Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Implementierung von Rewriting-Bibliotheken in Curry

Jan-Hendrik Matthes

29. September 2016

Betreut durch Prof. Dr. Michael Hanus
und Dipl.-Inf. Jan Rasmus Tikovsky

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum | Unterschrift

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Die Programmiersprache Curry . . . 3
2.1.1 Programme . . . 3
2.1.2 Datentypen und Typsynonyme . . . 4
2.1.3 Funktionen und lokale Deklarationen . . . 5
2.1.4 Rechnen mit freien Variablen . . . 6
2.2 Reduktionssysteme . . . 7
2.3 Termersetzungssysteme . . . 8
2.4 Kritische Paare . . . 11
2.5 Reduktionsstrategien . . . 12
2.6 Definierende Bäume . . . 13
2.7 Narrowing . . . 15

3 Implementierung **20**
3.1 Terme und Gleichungen . . . 20
3.2 Substitution . . . 25
3.3 Unifikation . . . 26
3.4 Positionen . . . 26
3.5 Regeln und Termersetzungssysteme . . . 28
3.6 Kritische Paare . . . 29
3.7 Reduktion von Termen . . . 30
3.8 Definierende Bäume . . . 33
3.9 Narrowing . . . 35
3.10 Transformation von Curry-Programmen . . . 37

4 Fazit **39**
4.1 Zusammenfassung und Bewertung . . . 39
4.2 Ausblick . . . 40

A Übersicht der Module **41**
A.1 Rewriting.CriticalPairs . . . 41
A.2 Rewriting.DefinitionalTree . . . 41
A.3 Rewriting.Files . . . 43
A.4 Rewriting.Narrowing . . . 44
A.5 Rewriting.Position . . . 47

A.6 Rewriting.Rules 48
A.7 Rewriting.Strategy 50
A.8 Rewriting.Substitution 52
A.9 Rewriting.Term 53
A.10 Rewriting.Unification 55
A.11 Rewriting.UnificationSpec 55

Literaturverzeichnis **57**

1 Einleitung

Das Rechnen mit Termen und das Aufstellen von Gleichungen, also das Formulieren von Aussagen über die Gleichheit zweier Terme, sind ein elementarer Bestandteil der Mathematik. Mit derartigen Aussagen werden jedoch nicht nur einzelne Terme in Beziehung zueinander gesetzt, sondern Terme können damit auch umgeformt beziehungsweise vereinfacht (reduziert) werden. Betrachten wir dazu als Beispiel zwei allgemein bekannte Funktionen, die durch die folgenden Gleichungen definiert sind:

$$\begin{aligned} f(x) &= x^2 \\ g(y) &= 2y \end{aligned}$$

Eine Gleichung kann auf einen Term angewendet werden, wenn entweder die linke oder die rechte Seite innerhalb des betrachteten Terms enthalten ist. Dieser sogenannte Teilterm wird dann durch die entsprechende andere Seite der Gleichung ersetzt. Enthält eine Gleichung freie Variablen (wie etwa x oder y in der Definition von f und g), so werden die Variablen zunächst mit den passenden Werten des Teilterms belegt, bevor ein Ersetzungsschritt stattfindet. Mit den obigen Funktionsdefinitionen ist es also nun möglich, $f(g(f(2)))$ in eine natürliche Zahl zu überführen:

$$\begin{aligned} f(g(f(2))) &= f(g(4)) && \mid \text{nach Definition von } f \text{ mit } x = 2 \\ &= f(8) && \mid \text{nach Definition von } g \text{ mit } y = 4 \\ &= 64 && \mid \text{nach Definition von } f \text{ mit } x = 8 \end{aligned}$$

Das Umformen von einzelnen Termen ist jedoch nicht das einzige Einsatzgebiet. Vielmehr können auch Termgleichungen mit freien Variablen verarbeitet und mögliche Belegungen der Variablen berechnet werden. Der Grundgedanke liegt dabei im Allgemeinen in der Theorie, Gleiches durch Gleiches zu ersetzen.

Dieses Prinzip findet jedoch nicht nur in der reinen Mathematik seine Anwendung, sondern auch in der Informatik. So werden Gleichungen etwa im Bereich der funktionalen Programmierung für die Definition von Funktionen genutzt. Hier ist insbesondere die Programmiersprache Curry [Han16a] als Vergleich heranzuziehen. Terme bestehen in Curry aus Datenkonstruktoren und der Anwendung von Funktionen auf Argumente, die wiederum Terme sind. Es stellt sich nun allerdings die Frage, wie derartige Terme umgeformt oder vereinfacht werden können. Die Lösung findet sich im Berechnungsmodell der *Termersetzungssysteme* (TES) aus dem Bereich der theoretischen Informatik. Ein Termersetzungssystem ist dabei eine Menge von Gleichungen (*Regeln*) und unterscheidet sich von Gleichungen im herkömmlichen Sinn im Wesentlichen nur in der Tatsache, dass jene Gleichungen lediglich von links nach rechts angewendet werden. Ein Ersetzungsschritt von 5^2 zu $f(5)$ ist also nicht gestattet.

Das Ziel dieser Arbeit ist es nun, Bibliotheken für das Rechnen mit Termen, Gleichungen und Termersetzungssystemen in der funktional-logischen Programmiersprache Curry zu entwickeln. Bestandteil der Implementierung sind insbesondere Funktionalitäten für das Umformen und Vereinfachen (*Rewriting*) von Termen anhand definierter Reduktionsstrategien und das Lösen von Gleichungen mittels *Narrowing* (Verengung des Lösungsraumes). Aber auch das Überführen von Curry-Programmen in Termersetzungssysteme wird umgesetzt. Als Compiler kommt KiCS2[1] zum Einsatz, mit dem bereits Bibliotheken für die *Substitution* und *Unifikation* von Termen bereitgestellt werden.

Die Arbeit ist dabei so aufgebaut, dass in Kapitel 2 zunächst eine Einführung in die Grundlagen der Programmiersprache Curry und die Theorie der Termersetzungssysteme erfolgt. In Kapitel 3 werden schließlich die Bibliotheken Schritt für Schritt gemäß der zuvor getroffenen Definitionen entwickelt.

[1] `https://www-ps.informatik.uni-kiel.de/kics2/`

2 Grundlagen

In diesem Kapitel werden die für die Entwicklung der Bibliotheken wichtigen und grundlegenden Konzepte vorgestellt. Neben der Programmiersprache Curry, auf die im ersten Abschnitt genauer eingegangen wird, sind das vor allem die formalen Grundlagen für Berechnungen in der funktionalen Programmierung, wie etwa Termersetzungssysteme, die Reduktion von Termen und kritische Paare, aber auch Teilaspekte der Logikprogrammierung. In das zuletzt genannte Themengebiet fallen insbesondere das Rechnen mit freien Variablen und *Narrowing* zur Lösung von Gleichungen.

Der Inhalt und der Aufbau der folgenden Abschnitte orientiert sich dabei im Wesentlichen am offiziellen Curry-Handbuch [Han16a] und an den Notizen zur Vorlesung *Deklarative Programmiersprachen* [Han16b], die zuletzt im Wintersemester 2015/2016 an der Christian-Albrechts-Universität zu Kiel angeboten wurde.

2.1 Die Programmiersprache Curry

Curry ist eine universelle Programmiersprache, die die wichtigsten beiden deklarativen Programmierparadigmen, also die funktionale Programmierung und die Logikprogrammierung, vereint. So werden die Aspekte der funktionalen Programmierung, wie etwa verschachtelte Ausdrücke, *Lazy Evaluation* und Funktionen höherer Ordnung, mit den Aspekten der Logikprogrammierung, wie etwa logische Variablen, partielle Datenstrukturen und Suchstrategien zum Auffinden von Lösungen, auf sinnvolle und effiziente Weise kombiniert. Die Entwicklung von Curry ist dabei eine internationale Initiative mit dem Ziel, eine gemeinsame Plattform für die Erforschung, Lehre und Anwendung von integrierten funktional-logischen Programmiersprachen bereitzustellen [Han16a].

Curry ist syntaktisch weitgehend durch die funktionale Programmiersprache Haskell[1] inspiriert, sodass ein Curry-Programm einem Haskell-Programm sehr ähnlich ist und in den meisten Fällen synonym verwendet werden kann. Aufgrund der großen Ähnlichkeit werden deshalb für ein besseres Verständnis dieses Abschnittes grundlegende Kenntnisse über die Programmiersprache Haskell vorausgesetzt.

2.1.1 Programme

Die Auswertung eines Curry-Programmes erfolgt durch die Vereinfachung eines Ausdruckes hin zu einem Wert. Da jedoch zwischen einfachen Werten und reduzierbaren Ausdrücken unterschieden werden muss, existieren in Curry sogenannte *Datenkonstruktoren* und *Operationen* (auch *definierte Funktionen*) auf diesen Daten. Ein Curry-Programm

[1] `https://www.haskell.org/`

besteht damit im Wesentlichen aus einer Menge von Typ- und Funktionsdeklarationen. Dabei definieren die Typdeklarationen die Wertebereiche (Konstruktoren) und die Funktionsdeklarationen die Operationen auf diesen Wertebereichen.

Wie auch andere moderne funktionale Programmiersprachen ist Curry eine streng getypte Sprache mit einem polymorphen Typsystem, sodass Programmierfehler bereits zur Übersetzungszeit erkannt werden. Jedes Objekt innerhalb eines Programmes besitzt somit einen eindeutigen Typ, der jedoch dank Typinferenz für Variablen und Funktionen nicht explizit angegeben werden muss.

2.1.2 Datentypen und Typsynonyme

Ein neuer Datentyp kann in Curry analog zu Haskell in der Form

$$\texttt{data}\ T\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ C_1\ \tau_{11} \ldots \tau_{1n_1}\ \texttt{|} \cdots \texttt{|}\ C_k\ \tau_{k1} \ldots \tau_{kn_k}$$

deklariert werden. Damit einhergehend werden ein n-stelliger Typkonstruktor T sowie k neue Datenkonstruktoren $C_1, \ldots, C_k$ eingeführt, wobei jeder Datenkonstruktor C_i mit $i \in \{1, \ldots, k\}$ den Typ

$$\tau_{i1}\ \texttt{->}\ \cdots\ \texttt{->}\ \tau_{in_i}\ \texttt{->}\ T\ \alpha_1 \ldots \alpha_n$$

und die Stelligkeit n_i aufweist. Datenkonstruktoren mit einer Stelligkeit von Null werden auch als *Konstanten* bezeichnet. Ein τ_{ij} ist ein Typausdruck, der aus Typkonstruktoren und den Typvariablen $\alpha_1, \ldots, \alpha_n$ zusammengesetzt ist.

