine Funktion auf Vollständigkeit:

```
analyseCompleteness types trans (Func name _ typ (Rule vs e))
  | isComplete types (initialTypeAss vs typ) e =
    (revTrans trans name,Message "complete" "blue")
  | otherwise =
    (revTrans trans name,Message "might be incomplete" "red")

analyseCompleteness _ trans (Func name _ _ (External _)) =
  (revTrans trans name,Message "complete" "blue")
```

Da über externe Funktion keine weiteren Informationen zur Verfügung stehen, sollen diese stets vollständig sein. Das Ergebnis für die anderen Funktionen kann mit der Funktion `isComplete` bestimmt werden. Da pattern matching mit `Case-` und `Or`-Ausdrücke dargestellt wird, kann die Vollständigkeit einer Funktionsdeklaration an ihrer rechten Seite abgelesen werden. Es handelt sich bei `isComplete` jedoch nur um eine Approximation: Eine unvollständige Funktion wird stets als solche erkannt, das heißt `isComplete` liefert `False` zurück. Bei vollständigen Funktionen kann `isComplete` jedoch unter Umständen auch `False` liefern. Aus diesem Grund wird in diesem Falle der Ergebnisstring `"might be incomplete"` zurückgegeben.

Die Funktion `isComplete` hat drei Argumente: `types` ist die Liste aller Datentypdeklarationen des Programms. `typeAss` ordnet jeder in einem `Case`-Ausdruck vorkommenden Variablen ihren Typ zu. Das dritte Argument ist der zu analysierende Ausdruck. Es sind lediglich zwei Fälle interessant:

```
isComplete types typeAss (Case _ v ces) =
  test (getConstructors (lookupType v typeAss) types) ces
  where test [] _ = True
        test (c:cs) [] = False
        test ((Cons c ar ts):cs) ((Pattern (Ident i) vs e):ps) =
          (test (removeConstructor i ((Cons c ar ts):cs)) ps) &&
          (isComplete types (makeTypeAss vs ts typeAss) e)
        test (_:_) ((Pattern (Intc _) _ _):_) = False
        test (_:_) ((Pattern (Floatc _) _ _):_) = False
        test (_:_) ((Pattern (Charc _) _ _):_) = False
```

Der `Case`-Ausdruck (`Case ct v ces`) ist dann vollständig, wenn es zu jedem Datenkonstruktor, der zum Typen von `v` gehört, ein Pattern gibt und seine Teilausdrücke ebenfalls vollständig sind. Dies wird durch die ersten drei Regeln der lokalen Funktion `test` geprüft. Mit den Hilfsfunktionen `lookupType` und `getConstructors` werden die Datenkonstruktoren zu der Variablen `v` bestimmt. Um die Vollständigkeit der Teilausdrücke zu bestimmen, muß die Typzuweisung mir der Funktion `makeTypeAss` erweitert werden. Eine Funktion, bei der Datenkonstruktoren der vordefinierten Typen `Int`, `Float` oder `Char` in der linken Seite vorkommen, ist mit aller höchster Wahrscheinlichkeit unvollständig. Deshalb gibt `test` der Einfachheit halber `False` zurück, falls in einem Pattern ein solcher Datenkonstruktor vorkommt. Dies kommt in den letzten drei Regeln zum Ausdruck.

`Or`-Ausdrücke werden wie folgt behandelt:

```
isComplete types typeAss (Or e1 e2) =
  (isComplete types typeAss e1) || (isComplete types typeAss e2)
```

Ein `Or`-Ausdruck ist vollständig, falls mindestens einer seiner Teilaudrücke vollständig ist.

Alle anderen Ausdrücke sind vollständig, da ihre Teilausdrücke weder `Case-` noch `Or`-Ausdrücke sein können. Deshalb wird in allen anderen Fällen einfach `True` zurückgegeben.

Das diese Implementierung lediglich eine Approximation ist, liegt nicht an der Behandlung der `Case`-Ausdrücke. Die einfachere Behandlung von Patterns mit Datenkonstruktoren eines vordefinierten Typs stellt lediglich eine sehr geringe Einschränkung dar. Das Problem liegt in der Behandlung der `Or`-Ausdrücke. Das Ergebnis der Analyse der Definition des parallelen `and` aus Kapitel 5 ist zum Beispiel falsch: Obwohl es zu jeder möglichen Kombination von Argumenten eine Regel gibt, liefert diese Implementierung das Ergebnis `"might be incomplete"`. Für den praktischen Einsatz ist dies jedoch nicht so schlimm: Es werden trotzdem alle Funktionsdefinitionen gefunden, die möglicherwise fehlerhaft sind. Der Programmierer muß lediglich eventuell ein paar Definitionen mehr auf Fehler überprüfen.

Abschließend soll noch kurz skizziert werden, wie eine korrekte Vollständigkeits-Analyse implementiert werden kann. In [AEH97] wurde gezeigt, wie zu jedem Ausdruck, der `Or`-Ausdrücke enthält, eine äquivalente Menge von Ausdrücken ohne `Or`-Ausdrücke bestimmt werden kann, die zusammen die gleichen Fälle abdecken. Auf der Basis dieses Verfahrens kann nun die Vollständigkeits-Analyse relativ einfach implementiert werden: Zu einem `Or`-Ausdruck wird zunächst die äquivalente Ausdrucks-Menge bestimmt. Dann bestimmt man zu einem beliebigen Ausdruck dieser Menge alle nicht abgedeckten Patterns. Schließlich muß für jedes nicht abgedeckte Pattern überprüft werden, ob es einen anderen Ausdruck in der äquivalenten Menge gibt, der dieses abdeckt. Falls es einen solchen Ausdruck für jedes nicht abgedeckte Pattern gibt, ist der `Or`-Ausdruck vollständig, ansonsten unvollständig. Ein `Case`-Ausdruck ist genau dann vollständig, wenn die Menge der Patterns, die nicht von ihm abgedeckt werden, leer ist.

6.4.4. Aufruf-Analyse

Um die Auswirkungen eines Fehlers besser zu verstehen, kann es nützlich sein, alle Funktion zu kennen, die die fehlerhafte Funktion aufrufen. Hat beispielsweise eine Funktion unbeabsichtigte überlappende Regeln, so sind auch alle Funktionen, die diese aufrufen, nichtdeterministisch. Die Aufruf-Analyse bestimmt deshalb zu jeder Funktion die Funktionen, die sie aufrufen.

Die Funktion `analyseCalledBy` wird mit Hilfe der Funktion `getCalls` definiert:

```
analyseCalledBy funs trans (Func name _ _ _) =
  (revTrans trans name,
   Message (ppCalls (concatMap (getCalls name) funs)) "black")
```

In jeder Funktionsdeklaration muß nach einem Aufruf der Funktion `name` gesucht werden. Die Funktion `(getCalls name)` tut dies für eine Deklaration. Mit `concatMap` wird diese Funktion zunächst auf alle Funktionsdeklarationen des Programms appliziert. Es ergibt sich eine Liste von Listen von Funktionsnamen, die dann in einer flachen Liste von Funktionsnamen umgewandelt wird. Diese wird dann mit der Funktion `ppCalls` in den Ausgabestring umgewandelt. Die Funktion `getCalls` baut auf `isCalled` auf:

```
getCalls name1 (Func name2 _ _ (External _)) = []
getCalls name1 (Func name2 _ _ (Rule _ e)) | isCalled name1 e = [name2]
                                           | otherwise = []
```

Externe Funktionen können keine anderen Funktionen aufrufen. Falls der Ausdruck `e` einen Aufruf von `name1` enthält, wird eine Liste zurückgegeben, die lediglich aus `name2` besteht, ansonsten die leere Liste. Die Funktion `isCalled` gibt `True` zurück, falls die Funktion `name` im Ausdruck `e` gefunden wird, ansonsten `False`:

```
isCalled _ (Var _) = False
isCalled _ (Lit _) = False
isCalled name (Comb _ c es) | c==name = True
                            | otherwise =
  foldr (||) False (map (isCalled name) es)
isCalled name (Apply e1 e2) =
  (isCalled name e1) || (isCalled name e2)
isCalled name (Constr _ e) = isCalled name e
isCalled name (Or e1 e2) =
  (isCalled name e1) || (isCalled name e2)
isCalled name (Case _ _ ces) =
  foldr (||) False (map isCalledCase ces)
  where isCalledCase (Pattern _ _ e) = isCalled name e
isCalled name (GuardedExpr _ e1 e2) =
  (isCalled name e1) || (isCalled name e2)
isCalled name (Choice e) = isCalled name e
```

Die Funktion `analyse` kann nun wieder einfach mittels `map` definiert werden.

6.4.5. Abhängigkeits-Analyse

Die Abhängigkeits-Analyse ist die zur Aufruf-Analyse inverse Analyse. Sie bestimmt alle Funktionen, die für die Definition einer bestimmten Funktion notwendig sind. Es wird zwischen direkten und indirekten Abhängigkeiten unterschieden: Eine Funktion `f` hängt direkt von einer anderen Funktion `g` ab, falls `g` in der Deklaration für `f` vorkommt. Auf Basis der direkten Abhängigkeiten kann einem Programm ein gerichteter Abhängigkeitsgraph zugeordnet werden. Dessen Knoten bestehen aus den deklarierten Funktionen, und es gibt eine Kante von Knoten `f` zu Knoten `g`, falls `f` direkt von `g` abhängig ist. Eine Funktion `h` ist nun indirekt von `f` abhängig, falls es im Abhängigkeitsgraphen einen Pfad von `f` nach `h` gibt.

Die Funktion `analyseDDependency` bestimmt die direkten Abhängigkeiten einer Funktion:

```
analyseDDependency _ (Func name (External _)) =
  (revTrans trans name,Message "" "black")

analyseDDependency trans (Func name _ _ (Rule e)) =
  (revTrans trans name,Message (ppDeps (getDeps e)) "black")
```

Die Funktion `ppDeps` berechnet aus der Liste der Funktionsnamen den Ausgabestring, `getDeps` sammelt alle Funktionsaufrufe, die in einem Ausdruck vorkommen, auf:

```
getDeps (Var _) = []
getDeps (Lit _) = []
getDeps (Comb ConsCall _ es) = nub (concatMap getDeps es)
getDeps (Comb PartCall c es) = nub (c:(concatMap getDeps es))
```

```
getDeps (Comb FuncCall c es) = nub (c:(concatMap getDeps es)
getDeps (Apply e1 e2) = nub ((getDeps e1)++(getDeps e2))
getDeps (Constr _ e) = getDeps e
getDeps (Or e1 e2) = nub ((getDeps e1)++(getDeps e2))
getDeps (Case _ e ces) = nub (concatMap getDepsCase ces)
  where getDepsCase (Pattern _ _ e) = getDeps e
getDeps (GuardedExpr _ e1 e2) = nub ((getDeps e1)++(getDeps e2))
getDeps (Choice e) = getDeps e
```

Dabei entfernt die Funktion `nub` alle doppelten Vorkommnisse eines Elementes in einer Liste.

Indirekte Abhängigkeiten können ebenfalls mit der Funktion `getDeps` über eine Fixpunkt-Iteration berechnet werden:

```
fix funs olddeps newdeps | olddeps==newdeps = newdeps
                         | otherwise =
  fix funs newdeps
      (nub ((concat (map (\fname->getDeps (lookupExpr fname funs))
       newdeps))++newdeps))
```

Die Funktion `fix` hat eine Liste von alten (`olddeps`) und neuen (`newdeps`) Abhängigkeiten als Argumente. In jedem Schritt wird aus `newdeps` eine weitere Liste von Abhängigkeiten berechnet. Dazu wird `getDeps` mit `map` auf die rechten Seiten aller Funktionen in `newdeps` appliziert. Die rechten Seiten der Funktionen werden dabei mit `lookupExpr` bestimmt. Es ergibt sich eine Liste von Listen von Abhängigkeiten, die durch `concat` geglättet wird. Damit die Abhängigkeiten in `newdeps` nicht verloren gehen, werden sie an diese neue Liste angehängt. Schließlich werden noch die doppelten Vorkommnisse mit `nub` entfernt und `fix` ruft sich selbst rekursiv mit der neuen Abhängigkeitslsite auf. Die ehemals neue Abhängigkeitsliste wird dabei zur alten Abhängigkeitsliste. Falls sich keine neuen Abhängigkeiten ergeben haben (also falls `(olddeps==newdeps)`), bricht die Iteration ab und `newdeps` ist das gewünschte Ergebnis. Die entsprechende Funktion `analyseIDependency` wird nun einfach wie folgt implementiert:

```
analyseIDependency funs trans (Func name _ _ _) =
  (revTrans trans name,
   Message (ppDeps (fix funs [] (getDeps (lookupExpr name funs))))
          "black")
```

Um `fix` aufrufen zu können, müssen vorher noch die direkten Abängigkeiten von `name` berechnet werden.

Die indirekten Abhängigkeiten einer Funktion können auch leicht als Graph dargestellt werden. Dies leistet die Funktion

```
buildDGraph :: [FuncDecl] -> [Translation] ->
               FuncDecl -> (String,AnaRes),
```

die in einer ähnlichen Art und Weise wie `analyseIDependecy` definiert werden kann. Der Graph für die Funktion `quicksort` ist in Abbildung 6.3 dargestellt.