In Curry sind bereits einige Datentypen vordefiniert. Dazu gehört auch ein Datentyp für Wahrheitswerte, der wie folgt definiert ist:

```
data Bool = False | True
```

Damit wird also der Datentyp `Bool` mit den beiden 0-stelligen Datenkonstruktoren `False` und `True` eingeführt. Ein nicht-leerer Binärbaum mit einem beliebigen Typ für den Wert der Blätter und Knoten kann mittels

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

deklariert werden. Ein Binärbaum des Typs `Tree a` entspricht somit entweder einem Blatt (`Leaf`) mit einem Wert vom Typ a oder einem Knoten (`Node`). Ein Knoten setzt sich gemäß Definition aus einem linken Binärbaum des Typs `Tree a`, einem Wert des Typs a und einem rechten Binärbaum des Typs `Tree a` zusammen. Da a ein beliebiger Typ ist, kann ein Binärbaum mit ganzen Zahlen als Beschriftung einfach mit dem Typ `Tree Int` angegeben werden.

Um die Lesbarkeit von Typausdrücken zu verbessern, besteht in Curry die Möglichkeit, bestehende Typausdrücke mit einem neuen (passenderen) Namen auszustatten. Diese sogenannten *Typsynonyme* können gemäß der allgemeinen Form

$$\texttt{type}\ T\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ \tau$$

eingeführt werden. Dabei ist T ein n-stelliger Typkonstruktor und $\alpha_1, \ldots, \alpha_n$ paarweise verschiedene Typvariablen. Der Typausdruck τ ist aus bestehenden Typkonstruktoren und den Typvariablen $\alpha_1, \ldots, \alpha_n$ zusammengesetzt. Insgesamt bedeutet die Definition des obigen Typsynonyms also, dass ein Typausdruck $T\ \tau_1 \ldots \tau_n$ genau dann äquivalent zu τ ist, wenn für jedes $i \in \{1, \ldots, n\}$ alle Vorkommen der Typvariable α_i in τ durch τ_i ersetzt werden.

Ein Typsynonym für einen Binärbaum mit ganzen Zahlen als Beschriftung für die Blätter und Knoten entspricht demnach der folgenden Deklaration:

```
type IntTree = Tree Int
```

Die Typausdrücke `IntTree` und `Tree Int` können dabei zu jeder Zeit durch den jeweils anderen ersetzt werden, ohne dass sich die Semantik eines Curry-Programmes ändert.

2.1.3 Funktionen und lokale Deklarationen

In Curry besteht eine Funktion aus zwei wichtigen Teilen. Dies ist zum einen die Typdeklaration, die jedoch dank Typinferenz weggelassen werden kann, und zum anderen eine Liste von *definierenden Gleichungen* (auch *Regeln* genannt). Eine Typdeklaration hat die Form

$$f \ :: \ \tau_1 \ \texttt{->} \ \tau_2 \ \texttt{->} \ \cdots \ \texttt{->} \ \tau_n \ \texttt{->} \ \tau$$

Dabei ist f der Name beziehungsweise der Bezeichner der deklarierten Funktion und $\tau_1, \ldots, \tau_n, \tau$ sind Typausdrücke. Eine Gleichung für eine n-stellige Funktion ist in der einfachsten Form nach dem Muster

$$f \ t_1 \ldots t_n \ = \ e$$

aufgebaut, wobei $t_1, \ldots, t_n$ Datenterme (auch *Patterns* genannt) sind und e ein Ausdruck ist. Ein Pattern besteht dabei nur aus Variablen und der Anwendung von Konstruktoren auf Datenterme. Eine Funktion kann aber nicht nur durch eine einzige Gleichung definiert werden, sondern auch aus mehreren Gleichungen bestehen, die jedoch alle die gleiche Anzahl an Argumenten aufweisen müssen. Dies erlaubt insbesondere nichtdeterministische Funktionen zu definieren, indem linke Seiten von Regeln sich überlappen. Eine solche Funktion hat dann mehrere Rückgabewerte für eine Eingabe.

Eine Funktion `sumTree`, die die Summe aller Werte eines Binärbaumes von ganzen Zahlen berechnet, kann beispielsweise wie folgt definiert werden:

```
sumTree :: IntTree -> Int
sumTree (Leaf v)     = v
sumTree (Node l v r) = (sumTree l) + v + (sumTree r)
```

Die beiden Gleichungen können aber auch zu einer einzigen zusammengefasst werden. Hierzu können die aus Haskell bekannten `case`-Ausdrücke genutzt werden. Somit ergibt sich folgende Funktionsdeklaration:

```
sumTree :: IntTree -> Int
sumTree t
  = case t of
      (Leaf v)     -> v
      (Node l v r) -> (sumTree l) + v + (sumTree r)
```

Für Funktionen können aber auch *bedingte Gleichungen* angegeben werden. Diese können gemäß der Form

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

definiert werden, wobei $c_1, \ldots, c_n$ Bedingungen (auch als *Guards* bezeichnet) sind. Bei der Anwendung der Funktion wird für die Gleichung überprüft, ob eine der Bedingungen, beginnend von c_1, zu `True` ausgewertet werden kann. Wurde ein solches $i \in \{1, \ldots, n\}$ gefunden, dann ist entsprechend e_i die passende rechte Seite der Gleichung.

In bestimmten Situationen kommt es vor, dass beispielsweise eine Funktion nur innerhalb eines bestimmten Bereiches sichtbar sein soll. In Curry ist es deshalb möglich, Funktionen und Variablen lokal zu deklarieren. Ähnlich zu Haskell kann dafür ein `let`-Ausdruck oder eine `where`-Klausel genutzt werden. Ein `let`-Ausdruck hat dabei die allgemeine Form `let` *decls* `in` *exp*. Der folgende Ausdruck

```
let x = 3 * y
    y = 7
 in 2 * x
```

wird so dementsprechend zu 42 ausgewertet. Die in `where`-Klauseln definierten Bezeichner sind innerhalb der rechten Seite einer Gleichung verfügbar. Zur Demonstration dient eine Funktion `fib`, die wie folgt definiert ist:

```
fib :: Int -> Int
fib n = fibs !! n
  where
    fibs = fibgen 0 1
    fibgen a b = a:(fibgen b (a + b))
```

Die Funktion berechnet dabei die n-te Fibonacci-Zahl mithilfe der beiden lokalen Funktionen `fibs` und `fibgen`.

2.1.4 Rechnen mit freien Variablen

Da Curry nicht nur eine funktionale sondern auch eine logische Programmiersprache ist, sind innerhalb von Ausdrücken auch sogenannte *freie* Variablen erlaubt. Bei der Auswertung eines solchen Ausdruckes, wird versucht eine Belegung für jede freie Variable

zu finden, damit der Ausdruck zu einem Term reduziert werden kann. Im Gegensatz zu Prolog, wo insbesondere freie Variablen im globalen Namensraum implizit eingeführt sind, müssen in Curry aufgrund der verschiedenen Geltungsbereiche freie Variablen einmalig explizit mit dem Schlüsselwort `free` gekennzeichnet werden. Für ein Beispiel betrachten wir zunächst folgende Definition:

```
data Person = Andreas | Maria | Markus | Michael | Sarah

mother :: Person -> Person
mother Andreas = Sarah
mother Markus  = Maria
mother Michael = Maria
```

Um ein Kind von `Maria` zu berechnen, kann sich nun dem Ausdruck

```
let x free
 in (mother x) == Maria
```

bedient werden. Die freie Variable x wird mit `Markus` belegt, um die linke Seite der Gleichung zu `Maria` zu reduzieren. Folglich kann die resultierende Gleichung zu `True` ausgewertet werden. Wir wissen also, dass `Markus` ein Kind von `Maria` ist. Aufgrund der nichtdeterministischen Auswertung, insbesondere aber wegen der Narrowing-Strategie von Curry, erhalten wir zusätzlich auch `Michael` als Lösung der Gleichung.

Das nächste Beispiel zeigt, wie freie Variablen genutzt werden können, um das letzte Element einer Liste zu berechnen:

```
last :: [a] -> a
last xs | ys ++ [y] == xs = y
  where
    y, ys free
```

Bei einem Aufruf der Funktion mit `last [1, 2, 3]` wird ys an die Liste `[1, 2]` und y an die Zahl 3 gebunden. Die Bedingung kann somit zu `True` reduziert werden, sodass folglich y und damit die Zahl 3 als Lösung zurückgegeben wird.

2.2 Reduktionssysteme

Das Rechnen in funktionalen Programmiersprachen kann durch das fortlaufende Anwenden von Reduktionsschritten beschrieben werden. Um das Modell der Reduktionen und die später darauf aufbauenden Begrifflichkeiten besser zu verstehen, treffen wir zunächst einige allgemeine Definitionen.

Definition 2.1 (Reduktionssystem) *Sei M eine Menge und $\rightarrow$ eine zweistellige Relation auf M. Wir schreiben $e_1 \rightarrow e_2$, falls $(e_1, e_2) \in \rightarrow$. Dann heißt $(M, \rightarrow)$* ***Reduktionssystem****. Darauf aufbauend definieren wir folgende Erweiterungen der Relation $\rightarrow$:*

- $x \to^0 y \;:\Leftrightarrow\; x = y$
- $x \to^i y \;:\Leftrightarrow\; \exists z\colon x \to^{i-1} z$ *und* $z \to y$
- $x \to^* y \;:\Leftrightarrow\; \exists i \geq 0\colon x \to^i y$

Für die Reduktion legen wir nun mit einem Reduktionssystem weitere Begriffe fest.

Definition 2.2 *Sei* $(M, \to)$ *ein Reduktionssystem.*

- $x \in M$ *heißt* ***reduzierbar*** *genau dann, wenn ein* y *mit* $x \to y$ *existiert.*
- $x \in M$ *heißt* ***irreduzibel*** *oder in* ***Normalform****, falls* x *nicht reduzierbar ist.*
- y *ist eine* ***Normalform von*** x *genau dann, wenn* $x \to^* y$ *und* y *Normalform.*
- $\to$ *heißt* ***konfluent****, falls* $\forall u, x, y$ *mit* $u \to^* x$ *und* $u \to^* y$ *ein* z *mit* $x \to^* z$ *und* $y \to^* z$ *existiert.*
- $\to$ *heißt* ***terminierend****, falls keine unendlichen Ketten* $x_1 \to x_2 \to \dots$ *existieren.*

2.3 Termersetzungssysteme

Das Berechnungsmodell der Termersetzungssysteme bildet die Grundlage für das Rechnen mit Funktionen. Ein Termersetzungssystem ist dabei eine Menge von Gleichungen (*Regeln*), die nur von links nach rechts angewendet werden. Für eine genaue formale Definition legen wir zunächst einige grundlegende Begrifflichkeiten fest:

Definition 2.3 (Grundbegriffe für Termersetzungssysteme)

- $\Sigma = (S, F)$ *heißt* ***Signatur****, wobei*
  - S *eine Menge von* ***Sorten*** *(Nat, Int, Bool, String, …) ist und*
  - F *eine Menge von* ***Funktionssymbolen*** $f :: s_1, \dots, s_n \to s$ *mit* $n \geq 0$ *ist. (*$f :: \to s$ *heißt auch* ***Konstante****)*
- $V = \{x :: s \mid x \text{ ist Variablensymbol und } s \in S\}$ *ist eine Menge von* ***Variablen****, wobei jede Variable nur eine Sorte haben darf.*
- *Die Menge* $T(\Sigma, V)_s$ *aller* ***Terme*** *über* Σ *und* V *der Sorte* s *ist wie folgt definiert:*
  - $x \in T(\Sigma, V)_s$ *falls* $x :: s \in V$
  - $f(t_1, \dots, t_n) \in T(\Sigma, V)_s$ *falls* $f :: s_1, \dots, s_n \to s \in F$ *und* $t_i \in T(\Sigma, V)_{s_i}$ *für alle* $i \in \{1, \dots, n\}$
- *Die Menge aller Variablen in einem Term ist wie folgt definiert:*

$$\mathcal{V}ar(x) = \{x\}$$
$$\mathcal{V}ar(f(t_1, \dots, t_n)) = \mathcal{V}ar(t_1) \cup \dots \cup \mathcal{V}ar(t_n)$$

- *Ein **Termersetzungssystem** (TES) ist eine Menge von **Regeln** der Form $l \to r$ mit $l, r \in T(\Sigma, V)_s$ und $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Dabei wird l als **linke Seite** und r als **rechte Seite** der Regel bezeichnet.*
- *Ein Term t heißt **Grundterm**, falls $\mathcal{V}ar(t) = \varnothing$.*
- *Ein Term t heißt **linear**, falls keine Variable in t mehrfach vorkommt.*

Für ein Beispiel betrachten wir ein Termersetzungssystem, das eine Oder-Verknüpfung auf Wahrheitswerten definiert. Die Signatur dieser Menge von Regeln ist gegeben durch $\Sigma = (\{\texttt{Bool}\}, \{False :: \to \texttt{Bool}, True :: \to \texttt{Bool}, or :: \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\})$ und die Menge der Variablen durch $V = \{x :: \texttt{Bool}\}$. Die Regeln sind die folgenden:

$$or(False, x) \to x \tag{1}$$

$$or(True, x) \to True \tag{2}$$

Mit der Hilfe dieser Regeln können wir nun Terme umformen und reduzieren, die ausschließlich aus Wahrheitswerten und Oder-Verknüpfungen bestehen. Die folgenden Reduktionen eines einzigen Terms zeigen jedoch, dass es noch einer präziseren Definition eines Reduktionsschrittes bedarf:

$$or(False, or(True, False)) \to or(True, False)$$
$$or(False, or(True, False)) \to or(False, True)$$

Wir sehen also, dass Terme auch an mehreren Stellen reduziert werden können, wobei der resultierende Term jeweils ein anderer ist. Dies führt uns unweigerlich zum Begriff der Position, den wir nun genauer definieren.

Definition 2.4 (Position)

- *Eine **Position** in einem Term bezeichnet einen Teilterm, ausgedrückt durch eine Folge von (natürlichen) Zahlen.*
- *Die Menge aller Positionen in einem Term t wird mit $\mathcal{P}os(t)$ bezeichnet und ist wie folgt definiert:*
  - $\epsilon \in \mathcal{P}os(t)$ *mit* ϵ *als leere Folge (Wurzelposition)*
  - $i \cdot p \in \mathcal{P}os(f(t_1, \ldots, t_n))$ *falls* $p \in \mathcal{P}os(t_i)$ *und* $i \in \{1, \ldots, n\}$
- *Für den Vergleich zweier Positionen p und q genügen die folgenden Begriffe:*
  - $p \leq q$ *(p **über** q), falls p ein Präfix von q ist.*
  - $p \geq q$ *(p **unter** q), falls q ein Präfix von p ist.*
  - *p und q sind **disjunkt**, falls weder $p \leq q$ noch $p \geq q$ gilt.*
  - *p ist **links** von q, falls $p = p_0 \cdot i \cdot p_1$ und $q = p_0 \cdot j \cdot q_1$ mit $i < j$.*
  - *p ist **rechts** von q, falls $p = p_0 \cdot i \cdot p_1$ und $q = p_0 \cdot j \cdot q_1$ mit $i > j$.*

Mit der Definition von Positionen ist es nun möglich, die unterschiedlichen Stellen, an denen in einem Term reduziert werden kann, zu unterscheiden. Im obigen Beispiel wurde so zunächst an der Position ϵ und dann an der Position 2 reduziert.

Für das Anwenden einer Regel an einer bestimmten Position müssen wir jedoch noch weitere Begrifflichkeiten klären. Dazu gehören etwa die Substitution von Variablen durch Terme und die Selektion beziehungsweise Ersetzung von Teiltermen.

Definition 2.5 (Teilterm, Ersetzung, Substitution, Unifikator)

- *Ein **Teilterm** von t an der Position $p \in \mathcal{P}os(t)$ wird mit $t|_p$ bezeichnet und ist mit $i \in \{1, \ldots, n\}$ wie folgt definiert:*

$$\begin{aligned} t|_\epsilon &= t \\ f(t_1, \ldots, t_n)|_{i \cdot p} &= t_i|_p \end{aligned}$$

- *Die **Ersetzung** des Teilterms an der Position p im Term t durch s wird mit $t[s]_p$ bezeichnet und ist mit $i \in \{1, \ldots, n\}$ definiert als:*

$$\begin{aligned} t[s]_\epsilon &= s \\ f(t_1, \ldots, t_n)[s]_{i \cdot p} &= f(t_1, \ldots, t_{i-1}, t_i[s]_p, t_{i+1}, \ldots, t_n) \end{aligned}$$

- *Eine **Substitution** σ ist eine Abbildung $\sigma \colon V \to T(\Sigma, V)_s$, wobei $\sigma(x) \in T(\Sigma, V)_s$ für alle $x :: s \in V$ gilt und der Definitionsbereich durch $\mathcal{D}om(\sigma) = \{x \mid \sigma(x) \neq x\}$ gegeben ist. Die Notation einer Substitution erfolgt in einer übersichtlichen Form mit $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Eine Erweiterung einer Substitution auf Terme ist mit der Abbildung $\sigma' \colon T(\Sigma, V)_s \to T(\Sigma, V)_s$ möglich. Diese wird durch die Gleichung $\sigma'(f(t_1, \ldots, t_n)) = f(\sigma'(t_1), \ldots, \sigma'(t_n))$ unter Verwendung von σ definiert.*

- *Eine Substitution σ heißt **Unifikator** für eine Gleichung der Form $l = r$, falls $\sigma(l) = \sigma(r)$ gilt.*

- *Ein Unifikator σ heißt **allgemeinster Unifikator (mgu)** für eine Gleichung $l = r$, falls für alle Unifikatoren σ' eine Substitution φ mit $\sigma' = \varphi \circ \sigma$ existiert. Genauer gesagt sind damit alle anderen Unifikatoren Spezialfälle von σ.*

Auf dieser Grundlage können wir nun die Reduktionsrelation bezüglich eines gewählten Termersetzungssystems definieren:

Definition 2.6 (Termersetzung) *Sei R ein Termersetzungssystem und t_1, t_2 Terme. Dann gilt für eine Reduktionsrelation $\to_R \subseteq T(\Sigma, V)_s \times T(\Sigma, V)_s$ genau dann $t_1 \to_R t_2$, wenn eine Regel $l \to r \in R$, eine Position $p \in \mathcal{P}os(t_1)$ sowie eine Substitution σ mit $t_1|_p = \sigma(l)$ und $t_2 = t_1[\sigma(r)]_p$ existieren. $t_1 \to_R t_2$ wird in diesem Fall auch **Reduktionsschritt** oder **Ersetzungsschritt** und $t_1|_p$ auch **Redex** genannt. Die Schreibweise eines Reduktionschrittes kann je nach Bedarf variieren. So kann etwa $t_1 \to_{p, l \to r} t_2$ bevorzugt werden, wenn die Position und die genutzte Regel von Bedeutung sind. Auch $t_1 \to t_2$ ist möglich, sollte das Termersetzungssystem fest gewählt sein.*

Betrachten wir dazu noch einmal das Beispiel der Oder-Verknüpfung von oben:

$$or(False, x) \to x \tag{1}$$
$$or(True, x) \to True \tag{2}$$

Jetzt ist es möglich, die Reduktion des Terms genau anzugeben. Die erste vollständige Reduktion ist somit:

$$\begin{aligned} or(False, or(True, False)) &\to_{\epsilon,(1)} or(True, False) \\ &\to_{\epsilon,(2)} True \end{aligned}$$

Die zweite Reduktion unterscheidet sich in der Auswahl der Positionen sowie der Reihenfolge der Regeln und lautet insgesamt wie folgt:

$$\begin{aligned} or(False, or(True, False)) &\to_{2,(2)} or(False, True) \\ &\to_{\epsilon,(1)} True \end{aligned}$$

Wir halten also fest, dass sich der betrachtete Term auf zwei unterschiedliche Weisen zur Normalform $True$ (die nicht weiter reduziert werden kann) reduzieren lässt.