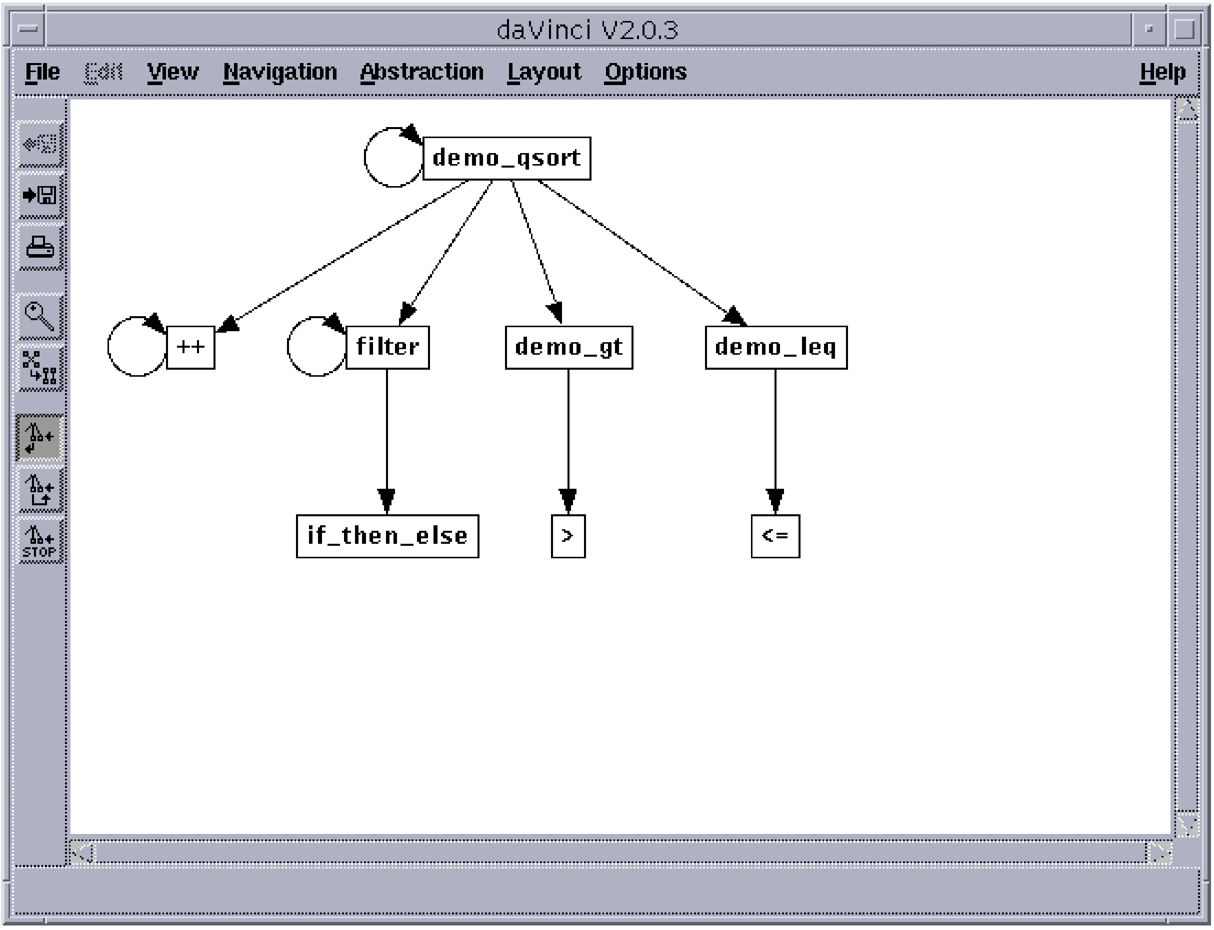

Abbildung 6.3.: Der Abhängigkeitsgraph für die Funktion `quicksort`

Diese kann wie folgt definiert werden:

```
qsort :: [Int] -> [Int]
qsort []    = []
qsort (x:l) = qsort (filter (gt x) l) ++ x : qsort (filter (leq x) l)

gt x y = x>y
leq x y = x<=y
```

7. Ein Meta-Evaluator

Ein Evaluator oder Interpretierer ist ein Programm, welches einen Ausdruck relativ zu einem Programm auswertet. Ein Meta-Evaluator ist ein Evaluator, der in der gleichen Programmiersprache wie die Ausdrücke, die er auswertet, implementiert ist.

Die Implementierung eines solchen Evaluators ist unter anderem aus den beiden folgenden Gründen sinnvoll: Erstens ermöglicht sie Erweiterungen oder Modifikationen der implementierten Sprache. Zweitens kann ein Evaluator als Grundlage für die Implementierung diverse Werkzeuge genommen werden. Ein Debugger oder Tracer kann entweder über eine Programmtransformation oder mit einem Meta-Evaluator implementiert werden. Dabei ist die zweite Lösung etwas aufwendiger und ineffizienter, sie bietet jedoch auch mehr Möglichkeiten und hat eine klarere Struktur. Partielle Evaluatoren können die Laufzeiten von Programmen verbessern.

Im Folgenden werden verschiedene Ansätze zur Implementierung eines Meta-Evaluators für Curry vorgeschlagen und verglichen.

7.1. Implementierung mit Redex-Listen

Der erste Ansatz zur Implementierung eines Evaluators basiert auf einfacher Termersetzung. Ein Reduktions- Schritt besteht aus zwei Teilen: Zunächst wird der Redex bestimmt, dann wird dieser im gesamten Ausdruck durch den entsprechenden reduzierten Ausdruck ersetzt.

Es ist wichtig, dass der Redex wirklich im gesamten Ausdruck ersetzt wird, es muß also in jedem Schritt der gesamte Ausdruck nach diesem durchsucht werden. Teilausdrücke, die durch Instantiierung der selben Variable entstanden sind, müßen gleichzeitig ausgewertet werden. Falls dies nicht geschieht, kann die Auswertung unter Umständen falsch sein. Dies wird an folgendem Beispiel verdeutlicht:

Bsp. 7.1.1 *Es wird folgendes Curry-Programm betrachtet:*

```
z = 1
z = 2

double x = x + x
```

Die Variable `x` *steht auf der rechten Seite der Funktion* `double` *beide male für den gleichen Wert. Deshalb kann der Ausdruck* `(double z)` *zu* `2` *oder* `4` *reduziert werden. Falls nun die Ersetzung nicht im gesamten Ausdruck gemacht würde, könnte das erste* `z` *zu* `1` *und das zweite* `z` *zu* `2` *reduziert werden. Es würde sich zusätzlich der Wert* `3` *ergeben.*

Der Ausdruck

```
z + z
```

kann dagegen zu allen drei Werten reduziert werden.

Damit der letzte Ausdruck aus dem vorhergehenden Beispiel korrekt reduziert wird, müssen Funktionsnamen in bestimmten Fällen geeignet umbenannt werden (siehe unten).

7.1.1. Der Evaluator-Zustand

Während der Auswertung eines Ausdrucks muß stets ein Zustand mitgeführt werden. Dieser dient hauptsächlich zur Vermeidung von Namenskonflikten. Zur Umbenennung von Variablen und Funktionen wird je ein Index gehalten. Der Evaluator-Zustand besteht zusätzlich aus den Funktionsdeklarationen des Programms und einem Stack-Baum:

```
data EvalState = EState [FuncDecl] StackTree Int Int
```

Auf die Bedeutung des Stack-Baums wird später noch eingegangen.

7.1.2. Die Funktion `getRedex`

Um die Funktionsweise des Evaluators zu beschreiben, wird die Funktion `getRedex` definiert, welche zu einem Ausdruck eine Redex-Liste bestimmt. Eine Redex-Liste enthält alle Informationen, die notwendig sind, um den nächsten Reduktionsschritt durchführen zu können: Die Redexe, die im nächsten Schritt reduziert werden können, die Ausdrücke durch die ein bestimmter Redex ersetzt werden kann und die neuen Zustände, die sich dann ergeben. Zu Ausdrücken, in denen nichtdeterministische Funktionsaufrufe vorkommen, kann es im Allgemeinen mehrere Redexe geben, die im nächsten Schritt reduziert werden können. Dies wird am parallelen `and` aus Kapitel 5 deutlich:

```
and False _ = False
and _ False = False
and True True = True

f = False

g = False
```

Der Ausdruck `(and f g)` kann wegen den beiden überlappenden Regeln in einem Schritt zu `(and False g)` oder `(and f False)` reduziert werden. `f` und `g` sind also beides Redexe, die im ersten Schritt reduziert werden können. Weiterhin kann es ebenfalls aufgrund von Nichtdeterminismus zu jedem Redex mehrere mögliche Ersetzungen geben. Die Funktion `getRedex` hat somit folgenden Typ:

```
getRedex :: EvalState -> Expr -> ([(Expr,[(EvalState,Expr)])],Bool)
```

Das erste Argument vom Typ `EvalState` ist der Evaluator-Zustand, das zweite Argument ist der Ausdruck, der ausgewertet werden soll. Die Funktion liefert dann eine Redex-Liste und einen booleschen Wert als Ergebnis. Eine Redex-Liste besteht aus Redexen, den entsprechenden Reduktionsmöglichkeiten und den jeweils neuen Zuständen. Der boolesche Wert gibt an, ob überhaupt eine Reduktion möglich ist.

Variablen und Literale können nicht weiter reduziert werden:

```
getRedex est (Var x) = ([(Var x,[Var x])],False)
getRedex est (Lit l) = ([(Lit l,[Lit l])],False)
```

Bei bewachten Ausdrücken (`GuardedExpr`) wird zwischen drei Fällen unterschieden:

```
getRedex est (GuardedExpr vs (Comb FuncCall c es) e)
  | c=="success" =
    ([(GuardedExpr vs (Comb FuncCall c es) e,[(est,e)])],True)
  | otherwise =
    getRedex est (Comb FuncCall c es)

getRedex est (GuardedExpr _ (Constr vs cons) _) =
  getRedex est2 cons2
```

Falls der Wächter (guard) `"success"` ist, kann der gesamte Ausdruck zu der rechten Seite `e` reduziert werden. Handelt es sich bei dem Wächter um einen Funktionsaufruf, so muß dieser zunächst reduziert werden. Handelt es sich um einen Ausdruck der Form `(Constr vs cons)`, also ein Constraint mit freien Variablen `vs`, so wird zunächst dieser reduziert.

Ein `Constr`-Ausdruck wird einfach durch den Teilausdruck `e` ersetzt:

```
getRedex est (Constr vs e) = ([(Constr vs e,[(est,e)])],True)
```

Ein Ausdruck mit einem Datenkonstruktor an der Wurzel kann reduziert werden, falls einer seiner Teilausdrücke reduziert werden kann:

```
getRedex est (Comb ConsCall c es) = get es
  where get [] =
          ([(Comb ConsCall c es,[(est,Comb ConsCall c es)])],False)
        get (e:es2) =
          let (res,red)=getRedex est e
          in if red then (res,True) else get es2
```

Die Funktion `get` testet von links nach rechts alle Ausdrücke in `es` auf Reduzierbarkeit. Falls einer reduzierbar ist, ist dessen Redex-Liste auch die Redex-Liste für den gesamten Ausdruck.

Mit `Apply`-Ausdrücken werden partielle Applikation und Funktionen höherer Ordnung dargestellt. Handelt es sich beim ersten Argument nicht auch um einen `Apply`-Ausdruck, wird der zweite Ausdruck einfach in die Argumentliste des partiellen Funktionsaufrufs `(Comb PartCall c es)` übernommen. Ansonsten muß zunächst der erste Ausdruck ausgewertet werden:

```
getRedex est (Apply (Apply e1 e2) e3) =
  getRedex est (Apply e1 e2)
```

```
getRedex est (Apply (Comb PartCall c es) e)
  | (getArity (getFuncDecls est) (getOrigName c))==(length es)+1 =
    ([(Apply (Comb PartCall c es) e,
     [(est,Comb FuncCall c (es++[e]))])],True)
  | otherwise =
    ([(Apply (Comb PartCall c es) e,
     [(est,Comb PartCall c (es++[e]))])],True)
   where getArity ((Func f1 ar _ _):fds) f2
         | f1==f2 = ar
         | otherwise = getArity fds f2
```

Handelt es sich bei dem erweiterten Funktionsaufruf nicht mehr um einen partiellen Aufruf, kann der gesamte Ausdruck zu einem normalen Funktionsaufruf reduziert werden, ansonsten wird er zu einem neuen partiellen Aufruf reduziert. Die beiden Fälle werden über die Stelligkeit der Funktion `c` unterschieden, die mit der Funktion `getArity` berechnet werden kann. Mit der Funktion `getOrigName` wird eine notwendige Umbenennung der Funktion `c` rückgängig gemacht (siehe unten).

Ein Funktionsaufruf wird durch die entsprechende rechte Seite ersetzt:

```
getRedex est (Comb FuncCall c es) =
  let (est2,e2)=apply est c es
  in ([(Comb FuncCall c es,[(est2,e2)])],True)
```

Die Funktion `apply` bestimmt zu einem Funktionsaufruf die entsprechende rechte Seite, durch die dieser ersetzt werden muß. Dies geschieht in zwei Schritten: Zunächst muß die Deklaration der Funktion `c` aus der Liste der Funktionsdeklarationen, die Teil des Evaluator-Zustands ist, herausgesucht werden. Die so gefundene Regel `(Rule vs e)` für `c` enthält die rechte Seite `e`, die jedoch noch modifiziert werden muß: Die formalen Parameter `vs` müssen in `e` durch die aktuellen Parameter `es` ersetzt werden. Variablen, die durch `Case`-Ausdrücke in `e` eingeführt werden, und freie Variablen müssen zur Vermeidung von Namenskonflikten umbenannt werden. Schließlich wird in jedem Funktionsaufruf, der in `e` vorkommt, der Funktionsname an der Wurzel durch Voranstellen eines Präfixes umbenannt. Hierfür gibt es den zweiten Index im Evaluator-Zustand. Der ursprüngliche Name kann mit der Funktion `getOrigName` bestimmt werden. Diese Umbenennung ist notwendig, damit beispielsweise die Funktion

```
test = z + z
```

korrekt ausgewertet wird: Die nichtdeterministische Funktion `z` sei so wie in Beispiel 7.1.1 definiert. Falls nun das erste `z` der aktuelle Redex wäre, würde das zweite ohne Umbenennung im selben Schritt wie das erste reduziert. Es würde als nur zwei statt drei Ergebnisse geben. Die Umbenennung sorgt somit dafür, dass auf der rechten Seite wirklich zwei verschiedene Funktionsaufrufe stehen. Schließlich gibt `apply` also den aufgrund der Umbenennungen aktualisierten Evaluator-Zustand und die geänderte rechte Seite `e2` zurück. Die Umbenennung der Funktionsnamen soll natürlich nur intern erfolgen. Falls die Ergebnisse der Reduktion ausgegeben werden sollen muß sie also wieder rückgängig gemacht werden. Dies kann durch einen entsprechenden Pretty-Printer geschehen.

Bei `Case`-Ausdrücken werden die folgenden Fälle unterschieden:

```
getRedex est (Case _ (Comb FuncCall c es) _) =
  getRedex est (Comb FuncCall c es)
getRedex est (Case _ (Apply e1 e2) _) =
  getRedex est (Apply e1 e2)
getRedex est (Case _ (GuardedExpr vs e1 e2) _) =
  getRedex est (GuardedExpr vs e1 e2)
```

Handelt es sich bei dem Argument, dessen Wert für das pattern matching benötigt wird, um einen Funktionsaufruf oder einen bewachten Ausdruck, so muß dieser zunächst ausgewertet werden.

Falls das Argument einen Konstruktor `c` an der Wurzel hat, kann das pattern matching durchgeführt werden:

```
getRedex est (Case ct (Comb ConsCall c es) ps) = match c es ps
  where match _ _ [] =
          ([(Case ct (Comb ConsCall c es) ps,[])],False)
        match c1 es ((Pattern (Ident c2) vs e):ps2)
          | c1==c2 =
            ([(Case ct (Comb ConsCall c ps) ps,
               [(est,bindFParams vs es e)])],True)
          | otherwise =
            match est c1 es ps2
```

Dazu wird `c` von der lokalen Funktion `match` mit allen Konstruktoren in den Patterns aus `ps` verglichen. Falls der richtige Konstruktor `c2` gefunden wird, kann der gesamte `Case`-Ausdruck zu dem Ausdruck `e` des entsprechenden Patterns reduziert werden. Dabei müssen noch die formalen Parameter `vs` in `e` mit der Funktion `bindFParams` durch die aktuellen Parameter `es` ersetzt werden. Es fehlt noch eine entsprechende Regel für `getRedex`, welche `Case`-Ausdrücke mit Literalen als Argument behandelt. Diese kann völlig analog zu obiger Regel definiert werden.