2.4 Kritische Paare

Für funktionale Programme ist es sinnvoll, dass die Normalformen eines Terms eindeutig sind, insbesondere also ein Term keine verschiedenen Normalformen aufweist. Hierfür müssen lediglich sogenannte kritische Paare eines Termersetzungssystems bestimmt werden [KB70]. Kritische Paare bezeichnen dabei Überlappungen linker Regelseiten und sind im Folgenden genauer definiert:

Definition 2.7 (Kritische Paare) *Sei R ein Termersetzungssystem. Seien weiterhin $l_1 \to r_1, l_2 \to r_2 \in R$, wobei die Variablen in l_1 und l_2 paarweise verschieden sind, $p \in \mathcal{P}os(l_1)$ mit $l_1|_p \notin V$ und σ ein allgemeinster Unifikator für $l_1|_p$ und l_2. Falls $p = \epsilon$, dann sei auch $l_1 \to r_1$ keine Variante (gleich bis auf die Umbenennung von Variablen) von $l_2 \to r_2$. Dann wird $\langle \sigma(r_1), \sigma(l_1)[\sigma(r_2)]_p \rangle$ als* ***kritisches Paar*** *von $l_1 \to r_1$ und $l_2 \to r_2$ bezeichnet. Die Menge aller kritischen Paare aus R ist mit $\mathcal{CP}(R)$ gegeben.*

Mit dieser Definition können wir nun weitere sinnvolle Bedingungen an ein Termersetzungssystem formulieren:

Definition 2.8 *Sei R ein links-lineares Termersetzungssystem, insbesondere also jede linke Regelseite linear. Dann heißt R*

- ***orthogonal**, falls $\mathcal{CP}(R) = \varnothing$.*
- ***schwach orthogonal**, falls für alle $\langle t, t' \rangle \in \mathcal{CP}(R)$ die Gleichung $t = t'$ gilt.*

2.5 Reduktionsstrategien

Ein Term hat im idealen Fall eine eindeutige Normalform, sodass jeder Reduktionsschritt, egal in welcher Reihenfolge, angewendet werden kann. Mit bestimmten Termersetzungssystemen ist es jedoch möglich, dass die Abfolge der Reduktionsschritte eine Auswirkung auf den resultierenden Term und darauf hat, ob die Berechnung überhaupt terminiert. Wir betrachten dazu den Term $f(a, a, d)$ und das folgende Termersetzungssystem, wobei nur mit x und y Variablen bezeichnet werden:

$$a \to b \tag{1}$$

$$b \to c(c(a)) \tag{2}$$

$$f(x, b, y) \to d \tag{3}$$

Es ist nun festzustellen, dass der Term die Redex-Positionen 1 und 2 hat, da jeweils das Ersetzen von a durch b möglich ist. Eine Reduktion an der Position 1 kann nun beispielsweise wie folgt aussehen:

$$\begin{aligned} f(a, a, d) &\to_{1,(1)} f(b, a, d) \\ &\to_{1,(2)} f(c(c(a)), a, d) \\ &\to_{1\cdot 1\cdot 1,(1)} \cdots \end{aligned}$$

Eine andere Reduktion, die den Term zunächst an der Position 2 und dann an der Wurzelposition reduziert, lautet wie folgt:

$$\begin{aligned} f(a, a, d) &\to_{2,(1)} f(a, b, d) \\ &\to_{\epsilon,(3)} d \end{aligned}$$

Damit ist klar, dass die Reihenfolge der einzelnen Reduktionsschritte eine eindeutige Auswirkung auf die Berechnung einer Normalform hat. Wir benötigen also eine Strategie, die ebendiese Reihenfolge bestimmt. Dies führt uns zum Begriff der Reduktionsstrategie, den wir nun etwas genauer definieren wollen.

Definition 2.9 (Reduktionsstrategie) *Eine Abbildung $S\colon T(\Sigma, V)_s \to 2^{\mathcal{P}os(T(\Sigma,V)_s)}$ wird als* ***Reduktionsstrategie*** *bezeichnet, wobei $S(t) \subseteq \mathcal{P}os(t)$ gilt. Weiterhin ist $t|_p$ ein Redex für alle $p \in S(t)$ und für alle $p_1, p_2 \in S(t)$ mit $p_1 \neq p_2$ sind p_1 und p_2 disjunkt. Intuitiv definiert S also die Positionen innerhalb eines Terms, an denen reduziert wird.*

Für einige Beispiele von Reduktionsstrategien, nehmen wir im Folgenden an, dass ein Term $t \in T(\Sigma, V)_s$ nicht in Normalform ist.

Definition 2.10 *Eine Reduktionsstrategie S heißt*

- ***leftmost innermost*** *(LI), falls $S(t) = \{p\}$ und $t|_p$ Redex sowie für alle Positionen q mit $t|_q$ Redex, die verschieden zu p sind, p links oder unter q gilt.*
- ***leftmost outermost*** *(LO), falls $S(t) = \{p\}$ und $t|_p$ Redex sowie für alle Positionen q mit $t|_q$ Redex, die verschieden zu p sind, p links oder über q gilt.*

- ***rightmost innermost*** *(RI), falls* $S(t) = \{p\}$ *und* $t|_p$ *Redex sowie für alle Positionen* q *mit* $t|_q$ *Redex, die verschieden zu* p *sind,* p *rechts oder unter* q *gilt.*
- ***rightmost outermost*** *(RO), falls* $S(t) = \{p\}$ *und* $t|_p$ *Redex sowie für alle Positionen* q *mit* $t|_q$ *Redex, die verschieden zu* p *sind,* p *rechts oder über* q *gilt.*
- ***parallel innermost*** *(PI), falls*
  $S(t) = \{p \mid t|_p \text{ Redex und } \forall q \neq p \text{ mit } t|_q \text{ Redex ist } q \text{ nicht unter } p\}$ *gilt.*
- ***parallel outermost*** *(PO), falls*
  $S(t) = \{p \mid t|_p \text{ Redex und } \forall q \neq p \text{ mit } t|_q \text{ Redex ist } q \text{ nicht über } p\}$ *gilt.*

Betrachten wir nun erneut das obige Beispiel, so entspricht die erste Reduktion der *Leftmost Innermost* Strategie und die zweite Reduktion der *Rightmost Outermost* Strategie. Dabei ist die Strategie LI nicht normalisierend, wie an diesem Beispiel zu erkennen ist. RO berechnet hingegen die Normalform des Terms. Die Wahl der richtigen Strategie ist also essentiell für die Berechnung von Normalformen.

Wir wollen aufgrund dessen eine weitere Bedingung an ein Termersetzungssystem stellen, sodass bestimmte Reduktionsstrategien auf dieser Klasse normalisierend sind.

Definition 2.11 *Ein Termersetzungssystem* R *heißt* ***links-normal****, falls für alle Regeln* $l \to r \in R$ *in* l *hinter einer Variablen kein Funktionssymbol steht.*

2.6 Definierende Bäume

Die Klasse der links-normalen Termersetzungssysteme ist jedoch sehr eingeschränkt. Daher betrachten wir eine weitere wichtige Klasse von Termersetzungssystemen, für die zum Beispiel mit φ ebenfalls interessante Reduktionsstrategien existieren.

Definition 2.12 *Sei* R *ein Termersetzungssystem mit der Signatur* $\Sigma = (S, F)$*.*

- *Eine Funktion* $f\colon s_1, \ldots, s_n \to s \in F$ *heißt* ***definierte Funktion*** *bezüglich* R*, falls eine Regel* $f(t_1, \ldots, t_n) \to r$ *in* R *existiert.*
- *Die Menge aller definierten Funktionen bezüglich des Termersetzungssystems* R *ist definiert als* $D = \{f \mid f \text{ ist definierte Funktion}\} \subseteq F$*.*
- $C = F \setminus D$ *heißt Menge der* ***Konstruktoren****.*
- $f(t_1, \ldots, t_n)$ *wird als* ***Muster*** *bezeichnet, falls* $f \in D$ *und* $t_1, \ldots, t_n$ ***Konstruktorterme*** *sind, also keine definierten Funktionen enthalten.*
- R *heißt* ***konstruktorbasiert****, falls die linke Seite jeder Regel ein Muster ist.*

Definition 2.13 (Definierender Baum) *Ein* ***definierender Baum*** *ist ein Baum, dessen Knoten allesamt mit einem Muster markiert sind. Zu unterscheiden ist dabei zwischen zwei Arten von definierenden Bäumen mit dem Muster* π*:*

- *Regelknoten der Form* $l \to r$ *mit* $\pi = l$.
- *Verzweigungsknoten der Form* $branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)$. *Hierbei gilt:*
  - p *ist eine Position in* π *mit* $\pi|_p \in V$.
  - $\mathcal{T}_i$ *ist für alle* $i \in \{1, \ldots, k\}$ *ein definierender Baum mit dem Muster* $\pi[C_i(x_1, \ldots, x_{m_i})]_p$. *Dabei sind* $x_1, \ldots, x_{m_i}$ *neue Variablen und* $C_1, \ldots, C_k$ *verschiedene Konstruktoren.*

Das Muster eines definierenden Baumes $\mathcal{T}$ *bezeichnen wir mit* $pattern(\mathcal{T})$.

Definition 2.14 (Induktiv-sequenziell) *Sei* R *ein Termersetzungssystem.*

- $\mathcal{T}$ *heißt definierender Baum für die Funktion* f, *falls* $\mathcal{T}$ *endlich ist und das Muster* $f(x_1, \ldots, x_n)$ *hat, jeder Regelknoten eine Variante einer Regel aus* R *ist, und jede Regel* $f(t_1, \ldots, t_n) \to r \in R$ *in* $\mathcal{T}$ *genau einmal vorkommt. In diesem Fall heißt* f ***induktiv-sequenziell****.*
- R *heißt* ***induktiv-sequenziell****, falls alle* $f \in D$ *induktiv-sequenziell sind.*

Induktiv-sequenzielle Termersetzungssysteme haben dabei intuitiv die Eigenschaft, dass Funktionen induktiv über Datenstrukturen definiert sind. Für ein Beispiel betrachten wir die Additionsfunktionen auf Peano-Zahlen, die mit einer Fallunterscheidung über das erste Argument umgesetzt ist:

$$0 + y \to y \tag{1}$$

$$s(x) + y \to s(x + y) \tag{2}$$

Die anschauliche grafische Darstellung des zugehörigen definierenden Baumes für diese Funktion ist der Abbildung 2.1 zu entnehmen.

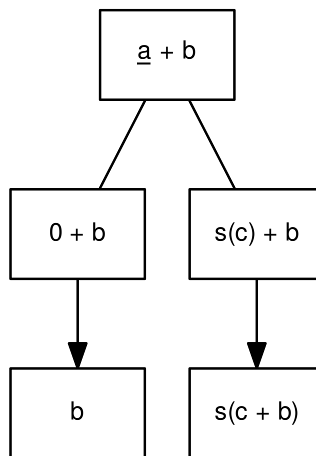

Abbildung 2.1: Definierender Baum der einfachen Peano-Additionsfunktion

Für induktiv-sequenzielle Termersetzungssysteme existiert mit φ eine einfache sequenzielle Reduktionsstrategie, die wir im Folgenden genauer betrachten wollen.

Definition 2.15 (Reduktionsstrategie φ) *Sei t ein Term, o die Position des linkesten äußersten definierten Funktionssymbols in t, also $t|_o = f(t_1, \ldots, t_n)$ mit $f \in D$, und $\mathcal{T}$ ein definierender Baum für f. Dann ist $\varphi(t) = \{o \cdot \varphi(t_o, \mathcal{T})\}$. Weiterhin ist:*

$$\varphi(t, l \to r) = \epsilon$$

$$\varphi(t, branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)) = \begin{cases} \varphi(t, \mathcal{T}_j) & \text{falls } t|_p = C_j(s_1, \ldots, s_{m_j}) \\ & \text{und } pattern(\mathcal{T}_j)|_p = C_j(x_1, \ldots, x_{m_j}). \\ p \cdot \varphi(t|_p, \mathcal{T}) & \text{falls } t|_p = f(\ldots) \text{ mit } f \in D \text{ und } \mathcal{T} \\ & \text{definierender Baum für } f. \end{cases}$$

Für ein Beispiel betrachten wir erneut das Termersetzungssystem für die Addition von Peano-Zahlen und den Term $t = (s(0) + 0) + 0$. Mit der Reduktionsstrategie φ können wir den Term dann nach dem folgenden Prinzip reduzieren:

$$\begin{aligned} (s(0) + 0) + 0 &\to_{1,(2)} s(0 + 0) + 0 && \text{da } \varphi(t) = 1 \\ &\to_{\epsilon,(2)} s((0 + 0) + 0) && \text{da } \varphi(t) = \epsilon \\ &\to_{1 \cdot 1,(1)} s(0 + 0) && \text{da } \varphi(t) = 1 \cdot 1 \\ &\to_{1,(1)} s(0) && \text{da } \varphi(t) = 1 \end{aligned}$$

Die Auswertung einer Funktion erfolgt dabei durch die Analyse des zugehörigen definierenden Baumes. Bei einem Regelknoten wird die enthaltene Regel angewendet und bei einem Verzweigungsknoten der Wert an der Verzweigungsposition betrachtet. Entspricht der Wert einem Konstruktor, so wird der passende Teilbaum gewählt. Bei einer Funktion wird diese entsprechend ausgewertet.

2.7 Narrowing

Terme bestehen im Allgemeinen nicht ausschließlich nur aus Konstruktoren sondern können auch Variablen enthalten. Mit den bereits vorgestellten Mitteln zur Reduktion von Termen ist es jedoch nicht möglich, derartige Konstrukte umzuformen beziehungsweise zu reduzieren. Wir betrachten dazu das folgende Termersetzungssystem:

$$\begin{aligned} f(0, x) &\to 0 \\ f(1, x) &\to x \end{aligned}$$

Mittels Reduktion können wir die Gleichung $f(a, 42) = 42$ nicht berechnen, obwohl eine Lösung mit $a = 1$ sofort ersichtlich ist. Wir benötigen also eine passende Belegung der freien Variablen, sodass der Ausdruck reduzierbar ist. Dabei ist einfaches Raten zu ineffizient, da es im Allgemeinen unendlich viele Lösungen gibt. Wir nutzen daher die Unifikation von Termen, was uns zum Begriff **Narrowing** führt.

Definition 2.16 (Narrowing-Schritt) *Seien t und t' Terme. Dann wird $t \leadsto_\sigma t'$ oder auch $t \leadsto_{p,l \to r,\sigma} t'$ als **Narrowing-Schritt** (bezüglich eines Termersetzungssystems R) bezeichnet, falls die folgenden Bedingungen gelten:*

- *p ist eine nichtvariable Position, also $t|_p \notin V$.*
- *$l \to r$ ist eine Variante einer Regel aus R.*
- *σ ist ein allgemeinster Unifikator für $t|_p$ und l, es gilt also $\sigma(t|_p) = \sigma(l)$.*
- *$t' = \sigma(t[r]_p)$*

Für das obige Beispiel ergeben sich nun die folgenden beiden Narrowing-Schritte:

$$f(a, 42) \rightsquigarrow_{\{a \mapsto 0\}} 0$$
$$f(a, 42) \rightsquigarrow_{\{a \mapsto 1\}} 42$$

Wir sehen also, dass mit der Belegung $a = 1$ die betrachtete Gleichung lösbar beziehungsweise gültig ist. Die Eigenschaft der Gültigkeit einer Gleichung ist wie folgt definiert:

Definition 2.17 *Eine Gleichung $s = t$ heißt **gültig**, falls s in t überführt werden kann. Eine Substitution σ heißt **Lösung** der Gleichung $s = t$, falls $\sigma(s) = \sigma(t)$ gültig ist.*

Insgesamt können wir festhalten, dass *Narrowing* allgemeine Repräsentanten aller möglichen Lösungen berechnet. Um diese Lösungen komplett zu ermitteln, ist das Berechnen aller möglichen Narrowing-Ableitungen (also das Raten von Position und Regel in jedem Schritt) notwendig. Da es jedoch sehr viele Narrowing-Ableitungen geben kann, ist eine Einschränkung und damit eine Verbesserung durch spezielle Strategien sinnvoll. Die Definition einer solchen Narrowing-Strategie übernehmen wir aus [AEH00]:

Definition 2.18 (Narrowing-Strategie) *Eine **Narrowing-Strategie** ist eine Funktion von Termen in eine Menge von Tripeln. Sei nun N eine Narrowing-Strategie, t ein Term und $(p, l \to r, \sigma) \in N(t)$, dann ist p eine Position in t, $l \to r$ eine Regel des Termersetzungssystems R und σ eine Substitution, sodass $t \rightsquigarrow_{p,l\to r,\sigma} \sigma(t[r]_p)$ ein Narrowing-Schritt ist.*

Wir betrachten zunächst strikte Narrowing-Strategien, die das einfache Ausprobieren an allen Positionen verbessern. Das Problem hierbei ist jedoch, dass die innerste und äußerste Position nicht immer eindeutig ist. Durch die Einschränkung auf konstruktorbasierte Termersetzungssysteme kann die fehlende Eindeutigkeit allerdings umgangen werden. Im Folgenden nehmen wir also an, dass das Termersetzungssystem R konstruktorbasiert ist. Wir können damit nun strikte Narrowing-Strategien formulieren:

Definition 2.19 *Eine Narrowing-Ableitung $t_0 \rightsquigarrow_{p_1,\sigma_1} t_1 \rightsquigarrow_{p_2,\sigma_2} \cdots \rightsquigarrow_{p_n,\sigma_n} t_n$ heißt **innermost** oder auch **strikt**, falls $t_{i-1}|_{p_i}$ für alle $i \in \{1, \ldots, n\}$ ein Muster ist.*

Innermost Narrowing entspricht der strikten Auswertung in funktionalen Sprachen, ist jedoch nur eingeschränkt vollständig. Zum einen sind die berechneten Lösungen eventuell zu speziell, zum anderen kann *Innermost Narrowing* bei partiellen Funktionen fehlschlagen. Ähnlich der Reduktionsstrategien wollen wir nun auch für das *Narrowing outermost* oder *lazy* Strategien vorstellen. Der Vorteil ist dabei, dass Teilterme nur dann ausgewertet werden, falls es notwendig ist. Die Eigenschaft *notwendig* ist dabei von der Strategie abhängig. Beginnen wir zunächst mit der Definition von zwei *outermost* Strategien.

Definition 2.20 *Ein Narrowing-Schritt $t \rightsquigarrow_{p,\sigma} t'$ heißt **outermost (leftmost outermost)**, falls für alle Narrowing-Schritte $t \rightsquigarrow_{p',\sigma'} t''$ mit $p \neq p'$ die Einschränkung $p' \not\leq p$ ($p' \not\leq p$ und p' nicht links von p) gilt.*

Im Allgemeinen ist *Outermost Narrowing* jedoch unvollständig. Es benötigt also weitere Einschränkungen an ein Termersetzungssystem (etwa konfluent und terminierend), damit diese Narrowing-Strategie vollständig ist. Wir stellen nun Strategien vor, bei denen die Forderung nach der Terminierung nicht benötigt wird. Vielmehr setzen wir voraus, dass ein Termersetzungssystem konstruktorbasiert und schwach orthogonal ist. Wir kürzen diese Eigenschaft auch mit **KB-SO** ab.

Definition 2.21 *Die Menge der **lazy** Positionen ist nach [MNRA92] für einen Term t mit $I = \{i \mid 1 \leq i \leq n,$ eine Regel für f benötigt das Argument an der Stelle $i\}$ wie folgt definiert:*

$$\mathcal{LP}(t) = \begin{cases} \varnothing & \text{falls } t \text{ eine Variable ist.} \\ \{i \cdot p \mid p \in \mathcal{LP}(t_i)\} & \text{falls } t = c(t_1, \ldots, t_n) \text{ und } c \in C. \\ \{\epsilon \mid \text{eine Regel ist mit } t \text{ unifizierbar}\} \cup & \\ \bigcup\limits_{i \in I} \{i \cdot p \mid p \in \mathcal{LP}(t_i)\} & \text{falls } t = f(t_1, \ldots, t_n) \text{ und } f \in D. \end{cases}$$

Definition 2.22 *Ein Narrowing-Schritt $t \rightsquigarrow_{p,\sigma} t'$ heißt **lazy**, falls $p \in \mathcal{LP}(t)$.*

Mit dieser Strategie können wir also Gleichungen mit unendlichen Strukturen lösen. Dagegen laufen hier alle strikten Strategien in eine Endlosschleife. Da es jedoch nicht nur eine *lazy* Position, sondern eventuell mehrere *lazy* Positionen in einem Term geben kann, ist es möglich, dass manche Narrowing-Schritte überflüssig bezüglich bestimmter Substitutionen sind. Somit sind zu spezielle Lösungen weiterhin möglich. *Lazy Narrowing* ist also nicht wirklich *lazy*.