Ist das Argument eine freie Variable, sind zwei Fälle zu unterscheiden:

```
getRedex est (Case Rigid (Var x) ps) =
  ([(Case Rigid (Var x) ps,[])],False)
getRedex est (Case Flex (Var x) ps) =
  bind est ps []
  where bind est [] res = ([(Var x,map (\e->(est,e)) res)],True)
        bind est ((Pattern (Ident i) vs _):ps2) res =
          let vs2=map (\x->Var x) vs
              (est2,e)=makeVarsFresh est (Comb ConsCall c vs2) vs
          in bind est2 ps2 (res++[e])
        bind est ((Pattern (Intc i) [] _):ps2) res =
          bind est ps2 (res++[Lit (Intc i)])
        bind est ((Pattern (Floatc f) [] _):ps2) res =
          bind est ps2 (res++[Lit (Floatc f)])
        bind est ((Pattern (Charc c) [] _):ps2) res =
          bind est ps2 (res++[Lit (Charc c)])
```

Falls der `Case`-Ausdruck zu einer Funktionsdeklaration mit der Auswertungsannotation `eval rigid` gehört, wird er mittels Residuation ausgewertet. Somit kann er nicht reduziert werden, die Auswertung dieses Ausdrucks suspendiert. Ist die Auswertungsannotation dagegen `eval flex`, so muß die Variable `(Var x)` an alle Patterns, die in `ps` vorkommen, gebunden werden. Dazu sammelt die lokale Funktion `bind` alle Patterns auf und ersetzt die in ihnen vorkommenden Variablen durch frische Variablen. Schließlich ist `(Var x)` der Redex, der dann zu allen Patterns reduziert werden kann.

`Or`-Ausdrücke werden wie folgt behandelt:

```
getRedex est (Or e1 e2) =
  ([(Or e1 e2,[(est,e1),(est,e2)])],True)
```

Der Ausdruck `(Or e1 e2)` kann entweder zu `e1` oder `e2` reduziert werden.

Mit der Funktion `getRedex` läßt sich nun die Auswertung eines Ausdrucks recht einfach beschreiben. Die Funktion `evaluate` wertet einen Ausdruck `e` schrittweise aus:

```
evaluate est e = let (res,red)=getRedex est e
                 in putStrLn (ppExpr 0 e) >>
                    getChar >>
                    aux res red
  where aux _ False =
          putStrLn "no more solutions"
        aux ((redex,((est,e2):_)):_) True =
          evaluate est (set redex e e2)
```

Zunächst wird die Funktion `getRedex` mit dem aktuellen Evaluator-Zustand `est` und dem aktuellen Ausdruck `e` aufgerufen. Dann wird der aktuelle Ausdruck ausgegeben und auf einen Tastendruck gewartet. Ein Ausdruck kann mit der Funktion `ppExpr` in einen String ungewandelt werden. Mit der lokalen Funktion `aux` wird eine Fallunterscheidung durchgeführt: Falls `e` nicht reduzierbar ist, also falls `red` den Wert `False` hat, wird lediglich die Meldung `"no more solutions"` ausgegeben. Ansonsten wird die erste Reduktionsmöglichkeit ausgewählt, alle anderen werden ignoriert. Mit der Funktion `set` wird der Redex `redex` im gesamten Ausdruck `e` durch den Ausdruck `e2` ersetzt. Schließlich wird `evaluate` rekursiv mit dem neuen Zustand und dem mittels `set` bestimmten neuen Ausdruck aufgerufen. Es handelt sich also um einen unvollständigen Evaluator: Bei nichtdeterministischen Berechnungen wird stets maximal eine Lösung gefunden. Er muß noch um Backtracking erweitert werden. Dazu ist ein Stack notwendig, auf dem die Alternativen abgelegt werden. Falls keine weitere Reduktion mehr möglich ist, wird mit dem Ausdruck fortgefahren, der oben auf dem Stack liegt.

Um die Unterteilung der Reduktion in zwei Teilschritte zu verstecken, kann eine Funktion

```
reduce :: EvalState -> Expr -> ([(Expr,[(EvalState,Expr)])],Bool)
```

definiert werden. Der Evaluator-Zustand kann zusätzlich durch Verwendung eines Objektes versteckt werden.

Die bisherige Implementierung von `getRedex` hat noch einen entscheidenden Nachteil: Die Ausdrücke, die im Laufe der Auswertung entstehen, können aus `Case-` und `Or-`Ausdrücken bestehen. Dies ist jedoch nicht wünschenswert. Diese Ausdrücke erleichtern zwar die Implementierung des Evaluators, sollten jedoch für den Benutzer nicht sichtbar sein, da sie in Curry nicht vorkommen und somit sehr verwirrend sein können. Im nächsten Abschnitt wird deshalb eine modifizierte Funktion `getRedex` vorgestellt, die diesen Nachteil behebt. Bei dieser modifizierten Funktion tritt dann auch der Fall auf, dass ein Ausdruck mehrere verschiedene Redexe haben kann, die im nächsten Schritt reduziert werden können.

7.1.3. Verbesserung von `getRedex`

Das Auftreten von `Case`- und `Or`-Ausdrücken wird durch die folgenden Veränderungen vermieden: Da diese Ausdrücke nur bei der Reduktion eines Funktionsaufrufes entstehen können, muß lediglich die Regel für Funktionsaufrufe geeignet abgeändert werden. Da dann keine `Case`- und `Or`-Ausdrücke mehr vorkommen, werden die Regeln, die diese Ausdrücke behandeln, nicht mehr benötigt.

Falls ein Funktionsaufruf `(Comb FuncCall c es)` nicht zu einem `Case`- oder `Or`-Ausdruck reduziert wird, die rechte Seite der Funktion `c` also keine solchen Ausdrücke enthält, gibt es kein Problem. Die neue Funktion `getRedex` leistet in diesem Falle das gleiche wie die alte, allerdings etwas umständlicher formuliert. Im anderen Falle darf die Reduktion nicht durchgeführt werden, weil sonst `Case`- oder `Or`-Ausdrücke im reduzierten Ausdruck vorkommen würden. In diesem Falle wird eventuell ein `Case`-Ausdruck mit der Funktion `pushExpr` auf den Stack, der Teil des Evaluator-Zustandes ist, gelegt und die Reduktion wird solange auf dem Stack durchgeführt, bis sich ein Ausdruck ergibt, der weder `Case`- noch `Or`-Ausdrücke enthält. Da die Reduktionen auf dem Stack auch den auszuwertenden Ausdruck betreffen, müssen diese auch auf diesen angewendet werden. Ein Stack wird wegen möglicher geschachtelter Funktionsaufrufe benötigt. Die modifizierte Regel der Funktion `getRedex` sieht dann wie folgt aus:

```
getRedex est (Comb FuncCall c es) | not (isPredefined c) =
  if not (stackEmpty est) then get (popExpr est) (topExpr est)
    else let (est2,e2)=apply est c es in get est2 e2
  where get est3 (Var x) =
          ([(Comb FuncCall c es,[(est3,Var x)])],True)
        get est3 (Lit l) =
          ([(Comb FuncCall c es,[(est3,Lit l)])],True)
        get est3 (Comb ct c2 es2) =
          ([(Comb FuncCall c es,[(est3,Comb ct c2 es2)])],True)
        get est3 (Constr vs e) =
          ([(Comb FuncCall c es,[(est3,Constr vs e)])],True)
        get est3 (GuardedExpr vs co e3) =
          ([(Comb FuncCall c es,[(est3,GuardedExpr vs co e3)])],True)
        get est3 (Or expr1 expr2) =
          let (res1,red1)=get est3 expr1
              (res2,red2)=get est3 expr2
          in ((if red1 then res1 else [])++
              (if red2 then res2 else []),red1 || red2)
        get est3 (Case ct e ps) =
          match est3 (Case ct e ps) ps
```

Falls der Stack leer ist, muß die rechte Seite mit `apply` bestimmt werden, ansonsten wird sie einfach vom Stack genommen. Mit der lokalen Funktion `get` wird nun eine Fallunterscheidung durchgeführt: Handelt es sich bei der rechten Seite weder um einen `Case`- noch um einen Or-Ausdruck, ergibt sich das gleiche Ergebnis wie bei der alten Definition von getRedex: Der Funktionsaufruf ist der Redex und kann zu der entsprechende rechten Seite reduziert werden. Handelt es sich um einen Or-Ausdruck, so werden die Redex-Listen beider Alternativen bestimmt, das Ergebnis ist dann die Konkatenation dieser Listen. Hier tritt der Fall auf, das es mehrere mögliche Redexe gibt, der bei der ersten Implementierung nicht auftrat.

Falls es sich um einen Case-Ausdruck handelt, wird mit der zu getRedex lokalen Funktion match die gesuchte Redex-Liste bestimmt. Diese hat den Evaluator-Zustand, den Case-Ausdruck und die Liste ps der Patterns als Argumente. Hat das Argument e, dessen Wert für das pattern matching benötigt wird, an der Wurzel einen Datenkonstruktor, so muß die passende Regel aus ps gesucht werden:

```
match est (Case ct (Comb ConsCall _ _) _) [] =
  ([(Comb FuncCall c es,[])],False)
match est (Case ct (Comb ConsCall c1 es2) ps)
      ((Pattern (Ident c2) vs e):ps2)
  | (c1==c2) = get est (bindFParams vs es2 e)
  | otherwise = match est (Case ct (Comb ConsCall c1 es2) ps) ps2
```

Die formalen Parameter vs werden dann an die aktuellen Parameter es gebunden und es wird mit dem Teilausdruck e weiter gemacht. Für Literale als Argumente werden noch zwei analoge Regeln benötigt.

Falls das Argument e ein Funktionsaufuf ist, so muß dieser zunächst ausgewertet werden:

```
match est (Case ct (Comb FuncCall c2 es2) ps) _ =
  let (res,red)=getRedex est (Comb FuncCall c2 es2)
      ce=(Case ct (Comb FuncCall c2 es2) ps)
  in if red then (map (\(e1,estes)->
                        (e1,map (\(est,e2)->
                                 (pushExpr est (set e1 ce e2),e2))
                                 estes)) res,red)
         else ([(Comb FuncCall c es,[])],False)
```

Es wird also zunächst die Redex-Liste für den Ausdruck (Comb FuncCall c2 es2) bestimmt. Ist dieser reduzierbar, so ergibt sich die Redex-Liste für den gesamten Ausdruck aus der für den Funktionsaufruf wie folgt: Der gerade bearbeitete Case-Ausdruck ce wird auf den Stack der Evaluator-Zustände zu den einzelnen Reduktions-Möglichkeiten gelegt. Die jeweils entsprechende Reduktion wird dabei mit der Funktion set auf ce angewendet. Ist der Funktionsaufruf nicht reduzierbar, so gilt dies auch für den gesamten Ausdruck. Entsprechende Regeln können auch für bewachte Ausdrücke und Apply-Ausdrücke als Argument formuliert werden.

Ist das Argument eine Variable, werden zwei Fälle unterschieden:

```
match est (Case Rigid (Var x) ps) _ =
  let ce=(Case Rigid (Var x) ps)
  in ([(Comb FuncCall c es,[])],False)
```

```
match est (Case Flex (Var x) ps) ps2 =
  bind est ps2 []
  where bind est [] res =
          let ce=(Case Flex (Var x) ps2)
          in ([(Var x,map (\e->
                          (pushExpr est (set (Var x) ce e),e)) res)],
              True)
        bind est ((Pattern (Ident i) vs _):ps) res =
          let vs2=map (\x->Var x) vs
              (est2,e2)=makeVarsFresh est (Comb ConsCall i vs2) vs
          in bind est2 ps (res++[e2])
        bind est ((Pattern (Intc i) [] _):ps) res =
          bind est ps (res++[Lit (Intc i)])
```

Falls der Ausdruck mit Residuation ausgewertet werden soll, kann er noch nicht reduziert werden und die Auswertung dieses Teilausdrucks suspendiert. Ansonsten werden wieder mit der Funktion `bind` alle Patterns, an die die Variable gebunden werden soll, aufgesammelt. Aus der Liste aller Patterns wird dann die Redex-Liste bestimmt. Der `Case`-Ausdruck `ce` muß dazu noch entsprechend modifiziert und auf die Stacks der neuen Evaluator-Zustände gelegt werden.

7.1.4. Implementierung von Constraints

Vordefinierte Funktionen wie `+` oder `*` werden gesondert behandelt. Mit der Funktion `isPredefined` wird geprüft, ob eine Funktion vordefiniert ist. Auch `=:=` wird als vordefinierte Funktion repräsentiert. Für vordefinierte Funkionen gibt es eine spezielle Regel von `getRedex`:

```
getRedex est (Comb FuncCall c es) | (isPredefined c) = get c es
  where get "+" (e1:e2:[])
          | (isGround e1) && (isGround e2) =
            ([(Comb FuncCall c es,[(est,add e1 e2)])],True)
          | isGround e1 = getRedex est e2
          | otherwise = getRedex est e1
          where add (Lit (Intc i1)) (Lit (Intc i2)) =
                      Lit (Intc (i1+i2))
                add (Var _) (Var _) =
                      Comb FuncCall c es
```

Für jede vordefinerte Funktion müssen passende Regeln der lokalen Funktion `get` definiert werden. Für `+` werden drei Fälle unterschieden: Falls beide Argumene `e1` und `e2` bereits komplett ausgewertet sind, können diese einfach addiert werden. Ansonsten müssen zunächst noch die noch nicht komplett ausgewerteten Argumente ausgewertet werden.

Gleicheits-Constraint werden in ähnlicher Art und Weise implementiert, wie es in [He00a] beschrieben ist. Es wird eine Fallunterscheidung über das erste Argument gemacht: Handelt es sich dabei um eine Variable, so muß diese eventuell gebunden werden:

```
get "=:=" ((Var x):(Lit l):[]) =
  ([(Var x,[(est,Lit l)])],True)
```

```
get "=:=" ((Var x):(Comb ConsCall c2 es2):[])
  | x `elem` (cv (Comb ConsCall c es)) =
    ([(Comb FuncCall c es,[])],False)
  | otherwise =
    let (est2,fv)=getfreshvars (length es2) est
    in ([(Var x,[(est2,Comb ConsCall c2 fv)])],True)
get "=:=" ((Var x):(Var y):[])
  | x==y =
    ([(Comb FuncCall c es,[(est,Comb FuncCall "success" [])])],True)
  | otherwise =
    ([(Var x,[(est,Var y)])],True)
get "=:=" ((Var x):(Comb FuncCall c es):[]) =
  getRedex est (Comb FuncCall c es)
get "=:=" ((Var x):(Apply e1 e2):[]) =
  getRedex est (Apply e1 e2)
get "=:=" ((Var x):(Constr vs e):[]) =
  getRedex est (Constr vs e)
get "=:=" ((Var x):(GuardedExpr vs co e):[]) =
  getRedex est (GuardedExpr vs co e)
```

Ist das zweite Argument eine Literal, so kann die Variable direkt an dieses gebunden werden. Handelt es sich um einen Ausdruck mit einem Datenkonstruktor an der Wurzel, muß zunächst der Occur-Check gemacht werden: Dazu wird die Menge der kritischen Variablen `(cv (Comb ConsCall c2 es2))` dieses Ausdrucks bestimmt. Falls `x` darin vorkommt, schlägt der Occur-Check fehl. Die Menge der kritischen Variablen besteht aus den Variablen, die in `(Comb ConsCall c2 es2)` ausserhalb eines Funktionsaufrufes vorkommen (siehe [He00a]). Wenn der Occur-Check fehl schlägt, kann der gesamte Ausdruck nicht weiter reduziert werden. Ansonsten wird `(Var x)` zu `(Comb ConsCall c2 fv)` reduziert, wobei `fv` eine Liste frischer Variablen ist. Wenn das zweite Argument eine Variable ist, werden wieder zwei Fälle unterschieden: Entweder die beiden Argumente sind gleich, dann kann der gesamte Ausdruck zu `"success"` reduziert werden, oder sie sind verschieden. In diesem Fall kann die erste Variable zu der zweiten reduziert werden. Handelt es sich bei dem zweiten Argument um einen Funktionsaufruf, Constraint oder bewachten Ausdruck, so muß dieser zunächst weiter ausgewertet werden.

Ein Literal `l1` kann mit einem zweiten Literal `l2` unifiziert werden, falls `l1==l2` gilt:

```
get "=:=" ((Lit l1):(Lit l2):[])
  | l1==l2 =
    ([(Comb FuncCall c es,[(est,Comb FuncCall "success" [])])],True)
  | otherwise =
    ([(Comb FuncCall c es,[])],False)
get "=:=" ((Lit l):(Var x):[]) =
  get "=:=" ((Var x):(Lit l):[])
get "=:=" ((Lit _):(Comb FuncCall c2 es2):[]) =
  getRedex est (Comb FuncCall c2 es2)
get "=:=" ((Lit _):(Comb ConsCall c2 es2):[]) =
  ([(Comb FuncCall c es,[])],False)
get "=:=" ((Lit _):(Apply e1 e2):[]) =
  getRedex est (Apply e1 e2)
```

```
get "=:=" ((Lit _):(Constr vs e):[]) =
  getRedex est (Constr vs e)
get "=:=" ((Lit _):(GuardedExpr vs co e):[]) =
  getRedex est (GuardedExpr vs co e)
```

Falls das zweite Argument ein Ausdruck mit Datenkonstruktor an der Wurzel ist, können die beiden Ausdrücke nicht unifiziert werden. Ist das zweite Argument eine Variable, so wird das Ergebnis durch Vertauschen der beiden Argumente gefunden. In allen anderen Fällen muß erst noch das zweite Argument weiter ausgewertet werden.

Ein Ausdruck mit Datenkonstruktor an der Wurzel kann mit einer Variable oder einem Ausdruck mit gleichem Konstruktor an der Wurzel unifiziert werden:

```
get "=:=" ((Comb ConsCall c2 es2):(Var x):[]) =
  get "=:=" ((Var x):(Comb ConsCall c2 es2):[])
get "=:=" ((Comb ConsCall c1 es1):(Comb ConsCall c2 es2):[])
  | c1==c2 = getArgs es1 es2
  | otherwise = ([(Comb FuncCall c es,[])],False)
  where getArgs [] _ =
          ([(Comb FuncCall c es,
            [(est,Comb FuncCall "success" [])])],True)
        getArgs (e1:e1s) (e2:e2s)
          | e1==e2 = getArgs e1s e2s
          | otherwise = let (res,red)=get "=:=" [e1,e2]
                        in if red then (res,True)
                              else ([(Comb FuncCall c es,[])],False)
get "=:=" ((Comb ConsCall _ _):(Lit _):[]) =
  ([(Comb FuncCall c es,[])],False)
get "=:=" ((Comb ConsCall _ _):(Comb FuncCall c2 es2):[]) =
  getRedex est (Comb FuncCall c2 es2)
get "=:=" ((Comb ConsCall _ _):(Apply e1 e2):[]) =
  getRedex est (Apply e1 e2)
get "=:=" ((Comb ConsCall _ _):(Constr vs e):[]) =
  getRedex est (Constr vs e)
get "=:=" ((Comb ConsCall _ _):(GuardedExpr vs co e):[]) =
  getRedex est (GuardedExpr vs co e)
```

Ist das zweite Argument eine Variable, wird das Ergebnis wieder durch Vertauschen der Argumente gefunden. Ist es ein Ausdruck mit Konstruktor `c2` an der Wurzel, so können die beiden Ausdrücke nur dann unifiziert werden, wenn dieses der gleiche wie `c1` ist. Zusätzlich müssen dann noch die Argumente in den Listen `es1` und `es2` unifiziert werden. Dies wird mit der lokalen Funktion `getArgs` erledigt. Falls schließlich die beiden Ausdrücke gleich sind, kann der gesamte Ausdruck zu `"success"` reduziert werden. In allen anderen Fällen muß zunächst noch das zweite Argument ausgewertet werden.

Falls das erste Argument ein Constraint, bewachter Ausdruck oder Funktionsaufruf ist, so muß dieses zuerst weiter ausgewertet werden, was durch die letzten vier Regeln erreicht wird:

```
get "=:=" ((Constr vs e):_:[]) =
  getRedex est (Constr vs e)
get "=:=" ((GuardedExpr vs co e):_:[]) =
```

```
    getRedex est (GuardedExpr vs co e)
  get "=:=" ((Comb FuncCall c2 es2):_:[]) =
    getRedex est (Comb FuncCall c2 es2)
  get "=:=" ((Apply e1 e2):_:[]) =
    getRedex est (Apply e1 e2)
```

Die nebenläufige Konjunktion & wird wie folgt implementiert:

```
  get "&" (e1:e2:[])
    | e1==(Comb FuncCall "success" []) =
      ([(Comb FuncCall c es,[(est,e2)])],True)
    | e2==(Comb FuncCall "success" [])=
      ([(Comb FuncCall c es,[(est,e1)])],True)
    | otherwise =
      let (res,red)=getRedex (takeLeft est) e1
      in if red then
           (map (\(e1,estes)->
                  (e1,map (\(est2,e2)->
                            (setLeft est est2,e2)) estes)) res,True)
           else
             let (res,red)=getRedex (takeRight est) e2
             in if red then
                  (map (\(e1,estes)->
                         (e1,map (\(est2,e2)->
                                   (setRight est est2,e2)) estes)) res,
                   True)
                  else ([(Comb FuncCall c es,
                         [(est,Comb FuncCall c es)])],False)
```

Falls eines der beiden Argumente bereits zu `"success"` reduziert werden konnte, kann der gesamte Ausdruck zu dem anderen Argument reduziert werden. Ansonsten wird zunächst geprüft, ob das erste Argumnent reduziert werden kann. Ist dies der Fall, so ist die zugehörige Redex-Liste die gesuchte Redex-Liste. Ansonsten wird das zweite Argument untersucht. Ist auch dieses auch nicht reduzierbar, gilt dies auch für die Konjunktion. Falls es reduzierbar ist, ist die entsprechende Redex-Liste wieder das gesuchte Ergebnis.

Durch die nebenläufige Auswertung der beiden Constraints `e1` und `e2` ergibt sich ein neues Problem: Es müssen die Stacks für die Auswertung beider Ausdrücke im Evaluator-Zustand gehalten werden. Ein einfacher Stack reicht also nicht mehr aus. Vielmehr ist ein binärer Baum notwendig, dessen Knoten Stacks enthalten. Die Funktionen `pushExpr`, `popExpr` und `topExpr` beziehen sich stets auf den Stack an der Wurzel. Falls nun der linke Ausdruck `e1` auf Reduzierbarkeit untersucht wird, wird mit der Funktion `takeLeft` zum linken Teilbaum übergegangen. Enstsprechend wird bei `e2` zum rechten Teilbaum übergegangen. Falls eine Redex-Liste gefunden wurde, müssen die modifizierte linken oder rechten Teilbäume der einzelnen Alternativen mit den Funktionen `setLeft` beziehungsweise `setRight` jeweils im alten Baum entsprechend eingefügt werden.

In dieser einfachen Version des Evaluators werden nebenläufige Constraints nicht fair ausgewertet: Der zweite Constraint `e2` wird lediglich dann weiter ausgewertet, wenn der erste suspendiert. Um eine faire Auswertung zu implementieren, müssen im Wesentlichen lediglich entsprechende Informationen im Evaluator-Zustand gehalten werden.

Die Implementierung von commited choice wurde hier bewußt ausgelassen. Stattdessen wird im nächsten Abschnitt kurz skizziert, wie Ports implementiert werden können, die eine effizientere n-zu-1 Kommunikation als mit commited choice erlauben.

7.1.5. Implementierung von internen Ports

Interne Ports können mittels Strömen implementiert werden. Dazu muß der Evaluator-Zustand geeignet erweitert werden: Zu jedem Port `p`, der mit `(openPort p s)` geöffnet wurde, wird eine Variable im Zustand eingetragen, an die die nächste Nachricht `msg`, die an `p` durch den Funktionsaufruf `(send msg p)` gesendet wird, gebunden werden soll. Der Ausdruck `(openPort p s)` kann stets zu `"success"` reduziert werden:

```
get "openPort" ((Var p):(Var s):[]) =
  ([(Comb FuncCall c es,
     [(addPort est p s,Comb FuncCall "success")])],True)
```

Dabei müssen der Port `p` und die zugehörige Variable `s` noch mit der Funktion `addPort` im Evaluator-Zustand eingetragen werden. Ein Funktionsaufruf `(send msg p)` kann reduziert werden, falls der Port `p` vorher geöffnet wurde:

```
get "send" (msg:(Var p):[])
  | isOpen p =
    let (est2,((Var fv):_))=getfreshvars 1 est
    in ([(Comb FuncCall c es,
          [(setPortVar est2 p fv,
            Comb FuncCall "=:="
                 [Var (getPortVar est p),
                  Comb ConsCall ":" [msg,Var fv]])])],
        True)
  | otherwise =
    ([(Comb FuncCall c es,[])],False)
```

Der Ausdruck wird dann zu einem Gleichheits-Constraint reduziert. Dieser bewirkt, dass die dem Port zugeordnete Variable `(getPortVar p s)` an den Ausdruck `(msg:(Var fv))` gebunden wird. Somit kann die Nachricht `msg` vom Empfänger gelesen werden. An die frische Variable `(Var fv)` soll die nächste Nachricht gebunden werden. Dazu muß sie mit der Funktion `(setPortVar est2 p fv)` im Evaluator-Zustand eingetragen werden.

Interne Ports wurden noch nicht implementiert.

7.2. Implementierung mit Bindungs-Tabellen

Die vorgestellte Implementierung mit Redex-Listen ist nicht sehr effizient. Dies liegt hauptsächlich daran, dass zur Durchführung eines Reduktions-Schritt jeweils der gesamte Ausdruck nach dem Redex durchsucht werden muß. Die Ausdrücke, die während der Auswertung entstehen, können im Allgemeinen sehr groß werden.

Das jeweilige Durchsuchen des gesamten Ausdrucks kann vermieden werden, wenn die Bindungen der Variablen nicht direkt im Ausdruck eingetragen werden, sondern in einer separaten Bindungs-Tabelle. Ein Reduktionsschritt kann somit sofort durchgeführt werden, er besteht nicht mehr aus zwei Teilschritten. Die Funktion `reduce`, die einen solchen Schritt durchführen soll, hat dann folgenden Typ:

```
reduce :: EvalState -> Expr -> ([(EvalState,Expr)],Bool)
```

Der Evaluator-Zustand enthält jetzt zusätzlich noch die Bindungs-Tabelle. Das zweite Argument ist der zu reduzierende Ausdruck. Es ergibt sich eine Liste von Reduktionsmöglichkeiten, die aus Paaren von neuen Zuständen und reduzierten Ausdrücken besteht. Der boolesche Wert, der zusätzlich zurückgegeben wird, zeigt wieder an, ob eine Reduktion möglich war.

Ist das zweite Argument eine Variable, so muß zunächst mit der Funktion `getBinding` ihre Bindung bestimmt werden:

```
reduce est (Var x)
  | isVar binding =
    ([(est,Var x)],False)
  | otherwise =
    let (res,red)=reduce est binding
    in (map (\(est2,e2)->
              (setBinding est2 (Var x) e2,Var x)) res,red)
  where binding=getBinding est (Var x)
```

Ist die Variable ungebunden oder an eine andere Variable gebunden, so wird keine Reduktion durchgeführt. Ansonsten wird der Ausdruck, an den die Variable gebunden ist reduziert. Es ergibt sich die Liste `res`, die die verschiedenen Reduktionsmöglichkeiten enthält. Hieraus wird dann das endgültige Ergebniss berechnet, indem die Variable mit der Funktion `setBinding` noch an die verschiedenen reduzierten Ausdrücke gebunden wird.

Literale können nicht weiter reduziert werden:

```
reduce est (Lit l) = ([(est,Lit l)],False)
```

Ausdrücke mit einem Datenkonstruktor an der Wurzel können reduziert werden, falls einer der Teilausdrücke reduziert werden kann:

```
reduce est (Comb ConsCall c es) = red es []
  where red [] _ =
          ([(est,Comb ConsCall c es)],False)
        red (e:es1) es2 =
          let (res,red)=reduce est e
          in if red then (map (\(est2,e2)->
                                (est2,Comb ConsCall c
                                           (es2++[e2]++es1)))
                          res,True)
             else red es1 (es2++[e])
```

Es wird mit der lokalen Funktion `red` von links nach rechts geprüft, ob einer der Teilaudrücke reduzierbar ist.