Eine verbesserte Strategie, die dieses Problem umgeht, heißt **Needed Narrowing**. Die grundlegende Idee bei der Auswertung des Aufrufs $f(t_1, \ldots, t_n)$ ist dabei:

- Bestimme ein t_i, dessen Wert von allen Regeln für f verlangt wird.
- Falls t_i
  - ein Konstruktor ist, wähle die passende Regel mit diesem Konstruktor.
  - ein Funktionsaufruf ist, werte diesen aus.
  - eine Variable ist, binde diese nichtdeterministisch an die verschiedenen Konstruktoren und mache weiter.

Doch wie findet man die verlangten Argumente. Die Lösung sind hier definierende Bäume und induktiv-sequenzielle Termersetzungssysteme. Damit können wir also die *Needed Narrowing* Strategie wie folgt definieren:

Definition 2.23 *Sei s ein Term, o eine Position des linkesten äußersten definierten Funktionssymbols in s und $\mathcal{T}_f$ ein definierender Baum für f. Die* ***Needed Narrowing*** *Strategie λ berechnet eine Tripel-Menge der Form $(p, l \to r, \sigma)$, sodass $s \rightsquigarrow_{p,l\to r,\sigma} \sigma(s[r]_p)$ ein Narrowing-Schritt ist. Dabei ist λ definiert als:*

$$\lambda(s) = \{(o \cdot p, l \to r, \sigma) \mid (p, l \to r, \sigma) \in \lambda(s|_o, \mathcal{T}_f)\}$$

wobei $\lambda(t, \mathcal{T})$ die kleinste Menge von Tripeln ist:

$$\lambda(t,\mathcal{T}) \supseteq \begin{cases} \{(\epsilon, l \to r, mgu(t,l))\} & \text{falls } \mathcal{T} = l \to l. \\ \lambda(t, \mathcal{T}_i) & \text{falls } \mathcal{T} = branch(\pi, p, \mathcal{T}_1, \dots, \mathcal{T}_n) \\ & \text{und } t \text{ und } pattern(\mathcal{T}_i) \text{ unifizierbar.} \\ \{(p \cdot p', l \to r, \sigma \circ \tau)\} & \text{falls } \mathcal{T} = branch(\pi, p, \mathcal{T}_1, \dots, \mathcal{T}_n), \\ & t|_p = g(\dots) \text{ mit } g \in D,\ \tau = mgu(t, \pi) \text{ und} \\ & \mathcal{T}_g \text{ definierender Baum für } g \text{ und} \\ & (p', l \to r, \sigma) \in \lambda(\tau(t|_p), \mathcal{T}_g). \end{cases}$$

Needed Narrowing ist nur für induktiv-sequenzielle Termersetzungssysteme definiert. Dies bedeutet, dass insbesondere überlappende Regeln nicht erlaubt sind. Wir betrachten dazu das Termersetzungssystem für das *parallele* Oder:

$$\begin{aligned} True \lor x &\to True \\ x \lor True &\to True \\ False \lor False &\to False \end{aligned}$$

Das Problem ist hier, dass es kein eindeutiges Induktionsargument gibt. Das bedeutet, dass kein Argument *needed* ist und somit bei der Auswertung von $t_1 \lor t_2$ unklar ist, ob zunächst t_1 oder t_2 ausgewertet werden soll. Da derartige Termersetzungssysteme aber erlaubt sind und durchaus auch vorkommen, erweitern wir definierende Bäume durch einen zusätzlichen Oder-Knoten:

Definition 2.24 *Ein* ***erweiterter definierender Baum*** *ist ein definierender Baum, der aus den nachfolgenden Knoten bestehen kann, wobei jeder Knoten ein Muster π hat:*

- *Regelknoten wie in Definition 2.13 eingeführt.*
- *Verzweigungsknoten wie in Definition 2.13 eingeführt.*
- ***Oder-Knoten*** *der Form $or(\mathcal{T}_1, \dots, \mathcal{T}_n)$, wobei jedes $\mathcal{T}_i$ mit $i \in \{1, \dots, n\}$ ein erweiterter definierender Baum mit dem Muster π ist.*

Der erweiterte definierende Baum für die *parallele* Oder-Funktion, die wir oben eingeführt haben, ist in Abbildung 2.2 zu betrachten:

Die Erweiterung der *Needed Narrowing* Strategie bezüglich erweiterter definierender Bäume ist nicht besonders schwer. Als Ergebnis erhalten wir eine *lazy* Strategie für konstruktorbasierte und schwach orthogonale Termersetzungssysteme:

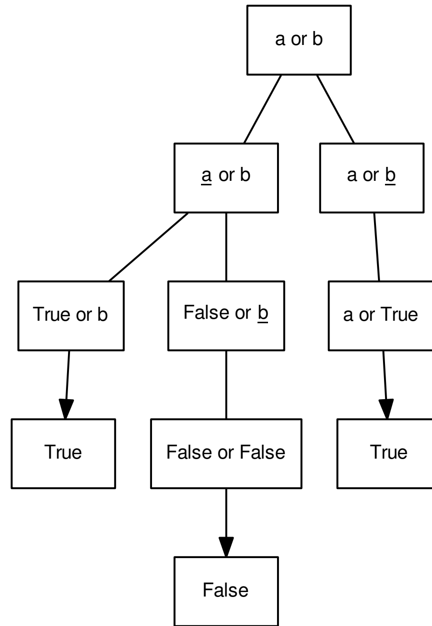

Abbildung 2.2: Erweiterter definierender Baum der parallelen Oder-Funktion

Definition 2.25 *Die Erweiterung von* Needed Narrowing *bezüglich erweiterten definierenden Bäumen heißt* ***Weakly Needed Narrowing*** *und ist wie* λ *definiert, jedoch um die folgende Erweiterung für Oder-Knoten ergänzt:*

$$\lambda(t, or(\mathcal{T}_1, \ldots, \mathcal{T}_k)) \supseteq \lambda(t, \mathcal{T}_i) \quad \text{für } i \in \{1, \ldots, k\}$$

Bei einem Oder-Knoten wird somit jede der möglichen Alternativen ausprobiert.

3 Implementierung

Im vorherigen Kapitel haben wir die Programmiersprache Curry und die Grundlagen der Termersetzungssysteme erläutert. Darauf aufbauend wollen wir nun Schritt für Schritt die Implementierung der einzelnen Module vorstellen. Für eine kompakte und vollständige Übersicht der Module sei hingegen auf Anhang A verwiesen.

3.1 Terme und Gleichungen

Terme bilden eine der Grundlagen für das Rechnen mit Termersetzungssystemen. Wir benötigen daher zunächst einen Datentyp für die Repräsentation von Termen. Hier orientieren wir uns an der Definition 2.3, nach der ein Term entweder eine Variable oder ein Konstruktor ist. Der Datentyp kann somit wie folgt definiert werden:

```
type VarIdx = Int

data Term f = TermVar VarIdx | TermCons f [Term f]
```

Der Typ `VarIdx` ist hier lediglich ein Synonym für den Typ `Int`, da wir Variablen einfach als ganze beziehungsweise natürliche Zahlen betrachten. Somit entspricht 0 der Variablen a und 23 der Variablen x. Des Weiteren nutzen wir Typvariablen (hier f), damit die Darstellung eines Konstruktors frei gewählt werden kann. Mit der getroffenen Definition können wir nun Terme in Curry definieren. Wir betrachten dazu den Term $x + y$. Diesen können wir darstellen als:

```
TermCons "+" [TermVar 23, TermVar 24]
```

Dabei hat der Term aufgrund des Konstruktors `"+"` den Typ `Term String`. Wir können aber auch eine andere Darstellung für Konstruktoren nutzen. Sind beispielsweise die Module der Funktionen wichtig, etwa bei mehrfachem Vorkommen gleichnamiger Funktionen, so können wir eine Tupeldarstellung als qualifizierten Bezeichner wählen. Der obige Term könnte dann wie folgt definiert werden:

```
TermCons ("Prelude", "+") [TermVar 23, TermVar 24]
```

Mit der erarbeiteten Darstellung von Termen, ist es nun möglich, weitere Datentypen für Gleichungen einzuführen. Gleichungen sind dabei im Wesentlichen nichts anderes, als die Gegenüberstellung zweier Terme, die meist durch ein Gleichheitszeichen oder ein ähnliches Symbol verbunden sind. In einer sehr einfachen Form lässt sich deshalb eine Termgleichung als ein Paar von zwei Termen definieren:

```
type TermEq f = (Term f, Term f)
```

Weiterhin können wir eine Menge von Termgleichungen einfach als Liste spezifizieren:

```
type TermEqs f = [TermEq f]
```

Die grundlegenden Strukturen haben wir nun eingeführt, sodass einige wesentliche Operationen auf diesen Datentypen definiert werden können. Beginnen wollen wir dabei mit der Unterscheidung zwischen Variablen und der Applikation von Konstruktoren. Wir definieren deshalb die beiden Prädikate `isVarTerm` und `isConsTerm`, die als Eingabe einen Term erwarten und genau dann `True` zurückliefern, wenn das Argument dem `TermVar`- beziehungsweise dem `TermCons`-Konstruktor entspricht. Für eine einfachere Definition nutzen wir in der `isTermVar`-Funktion den Negations-Operator `not`. Die genaue Implementierung lautet demnach:

```
isConsTerm :: Term f -> Bool
isConsTerm (TermVar _)     = False
isConsTerm (TermCons _ _) = True

isVarTerm :: Term f -> Bool
isVarTerm = not . isConsTerm
```

Für die Berechnung aller Variablen in einem beliebigen Term, folgen wir dem Ansatz aus Definition 2.3 und definieren zwei Operationen `tVars` sowie `tVarsAll`. Letztere berechnet eine Liste aller Variablen in einem Term t nach dem folgenden Muster:

- Falls `t = TermVar v`, so gebe eine einelementige Liste `[v]` zurück.
- Falls `t = TermCons c ts`, so wende `tVarsAll` auf alle Terme in `xs` an und konkateniere die resultierenden Listen.