Funktionsaufrufe werden wie folgt behandelt:

```
reduce est (Comb FuncCall c es) =
  if not (stackEmpty est) then red (popExpr est) (topExpr est)
    else red est2 e2
  where (est2,e2,fv)=apply est c es
        red est3 (Var x) =
          ([(est3,Var x)],True)
        red est3 (Lit l) =
          ([(est3,Lit l)],True)
        red est3 (Comb ConsCall c2 es2) =
          ([(est3,Comb ConsCall c2 es2)],True)
        red est3 (Comb FuncCall c2 es2) =
          ([(est3,Comb FuncCall c2 es2)],True)
        red est3 (Or e1 e2) =
          let (res1,red1)=red est3 e1
              (res2,red2)=red est3 e2
          in (res1++res2,red1 || red2)
        red est3 (Case ct e ps) =
          branch est3 (Case ct e ps) (getBinding est3 e)
```

Die Struktur der Regel ist ähnlich wie die der entsprechende Regel aus dem vorangegangenem Abschnitt. Die Funktion `apply` leistet nun folgendes: Zunächst wird die passende Funktionsdeklaration für die Funktion `c` gesucht. In der rechten Seite werden dann alle Variablen (auch die formalen Parameter) durch frische Variablen ersetzt. In der Bindungs-Tabelle werden die Bindungen für die Variablen eingetragen, durch die die formalen Parameter ersetzt wurden. Zusätzlich werden diese Variablen noch als dritter Wert neben dem neuen Evaluator-Zustand und der modifizierten rechten Seite zurückgeliefert. Mit der lokalen Funktion `red` wird wieder eine Fallunterscheidung durchgeführt: Ist die durch `apply` bestimmte rechte Seite weder `Or-` noch `Case`-Ausdruck, so wird der gesamte Ausdruck einfach zu dieser reduziert, der neue Evaluator-Zustand ist der, den `apply` liefert. Handelt es sich um einen `Or`-Ausdruck, so ergeben sich die Reduktionsmöglichkeiten aus denen der beiden Teilausdrücke `e1` und `e2`.

Bei einem `Case`-Ausdruck muß wieder mit der zu `reduce` lokalen Funktion `branch` eine Fallunterscheidung gemacht werden:

```
branch est3 (Case ct v ((Pattern l2 [] e):ps)) (Lit l1)
  | l1/=l2 = branch est3 (Case ct vs ps) (Lit l1)
  | otherwise = red est3 e
branch est3 (Case ct v ((Pattern (Ident c2) vs e):ps))
       (Comb ConsCall c1 es2)
  | (c1/=c2) =
    branch est3 (Case ct v ps) (Comb ConsCall c1 es2)
  | otherwise = let vs2=map (\x->Var x) vs
                    est4=setBindings est3 vs2 es2
                in red est4 e
```

Falls die Variable `v` an ein Literal oder einen Ausdruck mit Datenkonstruktor an der Wurzel gebunden ist, muß die passende Regel aus den Patterns des `Case`-Ausdrucks gesucht werden. Im zweiten Fall müssen die Variablen in der Liste `vs` noch an die Argumente `es2` gebunden werden. Schließlich wird mit dem Ausdruck `e` aus der passenden Regel fortgefahren.

Falls `v` an einen Funktionsaufruf gebunden ist, so muß dieser zunächst reduziert werden:

```
branch est3 ce (Comb FuncCall c2 es2) =
  let (res,red)=reduce est3 (Comb FuncCall c2 es2)
      newF=if stackEmpty est then Comb FuncCall c fv
             else Comb FuncCall c es
  in if red then (map (\(est4,e4)->
                      (setBinding (pushExpr est4 ce) v e4,newF))
                    res,True)
       else ([(est3,Comb FuncCall c es)],False)
```

Das Ergebnis für den Funktionsaufruf ist die Liste `res`, falls eine Reduktion möglich ist. Als reduzierter Ausdruck wird der Ausdruck `(Comb FuncCall c fv)` zurückgegeben, falls `(Comb FuncCall c es)` zum ersten mal reduziert wurde, ansonsten `(Comb FuncCall c es)`. Die Variablen `fv` sind dabei die Variablen, an die die Argumente `es` des äußersten Funktionsaufrufes gebunden sind. In den einzelnen Alternativen wird der `Case`-Ausdruck auf den Stack des Evaluator-Zustandes gelegt und die Variable `v` wird an den entsprechende reduzierten Ausdruck `e4` gebunden. Die Reduktion läst sich somit indirekt mit Hilfe der Bindungs-Tabelle am reduzierten Ausdruck `(Comb FuncCall c fv)` erkennen. Wenn der gesamte Ausdruck nicht beim ersten mal zu `(Comb FuncCall c fv)` reduziert würde, könnte der Benutzer unter Umständen keinen Fortschritt erkennen, da die reduzierten Ausdrücke ja nur in die Bindungs-Tabelle eingetragen werden.

Es fehlen noch Regeln für `branch` und `reduce`, die bewachte Ausdrücke, Constraints und Funktionen höherer Ordnung behandeln. Diese können jedoch ähnlich wie im vorherigen Abschnitt beschrieben definiert werden.

Um die reduzierten Ausdrücke ausgeben zu können, muß ein Pretty-Printer implementiert werden, der die einzelnen Variablen, die im Ausdruck vorkommen, durch ihre Bindungen ersetzt.

Die Effizienz dieses Evaluators hängt sehr stark von der Implementierung der Bindungs-Tabelle ab. Eine einfache Implementierung mit einer Liste von Paaren ist sicherlich viel zu ineffizient, da die Anzahl der eingetragenen Variablen mit jeder Reduktion ansteigt. Ideal wäre eine Implementierung mit Arrays, die einen Zugriff in konstanter Zeit erlauben. Da in Curry jedoch momentan keine Arrays zur Verfügung stehen, wurden zur Implementierung Rot-Schwarz-Bäume verwendet, die einen Zugriff in logarithmischer Zeit erlauben. Wie der nächste Abschnitt zeigt ist dies jedoch auch unzureichend, so dass diese zweite Variante des Evaluators eher eine Verschlechterung als eine Verbesserung mit sich bringt.

7.3. Vergleich beider Ansätze

Der erste Ansatz führt aus den genannten Gründen zu einer recht ineffizienten Lösung. Durch die Einführung von Bindungs-Listen sollte die Effizienz verbessert werden, allerdings kommt es dabei auf eine effiziente Implementierung der Tabelle an. Beide Ansätze sollen durch die Auswertung des folgenden Programms verglichen werden:

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

```
goal = rev [1..30]
```

Die Funktion `rev` dreht eine Liste um, `goal` appliziert `rev` auf eine Liste, die aus den Integer-Werten 1 bis 30 besteht. Dieses Beispiel reicht zwar nicht für einen fairen Vergleich aus, es ergeben sich jedoch trotzdem sehr aussagekräftige Ergebnisse.

Der erste Evaluator schneidet bei diesem Vergleich deutlich besser ab: Er benötigt für die Auswertung von `goal` ungefähr 75 Sekunden, im Gegensatz zu 300 Sekunden, die der zweite Evaluator benötigt. Beide Implementierungen sind jedoch um ein Vielfaches schlechter als PAKCS. Die Effizienz eines Meta-Evaluators muß zwar zwangsläufig schlechter sein als die des Systems in dem dieser implementiert wurde, trotzdem ist die Effizienz beider Evaluatoren eher entäuschend.

Ein effizienterer Evaluator kann nur durch eine Verbesserung der Implementierung mit Bindungs-Tabellen entstehen. Das Problem beim ersten Evaluator bestand ja darin, dass bei jedem Reduktionsschritt der ganze Ausdruck durchsucht werden musste. Hierdran kann jedoch nichts geändert werden. Beim zweiten Evaluator gibt es jedoch sehr wohl Möglichkeiten zur Verbesserung: Zum Einen kann die Implementierung der Bindungs-Tabellen verbessert werden. Die jetzige Implementierung mit Rot-Schwarz-Bäumen erlaubt zwar einen Zugriff in logarithmischer Zeit, dies ist jedoch offensichtlich nicht ausreichend. Auch das Einfügen von neuen Elementen hat logarithmischen Zeitbedarf. Da das Bestimmen von Bindungen relativ oft benötigt wird, würde eine Implementierung mit Arrays, die einen Zugriff in konstanter Zeit erlauben würden, eine erhebliche Verbesserung mit sich bringen. Zum Anderen kann versucht werden, die Zahl der Variablen, die in die Bindungs-Tabelle eingetragen werden müssen, zu reduzieren. Im obigen Beispiel werden 1891 Variablen benötigt, hier gibt es also ebenfalls einiges Potenzial. Falls eine Regel angewendet wird müssen nicht unbedingt alle Variablen der linken Seite in die Bindungs-Tabelle eingetragen werden. Dies ist beispielsweise der Fall, falls eine Variable in der rechten Seite maximal einmal auftritt. Zu dem Ausdruck, an den eine solche Variable gebunden ist, gibt es dann nämlich keinen zweiten Ausdruck welcher gleichzeitig ausgewertet werden muß. Für diesen Ausdruck ist Sharing dann also nicht notwendig.

8. Ein graphischer Debugger

In diesem Kapitel wird gezeigt, wie auf der Basis des Meta-Evaluators ein Debugger implementiert werden kann. Dieser soll dann in die Programmierumgebung integriert werden.

8.1. Anforderungen

Es soll ein einfacher Tracer implementiert werden. Dieser soll nicht, wie der Tracer aus der PAKCS-Distribution auf dem Box-Modell basieren, sondern auf Termersetzung. Ausdrücke sollen dabei nicht textuell, sondern graphisch als Baum dargestellt werden. Der Redex, der im jeweils nächsten Schritt reduziert wird, soll farblich hervorgehoben werden. Dadurch erhält der Benutzer einen besseren Überblick über die Auswertung. Weiterhin soll jeder Reduktionsschritt auch wieder rückgängig gemacht werden können. Damit nicht jedesmal die ganze Berechnung betrachtet werden muß, soll es die Möglichkeit geben, Haltepunkte zu definieren.

8.2. Implementierung

Die Implementierung besteht aus drei Teilen: Die Bentzerschnittstelle stellt die Ausgaben dar und reagiert auf Eingaben vom Benutzer. Der Server verwaltet alle notwendigen Informationen, die während der Auswertung benötigt werden: Den aktuellen Ausdruck, Redexe, Backtracking-Information und so weiter. Schließlich gibt es noch eine Funktion, die die Koordinaten der einzelnen Baum-Knoten berechnet.

8.2.1. Die Benutzerschnittstelle

Die Benutzerschnittstelle besteht aus drei Teilen: Einer Knopf-Zeile, einem Bereich, in dem die Bäume gezeichnet werden, und einem Eingabe-Fenster zur Eingabe von Haltepunkten. Die Knopf-Zeile besteht aus vier Knöpfen: Je einer zum Schließen des Debuggers, zum Anzeigen des nächsten oder vorherigen Ausdrucks und zum Setzen von Haltepunkten.

Die Schnittstelle kommuniziert mit dem Server durch Versenden von Nachrichten vom Typ `DebugServerMsg`:

```
data DebugServerMsg = NextExpr Expr Expr |
                      PrevExpr Expr Expr |
                      SetBreakPoint String |
                      TerminateDebugger
```

Mit den Nachrichten `NextExpr` und `PrevExpr` kann der nächste oder der vorhergehende Ausdruck und der zugehörige Redex, der farbig hervorgehoben werden soll, gelesen werden. Mit `SetBreakPoint` können Haltepunkte gesetzt werden. Wenn der Benutzer einen Knopf drückt, versenden die entsprechenden Event-Handler die passende Nachricht. Falls ein neuer Ausdruck als Baum dargestellt werden muß, werden die entsprechenden Tk-Elemente mit der Funktion

```
expr2citems ::  Expr -> Expr -> [TkCanvasItem]
```

berechnet und dann im Zeichenbereich ausgegeben. Dabei ist das erste Argument der darzustellende Ausdruck, das zweite der hervorzuhebende Redex.

8.2.2. Der Server

Der Server hält einen Zustand, der alle Informationen enthält, die während des Debuggings benötigt werden. Er empfängt die Nachrichten von der Benutzerschnittstelle, beantwortet sie und ändert seinen Zustand entsprechend. Der Zustand besteht aus folgenden Komponenten:

- Der aktuelle Ausdruck, also der Ausdruck, der mit der Nachricht `NextExpr` abgefragt wird
- Der zugehörige Redex
- Die entsprechenden reduzierten Ausdrücke und die zugehörigen Evaluator-Zustände
- Der Stack für das Backtracking
- Die History-Liste um Reduktionen rückgängig zu machen
- Der Haltepunkt

Wenn der Server die Nachricht `(NextExpr e re)` erhält, führt er die folgenden drei Schritte durch: Zuerst wird überprüft, ob die Berechnung gestoppt werden muß, also ob der Haltepunkt der leere String ist, oder dem Funktionsnamen der Wurzel des aktuellen Redex gleicht. Falls dies der Fall ist, wird `e` an den aktuellen Ausdruck und `re` an den aktuellen Redex gebunden. Im zweiten Schritt wird ein neuer Server-Zustand und ein neuer Nachrichtenstrom bestimmt: Es wird die erste aller möglichen Reduktionen durchgeführt, somit ergibt sich je ein neuer Ausdruck und eine Redex-Liste. Falls mehrere verschiedene Reduktionen möglich sind, müssen die entsprechenden Informationen auf dem Stack abgelegt werden. In der History-Liste werden alter Ausdruck und Redex-Liste gesichert. Falls die Variablen `e` und `re` schon gebunden wurden, ergibt sich der neue Nachrichtenstrom aus dem Reststrom, ansonsten ist der neue Strom gleich dem alten Strom und die Variablen werden erst dann gebunden, wenn die Auswertung anhält. Schließlich ruft sich der Server im dritten Schritt selbst mit neuem Nachrichtenstrom und Zustand auf.

Erhält der Server die Nachricht `(PrevExpr e re)` wird mit Hilfe der History-Liste der alte Zustand wiederhergestellt und die Variablen werden entsprechend gebunden. Bei Empfangen der Nachricht `(SetBreakPoint s)` wird lediglich der Haltepunkt aktualisiert.

8.2.3. Positionierung der Baum-Knoten

In diesem Abschnitt wird der Algorithmus zur Bestimmung der Positionierung der Baum-Knoten erläutert. Er orientiert sich stark an dem in [Tie99] vorgeschlagenen Algorithmus. Die Zeichenfläche wird in Zeilen und Spalten unterteilt. Der Algorithmus soll die folgenden Forderungen erfüllen:

1. Knoten gleicher Tiefe (mit gleichem Abstand von der Wurzel) sollen in der gleichen Zeile positioniert werden.
2. Ein Vater-Knoten soll stets mittig über seinen Kindern plaziert werden.
3. Der seitliche Abstand der Knoten soll einen bestimmten Mindestabstand nicht unterschreiten.

Während sich die Erfüllung der ersten Forderung als recht einfach erweist, erfordert die Erfüllung der zweiten einigen Aufwand.

Die Zeilen- und Spaltennummern der einzelnen Knoten werden in zwei Läufen bestimmt: Zunächst wird ein Ausdruck mit der Funktion `expr2node` in eine interne Darstellung eines Baumes umgewandelt. Ein interner Knoten besteht aus einer Markierung, einer Liste von Kindern, einer Farbe, einer Zeilennummer, einer Spaltennummer und einem Spaltenmodifizierer:

```
data Node = Node String [Node] String Int Int Int
```

Mit der Funktion `node2citems` werden dann in einem zweiten Lauf die Spaltennummern unter Berücksichtigung der Spaltenmodifizierer modifiziert und die Knoten werden in Tk-Elemente übersetzt. Die Strategie zur Vergabe der Spaltennummern ist wie folgt:

1. Jeder Knoten bekommt zunächst die Spaltennummer zugewiesen, die sich aus dem arithmetischen Mittel der Spaltennummern seiner Kinder berechnet. Knoten ohne Kinder (also Blättern) wird die nächste freie Spalte in der entsprechenden Zeile zugewiesen.
2. Falls die einem Knoten zugewiesene Position nicht mehr frei ist, wird der Modifizierer des Knotens so gesetzt, dass diese Überschneidung im zweiten Lauf beseitigt wird.

Die Funktion `expr2node` hat folgenden Typ:

```
expr2node :: Expr -> Expr -> String -> Int -> [Int] -> [Int] ->
             (Node,[Int],[Int])
```

Die ersten vier Argumente sind der zu übersetzende Ausdruck, der hervorzuhebende Redex, die Farbe des Knotens und die Zeilennummer, in der der Knoten positioniert werden soll. Mit den beiden Listen wird jeder Zeile `i` der Wert `next_col[i]` beziehungsweise `modifier[i]` zugewiesen. Initial gilt `next_col[i]=1` und `modifier[i]=0` für alle `i`. Diese Listen werden zur Berechnung der Spaltennummern benötigt: `next_col[i]` gibt die nächste freie Spalte in Zeile `i` an, `modifier[i]` steht für das Maximum aller Modifizierer der Knoten in Zeile `i`. Es wird schließlich ein Knoten und zwei aktualisierte Listen zurückgegeben.

Die Übersetzung von Ausdrücken in Knoten wird nun für zwei Fälle diskutiert. Ein Literal wird wie folgt in ein Blatt übersetzt:

```
expr2node (Lit l) redex color row nextcols mods =
  (newnode,setcol row nextcols ((nextcols!!row)+mindiff),mods)
  where newnode=makeleaf (show (Lit l)) color row cols mods

makeleaf label color row nextcols mods =
  Node label [] color row (nextcols!!row) (mods!!row)
```

Es wird ein neuer Knoten ohne Kinder mit der Beschriftung `(show (Lit l))`, der Farbe `color` und der Zeilennummer `row` zurückgegeben. Dieser wird mit der Funktion `makelaef` bestimmt. Die Spaltennummer und der Modifizierer ergeben sich aus den entsprechenden Listen. Da in Zeile `row` nun eine neue Spalte belegt wurde, muß `nextcols` mit der Funktion `setcol` an dieser Stelle um den Mindestabstand `mindiff` erhöht werden. Der Modifizierer für diese Zeile bleibt unverändert. Variablen werden auf die gleiche Art und Weise übersetzt.

Ein `Apply`-Ausdruck wird wie folgt behandelt:

```
expr2node (Apply e1 e2) re color row ncols mods | re==Apply e1 e2 =
 buildtree "@" [e1,e2] redex "red" row ncols mods
                                                | otherwise =
 buildtree "@" [e1,e2] redex color row ncols mods

buildtree label es redex color row ncols mods =
  let (ch,newcs,newms)=visitchildren es (row+1)
                                  (if (length ncols)<=(row+1)
                                    then (ncols++[1]) else ncols)
                                  (if (length ncols)<=(row+1)
                                    then (mods++[0]) else mods) []
      node=makenode label ch color row newcs newms
  in (node,setcol row newcs ((getcol node)+mindiff),
      updatemods newms node)
  where visitchildren [] _ cs ms nodes = (nodes,cs,ms)
        visitchildren (e:es) r cs ms nodes =
          let (node,cs2,ms2)=expr2node e redex color r cs ms
          in visitchildren es r cs2 ms2 (nodes++[node])

makenode label nodes color row ncols mods =
  Node label nodes color row (col+mod) mod
  where col=avg (map getcol nodes)
        mod=max (mods!!row) ((ncols!!row)-col)
```

Die Funktion `buildtree` erstellt einen Baum, der aus einem Knoten mit der Markierung `label` und der Farbe `color` besteht. Sie wird von der Funktion `expr2node` aufgerufen. Dabei wird als Markierung `"@"` genommen. Die Farbe des Knotens ist `"red"`, falls der zu übersetzende Ausdruck dem Redex entspricht, ansonsten wird die Farbe `color` des Vater-Knoten (beziehungsweise `"black"` für die Wurzel) genommen. Die Kinder des Knotens sind die Bäume zu den Ausdrücken in der Liste `es`. Diese werden zunächst mit `visitchildren` berechnet: Es ergibt sich eine Liste `ch` von Knoten und die beiden neuen Listen `newcs` und `newms`. Der Ergebnis-Knoten `node` wird mit der Funktion `makenode` berechnet: Die Spaltennummer ergibt sich aus dem arithmetischen Mittelwert der Spaltennummern der Kinder. Der Modifizierer berechnet sich aus dem Maximum des bisherigen maximalen Modifizierers für diese Zeile und dem Abstand zwischen der nächsten freien Spalte und der für den diesen Knoten berechneten Spalte `col`. Schließlich müssen noch die Listen `nextcols` und `mods` aktualisiert werden. Die nächste freie Spaltennummer in der aktuellen Zeile `row` ergibt sich aus der Spalte des neuen Knoten zuzüglich dem Mindestabstand. Mit der Funktion `setcol` wird der Wert in der Liste `ncols` entsprechend gesetzt. Mit der Funktion `updatemods` wird die Liste, die aus den Maxima der Modifizierer in einer Zeile besteht, entsprechend dem Modifizierer des neuen Knotens aktualisiert. Alle Ausdrücke, die noch nicht betrachtet wurden, werden nach dem gleichen Prinzip übersetzt.

Die Funktion `node2citems` addiert zu der Spaltennumer jedes Knotens die Summe der Modifizierer ihrer Vorgänger (die Knoten auf dem Weg zur Wurzel) auf und erzeugt dann die entsprechenden Tk-Elemente, also eine Liste von Linien und Knoten-Bezeichnern. Sie hat folgenden Typ:

```
node2citems :: Int -> Node -> [TkCanvasItem]
```

Dabei ist das erste Argument die Summe der Spaltenmodifizierer der Vorgänger des Knotens, der durch das zweite Argument gegeben ist. Spalten- und Zeilennummern werden durch Multiplikation mit den Konstanten `colsize` beziehungsweise `rowsize` in die entsprechenden Koordinaten umgerechnet.

`expr2citems` ruft lediglich nacheinander `expr2node` und `node2citems` auf:

```
expr2citems expr redex =
  node2citems 0 (fst3 (expr2node expr redex "black" 0 [1] [0]))
```

Von dem Ergebnis von `expr2node` interessiert nur die erste Komponente. Die Summe der Spaltenmodifizierer ist am Anfang `0`.

8.3. Beispiele

Bsp. 8.3.1 *Das folgende Curry-Programm implementiert den Quicksort-Algorithmus:*

```
qsort :: [Int] -> [Int]
qsort []     = []
qsort (x:l)  = qsort (filter (gt x) l) ++
               x : qsort (filter (leq x) l)

gt x y = x>y
leq x y = x<=y

goal = qsort [5,0,3,1,2,4,8,6,9,7]
```

Wenn man `filter` *als Haltepunkt setzt, wird die Auswertung stets bei Reduktion dieser Funktion gestoppt. Nachdem der erste Haltepunkt erreicht wurde ergibt sich das Bild in Abbildung 8.1*

Bsp. 8.3.2 *Im folgenden Programm werden zwei Threads gestartet: Ein Konto-Server*

(Objekt) und ein Client. Das Konto besteht aus einem Zustand (der vorhandene Betrag) und versteht die Nachrichten `Deposit` *(Einzahlen),* `Withdraw` *(Abheben) und* `Balance` *(Abfragen). Ist der Kontostand größer als 50, hebt der Client einen Betrag von 30 ab, ist er kleiner, zahlt er 70 ein. Beträgt der Kontostand 50, terminieren Client und Server. Nachrichten werden nicht über Ports verschickt, sondern durch Binden freier Variablen: Die Funktion* `sendMsg` *bindet das erste Argument an das zweite und gibt eine neue freie Variable zurück.*

```
data Message = Deposit Int | Withdraw Int | Balance Int

account :: Int -> [Message] -> Success
account eval rigid  -- account is a consumer
account _ []                =  success
account n (Deposit  a : ms)  =  account (n+a) ms
account n (Withdraw a : ms)  =  account (n-a) ms
account n (Balance  b : ms)  =  b=:=n & account n ms

-- create bank account
make_account s = account 0 s

-- send a message:
sendMsg msg obj | obj =:= msg:obj1 = obj1  where obj1 free

-- client process for bank account:
client s | s1 =:= sendMsg (Balance b) s =
  if b==50 then s1=:=[]   -- stop process
           else if b>50 then client (sendMsg (Withdraw 30) s1)
                        else client (sendMsg (Deposit  70) s1)
  where s1,b free

start s = make_account s & client (sendMsg (Deposit 100) s)

goal = let x free in start x
```

In Abbildung 8.2 sind die beiden Threads deutlich zu erkennen. Auf dem Konto befindet sich der Betrag 0 und über die Variable `v0` *wird die erste Nachricht* `(Deposit 100)` *versendet. Nach einigen Reduktionsschritten ergibt sich die Situation in Abbildung 8.3. Auf dem Konto befindet sich nun der Betrag 100, die nächste Nachricht wird über* `v8` *versendet.*

8.4. Integration in die Programmierumgebung

Der Debugger soll aus der Programmierumgebung heraus gestartet werden können. Hierzu gibt es im Tools-Menü den Eintrag “Debug”. Der Einfachheit halber wird immer die 0-stellige Funktion `goal` debugged. Um die Auswertung eines bestimmten Ausdrucks zu verfolgen, muß also eventuell noch diese Funktion geeignet definiert werden. Einerseits stellt dies keine große Einschränkung dar, andererseits würde eine bessere Lösung die Implementierung eines Parsers erfordern und somit um einiges aufwendiger sein.

Um den Debugger zu starten, ist das Öffnen eines neuen Fensters notwendig. Hiezu muß ein zweiter Tcl/Tk-Interpreter gestartet werden, was nur mit einer IO-Aktion getan werden kann. Aus diesem Grunde muß der Debugger vom IO-Server gestartet werden. Die Funktion

```
startDebugger :: Prog -> Expr -> IO ()
```

startet sowohl die Debug-GUI als auch den Debug-Server.

8.5. Alternativen

In Kapitel 6 ist bereits deutlich geworden, dass ein einfacher Tracer für eine deklarative Sprache, wie er in den vorhergehenden Abschnitten beschrieben wurde, in vielen Fällen nur sehr schlecht geeignet ist, den Programmierer bei der Fehlersuche zu unterstützen. Meistens werden die einzelnen Ausdrücke sehr groß und unübersichtlich, die Auswertungsreihenfolge entspricht oft nicht der Intuition des Programmierers. Trotzdem kann der vorgestellte Debugger nützlich sein: Er sorgt für einen besseren Überblick über nebenläufige Berechnungen und visualisiert die Kommunikation mittels freier Variablen. Anfängern wird das Verständnis der operationalen Semantik erleichtert.

Andere Debugger können nun jedoch relativ einfach auf der Basis des Meta-Evaluators implementiert werden. Das Konzept der Redex-Trails kann für Curry auf kanonische Art und Weise erweitert werden. Ein Redex-Trail kann während der Auswertung eines Ausdrucks mit dem Meta-Evaluator nach und nach aufgebaut werden. Dazu muß lediglich der Evaluator-Zustand um eine entsprechende Datenstruktur erweitert werden, diese muß dann bei jeder Reduktion aktualisiert werden. Algorithmische Debugger können ebenfalls auf Basis des Evaluators implementiert werden.

Allerdings haben diese Ansätze ebenfalls Nachteile: Mit Redex-Trail lassen sich Berechnungen immer nur rückwärts verfolgen. Der Fehler kann aber schon am Anfang der Berechnung sein, in diesem Falle ist es nicht einfach, dem Redex-Trail, der dann aus fehlerhaften Ausdrücken besteht, zu verfolgen. Der Nachteil algorithmischer Debugger ist hauptsächlich die zu große Anzahl der gestellten Fragen. Es ist also sinnvoll, mehrere verschiedene Debugger zur Verfügung zu stellen, der Evaluator bietet hierfür eine gemeinsame Grundlage.