Da in der mit `tVarsAll` berechneten Liste jedoch auch Variablen mehrfach vorkommen können, müssen wir die Liste unter der Verwendung von `nub` in eine Menge umwandeln. Insgesamt können wir die beiden Funktionen wie folgt umsetzen:

```
tVars :: Term f -> [VarIdx]
tVars = nub . tVarsAll

tVarsAll :: Term f -> [VarIdx]
tVarsAll (TermVar v)      = [v]
tVarsAll (TermCons _ ts) = concatMap tVarsAll ts
```

Ähnlich dem obigen Vorgehen, ist es auch möglich, die Liste beziehungsweise Menge aller Konstruktoren eines Terms zu berechnen. Dafür existieren die beiden Funktionen `tConsAll` und `tCons`, deren Implementierung wir zwecks Ähnlichkeit nicht genauer betrachten wollen. Allerdings führen wir noch zwei *Smart*-Konstruktoren `tConst` sowie `tOp`

ein, mit deren Hilfe Konstanten und binäre Operatoren leicht erstellt werden können. Für ein Beispiel betrachten wir hier den Term $x + y$ von oben, der äquivalent zu folgendem Funktionsaufruf ist:

```
tOp (TermVar 23) "+" (TermVar 24)
```

Weiterhin wollen wir Funktionen für die ebenfalls in Definition 2.3 beschriebenen Eigenschaften von Termen bereitstellen. Dabei prüft `isGround`, ob ein Term ein Grundterm ist, und `isLinear`, ob ein Term linear ist, insbesondere also keine Variable mehrfach vorkommt. Die Implementierung erfolgt im ersten Fall lediglich mit dem Prädikat `null`, das für eine leere Liste `True` liefert, und der zuvor definierten Funktion `tVarsAll`. Warum wir nicht `tVars` verwenden, begründet sich in der hier nicht benötigten Voraussetzung einer Menge von Variablen. Für die zweite Funktion definieren wir uns zunächst ein Hilfsprädikat `unique`, das eine Liste auf mehrfach vorkommende Elemente überprüft. Dann verknüpfen wir `tVarsAll` und `unique` zu folgender Funktionsdefinition:

```
isLinear :: Term f -> Bool
isLinear = unique . tVarsAll
```

Eine weitere Eigenschaft, die es umzusetzen gilt, ist die eines *normalen* Terms. Das bedeutet, dass hinter einer Variablen kein Funktionssymbol steht. Intuitiv müssen wir einen Term also Schritt für Schritt durchlaufen, und prüfen, dass hinter jeder Variablen ebenfalls nur Variablen stehen. Hier kommt uns die zuvor eingeführte Funktion `isVarTerm` zur Hilfe. Insgesamt führt uns das für einen Term t zu folgendem Algorithmus:

- Falls t eine Variable oder eine Konstante ist, so gebe `True` zurück.
- Falls $t = f(t_1, \dots, t_n)$:
  - Falls t_1 eine Variable ist, so müssen auch $t_2, \dots, t_n$ Variablen sein.
  - Falls t_1 ein Konstruktor ist, so prüfe die Eigenschaft für t_1 und $f(t_2, \dots, t_n)$.

Die Umsetzung des Algorithmus erfolgt ohne Modifikationen in eine Funktion `isNormal`:

```
isNormal :: Term f -> Bool
isNormal (TermVar _)         = True
isNormal (TermCons _ [])     = True
isNormal (TermCons c (t:ts))
  = case t of
      (TermVar _)    -> all isVarTerm ts
      (TermCons _ _) -> (isNormal t) && (isNormal (TermCons c ts))
```

Neben den Funktionen zum Nachweisen von Eigenschaften, wollen wir auch Operationen für Variablen definieren. Insbesondere aber sind das Berechnen von minimalen und maximalen Variablen sowie das Umbenennen von Variablen interessant. Da wir bereits eine Funktion für das Berechnen einer Menge der Variablen in einem Term eingeführt

haben, können wir auf simple Weise das minimale beziehungsweise das maximale Element extrahieren. Da es natürlich auch vorkommen kann, dass ein Term keine Variablen besitzt, wählen wir als Rückgabetyp der beiden Funktionen den `Maybe`-Datentyp. Die Definitionen entsprechen somit:

```
maxVarInTerm :: Term f -> Maybe VarIdx
maxVarInTerm t = case tVars t of
                   []       -> Nothing
                   vs@(_:_) -> Just (maximum vs)

minVarInTerm :: Term f -> Maybe VarIdx
minVarInTerm t = case tVars t of
                   []       -> Nothing
                   vs@(_:_) -> Just (minimum vs)
```

Das Umbenennen von Variablen ist ebenfalls nicht weiter kompliziert umzusetzen. Vielmehr reicht es aus, einen Term zu durchlaufen und jedes Vorkommen einer Variable um einen übergebenen Wert zu erhöhen. Wir können dies mit einer simplen Addition lösen, da wir zuvor den Typ von Variablen als eine natürliche Zahl festgelegt haben. Somit können wir eine Funktion `renameTermVars` mit der folgenden Signatur umsetzen:

```
renameTermVars :: Int -> Term f -> Term f
```

Weiterhin wollen wir einen Term normalisieren können. Das bedeutet, dass die Variablen eines Terms durch einen von Null an aufsteigenden Wert ersetzt werden. Mehrfache Vorkommen einer Variablen erhalten dabei natürlich den gleichen Wert. An einem Beispiel wird die Transformation deutlich:

$$x + (z + (y + x)) \rightarrow a + (b + (c + a))$$

Für die Implementierung benötigen wir zunächst eine geeignete Darstellung, die es ermöglicht, die einzelnen Zuordnungen der alten und neuen Variablen zu speichern. Das Modul `FiniteMap` stellt hierfür die notwendigen Datentypen und Operationen bereit. Da wir in den nachfolgenden Abschnitten häufiger Teile dieser Bibliothek verwenden, sollte sich für ein besseres Verständnis mit diesem Modul vertraut gemacht werden. Somit erstellen wir uns eine Instanz der `FiniteMap`, die als Ersetzungsfunktion dient. Wir müssen nun lediglich durch den zu normalisierenden Term durchgehen und auf jede Variable die Ersetzungsfunktion anwenden. Das Ergebnis können wir in der folgenden Deklaration der Operation `normalizeTerm` erkennen:

```
normalizeTerm :: Term f -> Term f
normalizeTerm t = normalize t
  where
    sub = listToFM (<) (zip (tVars t) (map TermVar [0..]))
    normalize :: Term f -> Term f
```

```
normalize t'@(TermVar v)   = fromMaybe t' (lookupFM sub v)
normalize (TermCons c ts) = TermCons c (map normalize ts)
```

Ähnlich der Umbenennung von Variablen wollen wir auch eine derartige Funktion für Konstruktoren bereitstellen. Dabei wird die eigentliche Umbenennung nicht in der zu definierenden Funktion selbst umgesetzt, sondern von einer ihr übergebenen Funktion. Die Implementierung erfolgt auch hier lediglich mit einem einfachen Durchlaufen des Terms, wobei auf jeden Konstruktor die übergebene Funktion angewendet wird. Somit gelangen wir zu der Definition von `mapTerm`:

```
mapTerm :: (a -> b) -> Term a -> Term b
mapTerm _ (TermVar v)     = TermVar v
mapTerm f (TermCons c ts) = TermCons (f c) (map (mapTerm f) ts)
```

Der Vergleich von Termen hinsichtlich des enthaltenen Konstruktors ist für einige Funktionalitäten, die im Verlauf diese Kapitels implementiert werden, eine wichtige Unterstützung. Gerade das Auffinden einer Regel für einen Term erfordert das Wissen über die Gleichheit der Funktionssymbole aber auch der Anzahl der Argumente. So ist es prinzipiell möglich, dass etwa der Konstruktor zweier Terme identisch ist, nicht aber die Anzahl der Argumente. Für genau diesen Zweck definieren wir eine Operation `eqConsPattern`, die zwei Terme auf ebendiese Eigenschaft prüft:

```
eqConsPattern :: Term f -> Term f -> Bool
```

Zu guter letzt wollen wir noch einige nützliche Funktionen für die anschauliche Ausgabe (*Pretty-Printing*) der am Anfang dieses Abschnittes eingeführten Datentypen umsetzen. Dazu gehören zwei Operationen mit den folgenden Signaturen:

```
showVarIdx :: VarIdx -> String

showTerm :: (f -> String) -> Term f -> String
```

Da in Curry aktuell keine Typklassen existieren, erledigt eine zusätzlich übergebene Funktion für `showTerm` die Transformation von Konstruktoren in Zeichenketten. Ein Beispiel soll zeigen, wie ein Term in eine übersichtliche Darstellung überführt werden kann:

```
kics2> showTerm id (tOp (tConst "0") "+"
                        (tOp (tConst "0") "+" (TermVar 0)))
0 + (0 + a)
```

Ähnlich der `showTerm`-Funktion existieren auch Operationen für Termgleichungen und den Datentyp `TermEqs`. Aufgrund der fehlenden Relevanz, wollen wir an dieser Stelle jedoch nicht weiter darauf eingehen.