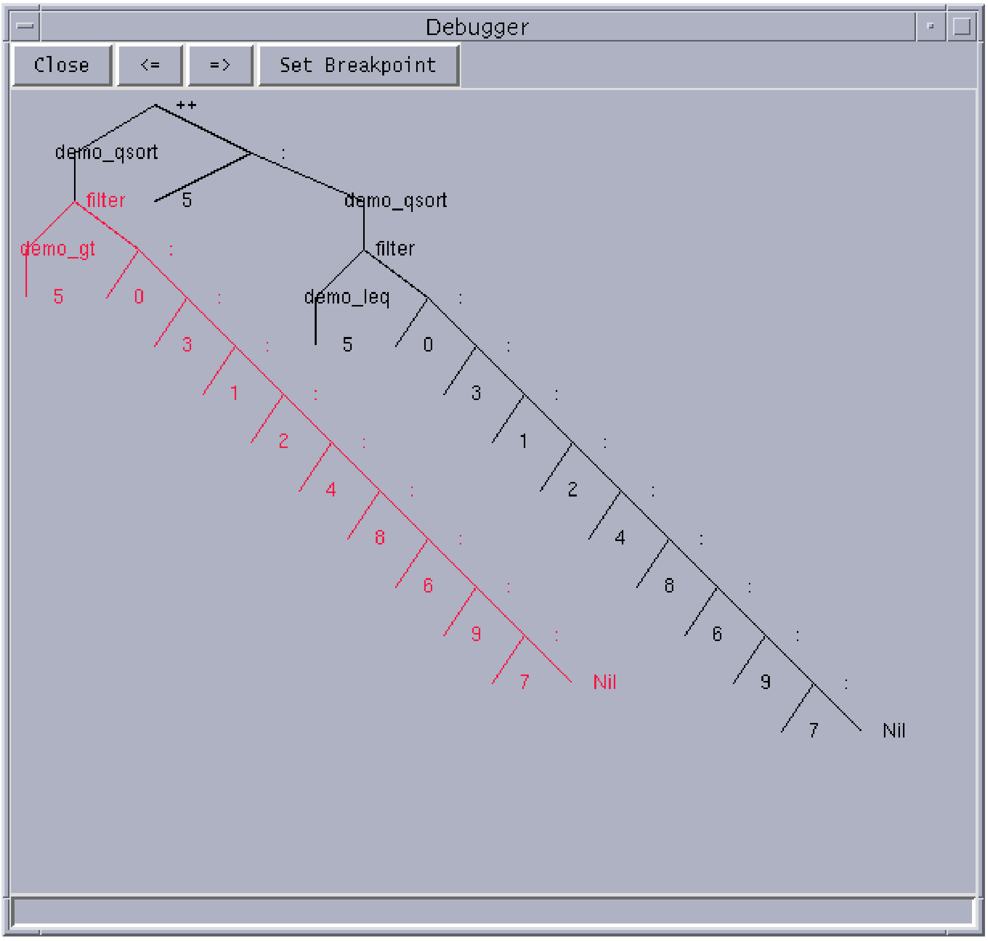

Abbildung 8.1.: Auswertung von quicksort

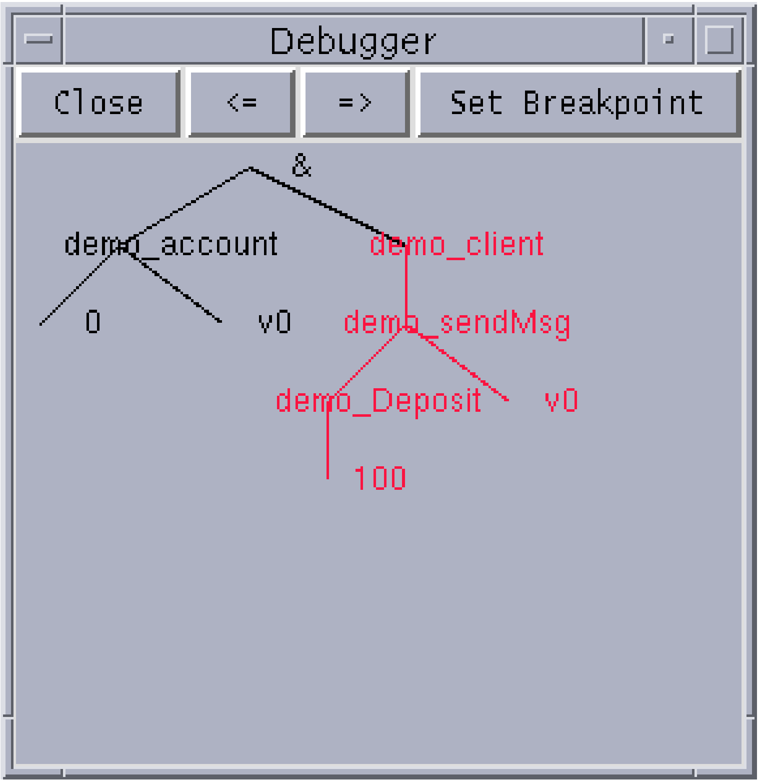

Abbildung 8.2.: Konto-Server 1

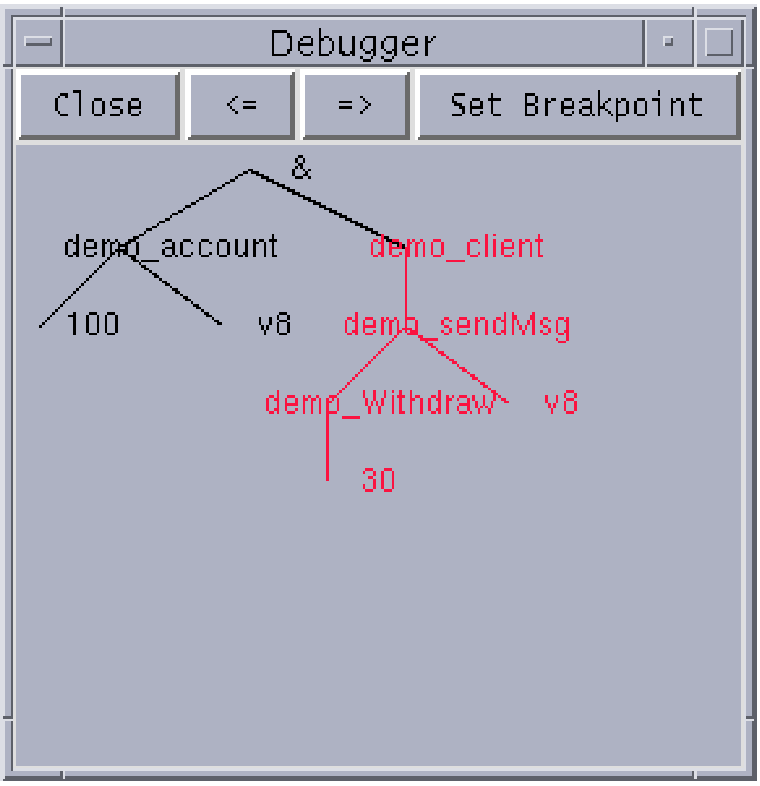

Abbildung 8.3.: Konto-Server 2

9. Eine Bibliothek zur Graph-Visualisierung

Graphen sind eine Datenstruktur, die auf vielen verschiedenen Gebieten Anwendung finden. Sie werden überall da benutzt, wo bestimmte Objekte und ihre Beziehungen dargestellt werden müssen: Netzwerke, Stadtpläne, Hierarchien usw. Da Graphen im Allgemeinen sehr groß werden können, ist deren Visualisierung relativ aufwendig. Deshalb gibt es Systeme, die das Layout von Graphen automatisch berechnen und ausgeben können.

In diesem Abschnitt wird gezeigt, wie auf Basis des DaVinci-Systems eine Bibliothek für Curry implementiert werden kann, die Graph-Visualisierung in einer komfortablen Art und Weise unterstützt. Diese kann dann für die Programmierumgebung verwendet werden, um beispielsweise Abhängigkeitsgraphen darzustellen.

9.1. DaVinci

DaVinci ist ein interaktives Werkzeug zur Graph-Visualisierung, welches seit 1992 an der Universität Bremen entwickelt wird. Es beschränkt sich lediglich auf die Darstellung von Graphen, und hat selbst keine direkte Möglichkeit zur Manipulation dieser Graphen. Dafür wird eine Schnittstelle zur Verfügung gestellt, über die andere Applikation den dargestellten Graph manipulieren können. Somit wird eine klare Trennung zwischen Kontrolle und Visualisierung erreicht.

Nach dem Start des DaVinci-Systems öffnet sich ein Fenster, in dem die Graphen angezeigt werden sollen. Es besteht aus einem Zeichenbereich und einer Menüleiste, über die die Darstellung der Graphen beeinflußt werden kann. Applikationen, die in einer beliebigen Sprache implementiert sein können, können sich dann mit diesem verbinden. Ein neuer Graph kann nun durch Versenden entsprechender Kommandos über Unix-Pipes im DaVinci-Fenster ausgegeben werden. Falls in diesem Events ausgelöst werden (zum Beispiel durch Maus-Klicks) wird die angeschlossene Applikation ebenfalls über eine Pipe informiert.

Die Verbindung zwichen DaVinci und der Applikation kann auf zwei verschiedene Weisen aufgebaut werden: Die erste Möglichkeit ist, dass DaVinci die Applikation startet und die Verbindung etabliert. Dies ist die einfachere Möglichkeit. Die andere Möglichkeit ist es, dass die Applikation den DaVinci-Prozess startet. Dies ist die etwas flexiblere, aber aufwendigere Alternative: Der Programmierer muß einen neuen Prozess kreieren und zwei Pipes für die Kommunikation einrichten.

Wenn die Verbindung steht, kann die Applikation Kommandos versenden, die die Darstellung oder Manipulation eines Graphen bewirken. Jedes Kommando wird mit dem String `ok` bestätigt. Die Kommandos lassen sich in verschiedene Kategorien einteilen, von denen die wichtigsten im Folgenden kurz vorgestellt werden:

- Mit *Graph-Kommandos* können Graph-Komponenten (Knoten, Kanten) hinzugefügt, manipuliert oder entfernt werden. Dazu müssen Knoten und Kanten in eine spezielle Darstellung gebracht werden.
- *Multi-Kommandos* ermöglichen die gleichzeiteige Darstellung mehrerer Graphen.
- Mit *Menü-Kommandos* können die Funktionen der Menüs aufgerufen werden, die Teil des DaVinci-Fensters sind.
- Mit *Set-Kommandos* können gewisse Optionen festgelegt werden (zum Beispiel Schriftgrößen oder andere Layout-Optionen).
- Mit *Fenster-Kommandos* kann das DaVinci-Fenster manipuliert werden.

Die genaue Syntax der Kommandos kann in [Tec] nachgelesen werden.

DaVinci kennt unter anderem folgende Events, über die die Applikation informiert wird:

- Einfaches und doppeltes Anklicken von Knoten
- Einfaches und doppeltes Anklicken von Kanten
- Auswahl von Menü-Punkten
- Öffnen neuer Fenster
- Schließen von Fenstern
- Wechseln zwischen verschiedenen Fenstern

9.2. Anforderungen an die Bibliothek

Es soll eine Bibliothek entwickelt werden, die eine komfortable Benuztung von DaVinci erlaubt. Der Aufbau der Verbindung mit dem DaVinci-System soll automatisch erfolgen. Die komplizierte Syntax der Kommondo-Syntax soll versteckt werden, stattdessen wird dem Programmierer eine einfache Schnittstelle zu Verfügung gestellt. Die Interpretation der Events, die vom DaVinci-System empfangen werden, und die Reaktion auf diese sollen dem Programmierer ebenfalls abgenommen werden: Er kann zu jedem möglichen Event einen Event-Handler definieren, der dann bei Eintreten dieses Events ausgeführt wird. Um Referenzen auf einzelne Objekte zu ermöglichen, muß jedem Objekt ein eindeutiger Identifier zugeordnet werden, der aus einem String besteht. Der Programmierer soll jedoch logische Variablen anstatt Strings als Referenzen benutzen. Hierdurch wird die Programmiersicherheit erheblich erhöht: Referenzen auf nicht vorhandene Objekte können schon zur Übersetzungszeit entdeckt werden.

9.3. Die Schnittstelle

Die Schnittstelle der DaVinci-Bibliothek bietet drei verschiedene Klassen von Funktionen an: Funktionen zur Ausgabe von Graphen, zur Definition von Graphen und zur Manipulation von Graphen.

Mit dem Aufruf `(dvDisplay g)` wird ein Graph `g` angezeigt. Ein Graph besteht aus einer Liste von Knoten und kann mit der Funktion

```
dvNewGraph :: [DvNode] -> DvGraph
```

erzeugt werden. Einfache Knoten ohne Kanten werden mit

```
dvSimpleNode :: DvId -> String ->
                (Port DvScheduleMsg -> Success) -> DvNode
```

erzeugt. Dabei ist das erste Argument der Identifier (der automatisch gebunden wird), das zweite eine Beschriftung für den Knoten und das dritte ein Event-Handler, der bei Anklicken des Knotens aufgerufen wird. Event-Handler haben den Typ

```
Port DvScheduleMsg -> Success
```

Sie werden bei Eintreten des entsprechenden Events auf einen bestimmten Port appliziert, über den mit dem Scheduler (siehe unten) kommuniziert wird. Dieser wird in den Manipulations-Funktionen verwendet. Knoten mit Kanten werden mit

```
dvNodeWithEdges :: DvId -> String -> [DvEdge] ->
                   (Port DvScheduleMsg -> Success) -> DvNode
```

erzeugt. Die Liste von Kanten entspricht einer Adjazenz-Liste. Der Erschaffung von Kanten dient die Funktion

```
dvSimpleEdge :: DvId -> dvNode ->
                (Port DvScheduleMsg -> Success) -> DvEdge.
```

Kanten bestehen aus einem Identifier, einem Knoten und einem Event-Handler.

Die Manipulations-Funktionen können innerhalb der Event-Handler benutzt werden: Mit den Funktionen

```
dvAddNode :: DvId -> String -> (Port DvScheduleMsg -> Success) ->
             (Port DvSchedulerMsg) -> Success
dvAddEdge :: DvId -> DvId -> DvId ->
             (Port DvScheduleMsg -> Success) ->
             (Port DvSchedulerMsg) -> Succes
```

können Knoten beziehungsweise Kanten hinzugefügt werden. Dabei ist das letzte Argument jeweils der Scheduler-Port, auf den ein Event-Handler bei Eintreten des Events appliziert wird. Event-Handler können mit

```
dvSetClickHandler :: DvId -> (Port DvScheduleMsg -> Success) ->
                     (Port DvSechedulerMsg) -> Success
```

verändert werden. Schließlich kann die Farbe eines Knoten mit

```
dvSetNodeColor :: DvId -> String -> (Port DvSchedulerMsg)
```

geändert werden. Es wird zwar nur ein kleiner Teil von DaVinci ausgenutzt, eine Erweiterung der Schnittstelle ist jedoch leicht möglich.

9.4. Implementierung

Bei einem Aufruf von `dvDisplay` muß zunächst das DaVinci-System gestartet werden und die Verbindung zu diesem hergestellt werden. Dies läßt sich recht einfach mit der vordefinierten Funktion

```
openProcessPort :: String -> IO (Port ppMsg)
```

realisieren. Über den so erhaltenen Port kann nun mit DaVinvi kommuniziert werden: Um das Kommando `cmd` zu verschicken, wird die Nachricht `(ppPut cmd)` an den Port gesendet. Das Vesenden von `(ppGet ans)` bewirkt die Bindung der freien Variablen `ans` an die nächste Antwort vom DaVinci-System. Nach dem Start von DaVinci wird die Graph-Definition in die entsprechenden DaVinci-Kommandos übersetzt und durch Versendung dieser angezeigt. Um die danach auftretenden Events abzufangen und entsprechend darauf zu reagieren, wird ein *Scheduler* gestartet.

Der Scheduler kennt alle Event-Handler und hält einen Index zur Vermeidung von Namenskonflikten. Er muß Nachrichten von zwei verschiedenen Seiten empfangen: Einerseits empfängt er die Events vom DaVinci-System, um den entsprechenden Handler zu bestimmen und auszuführen. Andererseits muß er auch Nachrichten von den Manipulations-Funktionen (die er selbst aufruft) empfangen: Diese können Kommandos nicht direkt an das DaVinci-System senden, sondern nur über den Scheduler. Wenn Objekte hinzugefügt werden sollen, muß ein neuer eindeutiger Identifier gefunden werden, dies kann der Scheduler mit seinem Index tun. Auch das Ändern von Event-Handlern geht nur über den Scheduler. Somit empfängt dieser drei verschiedene Arten von Nachrichten, was im Datentyp `DvScheduleMsg` zum Ausdruck kommt:

```
data DvScheduleMsg = DvAnswer String |          --DaVinci answers
                     DvCmdMsg DvCmd |           --DaVinci commands
                     DvSetEvent String DvEventH --new event
```

Kommandos werden intern mit dem Datentyp `DvCmd` dargestellt, Event-Handler mit `DvEventH`. Eine interne Darstellung von Kommandos ist zwar nicht unbedingt notwendig, erleichtert jedoch die Implementierung der Schnittstellen-Funktionen und die Erweiterung um neue Kommandos. Die Implementierung der Schnittstellen-Funktionen ist somit recht einfach: Kommandos zur Definition neuer Graphen erzeugen lediglich das entsprechende interne Kommando. Dieses wird dann von der Funktion `dvDisplay` in ein DaVinci-Kommando übersetzt und versendet. Manipulationsfunktionen versenden das entsprechende interne Kommando an den Scheduler. Falls der Scheduler die Nachricht `DvSetEvent` empfängt, so muß er lediglich seine Liste mit den Event-Handlern aktualisieren. Empfängt er die Nachricht `(DvCmdMsg cmd)`, so muß das interne Kommando `cmd` in ein DaVinci-Kommando übersetzt werden und an den DaVinci-Port versendet werden. Falls er die Nachricht `(DvAnswer s)` vom DaVinci-System empfängt, so muß die Antwort `s` zunächst interpretiert werden. Eventuell muß der passende Event-Handler bestimmt werden und auf den Scheduler-Port appliziert werden.

Damit der Scheduler sowohl Nachrichten über interne als auch über externe Ports empfangen kann, ist ein *Adapter* notwendig: Dieser holt sich die Nachrichten vom (externen) DaVinci-Port und reicht sie leicht abgewandelt an den Scheduler weiter:

```
dvAdaptor sport dvport | send (ppGet ans) dvport = --receive answer
  send (DvAnswer ans) sport &>              --send it to scheduler
  if ans=="quit" then success else dvAdaptor sport dvport
  where ans free
```

Der Adapter muß nebenläufig zum Scheduler laufen (siehe Abbildung 9.1). Seine Notwendigkeit ergibt sich aus der anderen Empfangstechnik bei externen Ports.

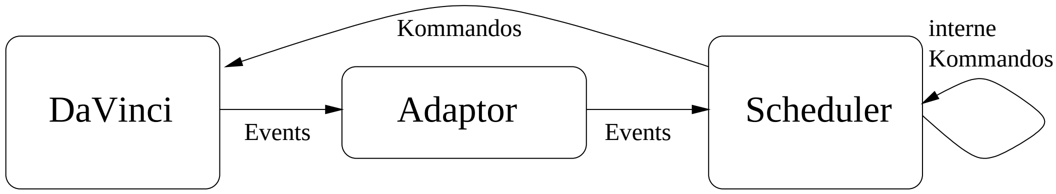

Abbildung 9.1.: Adaptor und Scheduler

9.5. Beispiele

Bsp. 9.5.1 *Das folgende Programm gibt zunächst einen Graphen aus, der lediglich aus einem Knoten besteht. Durch Anklicken dieses Knotens werden eine neuer Knoten und eine Kante von diesem neuen Knoten zum alten hinzugefügt:*

```
--import DaVinci

testGraph = dvNewGraph [dvSimpleNode node1 "Knoten1" nodeH]
  where node1 free
        nodeH sport = let node2,edge free
                      in dvAddNode node2 "Knoten" dvEmptyH sport &>
                         dvAddEdge edge node2 node1 sport

test = dvDisplay testGraph
```

Die Ausgabe des Programms zeigt Abbildung 9.2.

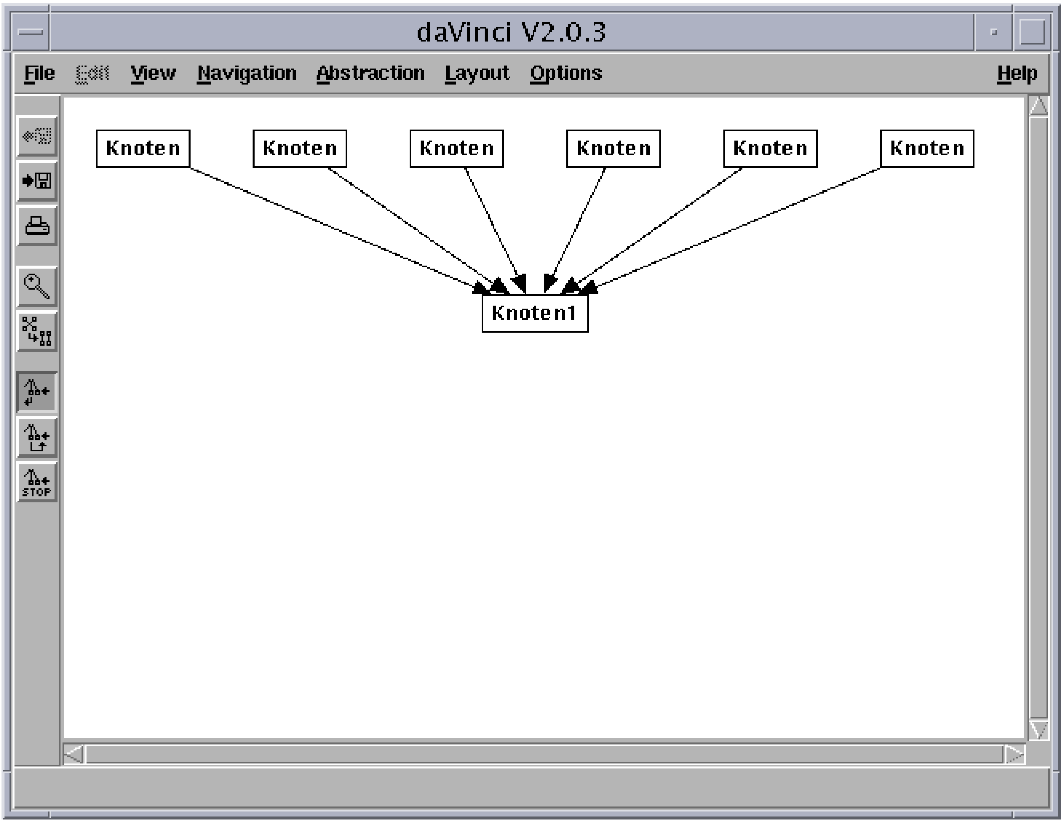

Abbildung 9.2.: Ausgabe des Programms aus Beispiel 9.5.1

Bsp. 9.5.2 *Der folgende Graph besteht aus zwei Knoten und einer verbindenden Kante. Durch Anklicken des ersten Knotens wird dieser rot gefärbt und der Event-Handler ausgetauscht. Somit wird er beim zweiten Anklicken blau gefärbt und so weiter. Anklicken der Kante bewirkt das Hinzufügen beziehungsweise Entfernen einer zweiten Kante.*

```
testGraph = dvNewGraph [dvNodeWithEdges node1 "Knoten1"
                                        [dvSimpleEdgeTo edge1
                                                        node2
                                                        eventH3]
                                        eventH1,
                        dvSimpleNode node2 "Knoten2" dvEmptyH]
  where node1,node2,edge1 free
        eventH1 sport = dvSetNodeColor node1 "red" sport &>
                        dvSetClickHandler node1 eventH2 sport
        eventH2 sport = dvSetNodeColor node1 "blue" sport &>
                        dvSetClickHandler node1 eventH1 sport
        eventH3 sport = let edge2 free
                        in dvAddEdge edge2 node1 node2 sport &>
                           dvSetClickHandler edge1 (eventH4 edge2)
                                             sport
        eventH4 edge sport = dvDelEdge edge sport &>
                             dvSetClickHandler edge1 eventH3 sport
test = dvDisplay testGraph
```

10. Zusammenfassung

Deklarative Programmiersprachen erleichtern die Programmerstellung und sorgen für weniger fehleranfällige und leichter wartbare Programme. Sie abstrahieren vom von Neumann-Rechner, somit sind ihre Konzepte leichter zu verstehen und zu beherrschen als die imperativer Sprachen. Damit diese Sprachen auch in der Praxis eingesetzt werden, müssen geeignete Programmierwerkzeuge vorhanden sein, die den Programmierer hauptsächlich bei der Fehlersuche unterstützen. In dieser Arbeit wurde ein erstes solches Werkzeug für die deklarative Programmiersprache Curry erstellt.

Die graphische Programmierumgebung integriert verschiedene Werkzeuge. Zum Einen kann sich der Benutzer mit dem Tracer einen Überblick über den Ablauf der Berechnungen verschaffen. Zum Anderen können beliebige Analyse-Module in einfacher Art und Weise in die Umgebung integriert werden. Diese dienen meistens dazu, Fehler einer bestimmten Klasse zu finden. Sie liefern als Ergebnis entweder einen Ergebnisstring oder einen Graphen, der dann mit der DaVinci-Bibliothek angezeigt wird. Es wurden einige nützliche Analysen implementiert, neue Analysen können jederzeit hinzugefügt werden. Nicht zuletzt ergibt sich ein Nutzen aus der Kombination des Tracers und verschiedener Analysen, was durch die Umgebung stark vereinfacht wird.

Die graphische Programmierumgebung wurde in Curry implementiert, was eine Reihe von Vorteilen mit sich brachte: Die Verwendung einer deklarativen Sprache hat natürlich die bereits erwähnten Vorteile. Weiterhin bietet die PAKCS-Ditribution komfortable Bibliotheken für graphische Benutzerschnittstellen und Meta-Programmierung an. Insbesondere die GUI-Bibliothek nutzt viele Eigenschaften von Curry aus um die Programmiersicherheit zu erhöhen und die schnelle Entwicklung gut stukturierter Programme zu ermöglichen. Allerdings hat die Verwendung einer rein deklarativen Sprache auch Nachteile: Es ist nicht möglich, festzustellen, ob eine Variable frei oder gebunden ist. Diese Information kan jedoch manchmal sehr nützlich sein. Zum Beispiel könnte sie benutzt werden, um mehrfache Verwendungen der selben Referenz bei der Benutzung der GUI-Bibliothek zu vermeiden.

Es wurden verschiedene Ansätze zur Implementierung eines Meta-Evaluators diskutiert. Auch wenn letztlich keine wirklich effiziente Implementierung erreicht wurde, kann der implementierte Evaluator trotzdem die Basis für diverse Werkzeuge sein. Mit dem Tracer wurde ein erstes solches Werkzeug geschaffen, es sind jedoch auch noch andere denkbar, die auf dem Evaluator aufbauen können.

Mit der DaVinci-Bibliothek wurde eine Bibliothek zur Visualisierung von Graphen erstellt, die nach ähnlichen Prinzipien wie die GUI-Bibliothek aufgebaut ist. Auch hier hat die Verwendung von Curry aufgrund des einfachen Umgangs mit externen Ports Vorteile. Bei der Implementierung der DaVinci-Bibliothek wurde das Konzept, auf dem die GUI-Bibliothek basiert, erweitert. Somit ist es beispielsweise auch möglich, Event-Handler zur

Laufzeit zu ändern. Diese Erweiterungen können prinzipiell auch auf die GUI-Bibliothek übertragen werden.

Auch wenn PAKCS noch einige Defizite (fehlendes Modulsystem, schlechte Laufzeit) hat, ist es die Erstellung größerer Applikationen in Curry ohne weiteres möglich. In Zukunft wird dies durch bessere Compiler und (hoffentlich) durch die vorgestellte graphische Programmierumgebung noch komfortabler möglich sein.

A. Benutzung der graphischen Programmierumgebung

Um die graphische Programmierumgebung benuzten zu können, muß eine Installation der PAKCS-Distribution, des Tcl/Tk- und des DaVinci-Systems vorhanden sein. vorhanden sein. Um die Umgebung zu starten, muß erst der IO-Server und dann die graphische Benutzerschnittstelle und der GUI-Server gestartet werden. Dazu muß PAKCS in zwei verschiedenen Terminal-Fenstern gestartet werden. In dem einen wird durch die Befehle

```
prelude> :l IOServer
           ⋮
IOServer> startIOServer
```

der IO-Server gestartet. Jetzt kann im anderen Fenster die eigentliche Umgebung gestartet werden:

```
prelude> :l Main
           ⋮
Main> start
```

Damit PAKCS gefunden werden kann, muß im Modul `IOServer` die Konstante `pakcshome` entsprechend gesetzt werden. In der DaVinci-Bibliothek muß die Konstante `dvHome` noch auf das Heimatverzeichnis des DaVinci-Systems gesetzt werden, damit die Umgebungsvariable `DAVINCIHOME` richtig gesetzt werden kann.

B. Dateiliste

Die graphische Programmierumgebung besteht aus folgenden Dateien:

```
Main.curry -- das Hauptprogramm
Gui.curry  -- die graphische Benutzerschnittstelle
GUIServer.curry -- der Gui-Server
IOServer.curry -- der IO-Server
Evaluator.curry -- der Meta-Evaluator
Debugger.curry -- der Debugger
Conf.curry -- das Konfigurations-Modul
Types.curry -- enthält von den Analyse-Modulen
               benötigte Typdeklarationen
TopFunAnalysis.curry -- Analyse zur Bestimmung der
                        globalen Funktionen
TypeAnalysis.curry -- Typ-Analyse
CompletenessAnalysis.curry -- Vollständigkeits-Analyse
CalledByAnalysis.curry -- Aufruf-Analyse
Dependency-Analysis.curry -- Abhängigkeits-Analyse
check.curry -- enthält Anpassungen zur Integration
               der Überlappungs-Analyse
DaVinci.curry -- DaVinci-Bibliothek
daVinciTest.curry -- Test-Graphen für die DaVinci-Bibliothek
XTk.curry -- kleinere Erweiterungen für die GUI-Bibliothek
EvaluatorBT.curry -- Der Evaluator mit Bindungs-Tabellen
RBTArray.curry -- Implementierung von Rot-Schwarz-Bäumen
```

Literaturverzeichnis

[AEH97] S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages. In *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 138–152. MIT Press, 1997.

[FR95] S. Foubister and C. Runciman. *Techniques for Symplifying the Visualization of Graph Reduction*, pages 61–65. Functional Programming. Springer, 1995.

[Han87] M. Hanus. *Problemlösen mit Prolog.* Teubner, 1987.

[Han00] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages*, pages 47–62. Springer LNCS 1753, 2000.

[He00a] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2000.

[He00b] M Hanus (ed.). PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs`, 2000.

[Oka98] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[Ous94] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison Wesley, 1994.

[Pop98] B. Pope. *Buddha, A Declarative Debugger for Haskell.* PhD thesis, University of Melbourne, 1998.

[Pü99] E. Pühringer. Eine integrierte Entwicklungsumgebung fur das JDK von SUN. Master's thesis, Johannes Kepler Universität Linz, 1999.

[Ree94] H.J. Reekie. Visual Haskell: A First Attempt. Research Report 94.5, Key Centre for Advanced Computing Science, University of Technology, Sydney, 1994.

[Sha82] E.Y. Shapiro. *Algorithmic Program Debugging.* 1982.

[SR97] J. Sparud and C. Runciman. Tracing Lazy Functional Computations using Redex Trails. In *Symposium on Programming Language Implementation and Logic Programming*, pages 291–308. Springer LNCS 1292, 1997.

[Tec] Technologie-Zentrum Informatik. DaVinci Online Documentation. Available at `http://www.tzi.de/daVinci/docs/referenceF.html`.

[Tie99] M. Tiedt. Kiel Interactive Evaluation Laboratory: Eine Arbeitsumgebung zur Visualisierung von Termersetzungssemantik. Master's thesis, Christian-Albrechts-Universität zu Kiel, 1999.