3.2 Substitution

Mit dem Modul `Rewriting.Substitution` wird in KiCS2, basierend auf unserer Termdarstellung, bereits eine Möglichkeit bereitgestellt, Variablen durch Terme zu ersetzen. Dennoch wollen wir einige Veränderungen und Erweiterung an diesem Modul vornehmen. Für ein besseres Verständnis ist es jedoch ratsam, zunächst den allgemeinen Typ für eine Substitution vorzustellen. Dieser lautet dabei wie folgt:

```
type Subst f = FM VarIdx (Term f)
```

Eine Substitution ist also nichts weiter als eine Menge von Zuordnungen, mit Variablen auf der einen und Termen auf der anderen Seite. Dies entspricht im Wesentlichen der in Definition 2.5 getroffenen formalen Festlegung. Die folgenden Signaturen liefern eine Übersicht über die bereits implementierten Funktionen, an denen nur kleine Anpassungen vorgenommen wurden:

```
emptySubst :: Subst f

applySubst :: Subst f -> Term f -> Term f

extendSubst :: Subst f -> VarIdx -> Term f -> Subst f

lookupSubst :: Subst f -> VarIdx -> Maybe (Term f)

showSubst :: (f -> String) -> Subst f -> String
```

Analog zu `applySubst` existieren natürlich auch äquivalente Operationen für Termgleichungen und Listen von Termgleichungen. Wir können also bereits Substitutionen auf Terme anwenden, sie erweitern und in diesen nach Zuordnungen suchen. Das Erstellen von Substitutionen ist allerdings noch etwas umständlich. Gerade bei vielen Zuordnungen würden wir eine Vielzahl an verschachtelten `extendSubst`-Aufrufen benötigen. Daher definieren wir zuerst eine neue Funktion, die eine Liste von `VarIdx`-`Term`-Paaren in eine Substitution umwandelt:

```
substOrder :: VarIdx -> VarIdx -> Bool
substOrder = (<)

listToSubst :: [(VarIdx, Term f)] -> Subst f
listToSubst = listToFM substOrder
```

Für das *Narrowing*, auf das wir in Abschnitt 3.9 genauer eingehen, benötigen wir noch zwei weitere Operationen. Dies sind die Komposition zweier Substitutionen und das Einschränken einer Substitution auf eine Menge von Variablen. Daher definieren wir die folgenden beiden Funktionen:

```
composeSubst :: Subst f -> Subst f -> Subst f

restrictSubst :: Subst f -> [VarIdx] -> Subst f
```

Mit der Operation `composeSubst` ist es dabei möglich zwei Substitutionen φ und σ nach dem Muster $\varphi \circ \sigma$ zu vereinen.

3.3 Unifikation

Neben dem Modul `Rewriting.Substitution` beinhaltet KiCS2 auch das darauf aufbauende Modul `Rewriting.Unification`, zur Unifikation von Termen. Mit der Funktion

```
unify :: TermEqs f -> Either (UnificationError f) (Subst f)
```

kann dabei eine passende Substitution für eine Liste von Termgleichungen berechnet werden. Die Unifikation von zwei Termen t_1 und t_2 kann entsprechend mit dem Aufruf `unify [(`t_1`,`t_2`)]` berechnet werden. Im Fall eines Fehlschlags wird eine Instanz des Datentyps `UnificationError` zurückgegeben, der wie folgt definiert ist:

```
data UnificationError f = Clash (Term f) (Term f)
                        | OccurCheck VarIdx (Term f)
```

Die Konstruktoren `Clash` und `OccurCheck` entsprechen dabei den beiden Szenarien, in denen eine Unifikation nicht möglich ist. Dies ist zum einen, wenn zwei Terme verschiedene Konstruktoren haben, und zum anderen, wenn eine Variable im zweiten Term als Teilterm vorkommt. Für den Fall, dass man wissen will, ob zwei Terme unifizierbar sind, aber nicht am eigentlichen Unifikator interessiert ist, definieren wir ein Prädikat `unifiable`. Abschließend ermöglichen wir mit einer Funktion `showUnificationError` das *Pretty-Printing* des Datentyps `UnificationError`.

3.4 Positionen

Zur Bestimmung von Teiltermen haben wir in Definition 2.4 den Begriff der Position eingeführt. Eine Position ist dabei eine Folge natürlicher Zahlen. Es liegt deshalb nahe, die Umsetzung in Curry als einfache Liste ganzer Zahlen vorzunehmen. Wir definieren also ein Typsynonym für eine Position:

```
type Pos = [Int]
```

Weiterhin entnehmen wir der Definition, dass jeder Term mindestens eine Wurzelposition aufweist. Es ist also sinnvoll, dieser mit `eps` ein Äquivalent in Curry zu widmen. Dabei ist `eps` lediglich als leere Liste definiert. Für den Vergleich von Positionen (wie über, unter, usw.) finden wir in Definition 2.4 eindeutige Aussagen vor, sodass die Umsetzung sofort ersichtlich ist. Wir definieren demnach fünf Operationen, die jeweils zwei Positionen als Eingabe erwarten und einen Wahrheitswert als Ausgabe liefern:

```
isPosAbove :: Pos -> Pos -> Bool

isPosBelow :: Pos -> Pos -> Bool

isPosDisjunct :: Pos -> Pos -> Bool

isPosLeft :: Pos -> Pos -> Bool

isPosRight :: Pos -> Pos -> Bool
```

Die Funktionen `isPosBelow` und `isPosRight` sind dabei mittels `flip` umgesetzt, sodass ein Aufruf jeweils die gegenteilige Funktion mit umgekehrten Argumenten aufruft. Ein Beispiel soll die Funktionsweise verdeutlichen:

```
kics2> isPosBelow [1, 2] [1, 2, 3]
False

kics2> isPosRight [1, 3] [1, 2, 3]
True
```

Zur Bestimmung aller Positionen innerhalb eines Terms nutzen wir das Konstrukt der *List Comprehension*. Dies ermöglicht eine Implementierung anhand der Spezifikation in Definition 2.4. Die genaue Umsetzung ist dem nachfolgenden Listing zu entnehmen:

```
positions :: Term f -> [Pos]
positions (TermVar _)     = [eps]
positions (TermCons _ ts) = eps:[i:p | (i, t) <- zip [1..] ts,
                                       p <- positions t]
```

In Definition 2.5 haben wir neben der Substitution auch die Selektion und das Ersetzen von Teiltermen spezifiziert. Wir führen deshalb die Funktionen `|>` und `replaceTerm` ein. Das nachfolgende Beispiel verdeutlicht, wie diese genutzt werden können, um Teilterme zu selektieren oder zu ersetzen:

```
kics2> (tOp (TermVar 0) "+" (TermVar 1)) |> [1]
TermVar 0

kics2> replaceTerm (tOp (TermVar 0) "+" (TermVar 1)) [1] (TermVar 2)
TermCons "+" [TermVar 2, TermVar 1]
```

Abschließend wollen wir aber auch für Positionen statt der Listen-Darstellung eine anschauliche Ausgabe implementieren. Dazu definieren wir die Funktion `showPos`, die eine Position in eine Zeichenkette überführt. Die Operation `.>`, die zwei Positionen mittels der Konkatenation von Listen verbindet, rundet schließlich die Schnittstelle des Moduls `Rewriting.Position` ab.

3.5 Regeln und Termersetzungssysteme

Nachdem wir in den letzten Abschnitten einige grundlegende Aspekte, wie etwa Terme und Positionen, umgesetzt haben, können wir uns nun an die Implementierung von Regeln und Termersetzungssystemen wagen. Dabei werden wir jedoch nicht nur einige Typen definieren, sondern auch eine Reihe an Funktionalitäten umsetzen, mit denen Regeln und Termersetzungssysteme verarbeitet werden können. Beginnen wollen wir allerdings mit der Repräsentation der wichtigsten Typen. Nach Definition 2.3 wissen wir, dass ein Termersetzungssystem nichts weiter als eine Menge von Regeln der Form $l \to r$ ist. Dabei sind l und r Terme. Wir können also folgende Definitionen vornehmen:

```
type Rule f = (Term f, Term f)

type TRS f = [Rule f]
```

Eine Regel ist demnach ein Paar von Termen, wobei der erste Eintrag die linke und der zweite Eintrag die rechte Seite repräsentiert. Die Darstellung von Termersetzungssystemen erfolgt als Liste von Regeln. Für ein Beispiel nutzen wir die Addition von Peano-Zahlen:

```
-- 0 + y -> y
-- s(x) + y -> s(x + y)
-- x + 0 -> x
-- x + s(y) -> s(x + y)
peanoAddTRS :: TRS String
peanoAddTRS = [(tOp (tConst "0") "+" varY, varY),
               (tOp (TermCons "s" [varX]) "+" varY,
                TermCons "s" [tOp varX "+" varY]),
               (tOp varX "+" (tConst "0"), varX),
               (tOp varX "+" (TermCons "s" [varY]),
                TermCons "s" [tOp varX "+" varY])]
```

Nachdem wir nun die wichtigsten beiden Typen eingeführt haben, wollen wir darauf aufbauende Funktionen entwickeln. Dazu gehört die Berechnung minimaler und maximaler Variablen, wie wir es bereits aus Abschnitt 3.1 kennen. Wir nutzen dabei ebendiese Operationen, da für Regeln aufgrund der Eigenschaft $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ nur der linke Term untersucht werden muss. Für ein Termersetzungssystem bestimmen wir zunächst das minimale beziehungsweise das maximale Element jeder Regel und ermitteln daraus das Minimum und Maximum. Somit erhalten wir folgende Deklarationen:

```
minVarInRule :: Rule f -> Maybe VarIdx

maxVarInRule :: Rule f -> Maybe VarIdx

minVarInTRS :: TRS f -> Maybe VarIdx
```

```
maxVarInTRS :: TRS f -> Maybe VarIdx
```

Ebenfalls wollen wir das Umbenennen und Normalisieren von Regeln ermöglichen. Das Umbennnen ist dabei mit `renameTermVars` leicht umzusetzen, da die Funktion lediglich auf beide Seiten einer Regel angewendet werden muss. Für das Normalisieren nutzen wir die in Abschnitt 3.2 eingeführten Substitutionen, um die alten Variablen neuen zuzuordnen. Anschließend wenden wir die Substitution mit `applySubst` auf beide Seiten der Regel an. Die Signatur der Funktionen ist dabei die Folgende:

```
renameRuleVars :: Int -> Rule f -> Rule f

normalizeRule :: Rule f -> Rule f
```

Für Termersetzungssysteme existieren mit `renameTRSVars` und `normalizeTRS` äquivalente Funktionen. Die Implementierung erfolgt dabei unter Verwendung der zuvor definierten Operationen, da diese lediglich auf alle Regeln angewendet werden müssen. Die Menge aller Variablen und aller Konstruktoren einer Regel können mit `rVars` und `rCons` berechnet werden. Zur Überprüfung, ob ein Term eine passende Regel (gleiches Funktionssymbol und gleiche Stelligkeit) in einem Termersetzungssystem hat, definieren wir eine Funktion `hasRule`. Dafür nutzen wir `eqConsPattern` und vergleichen den Term mit jeder linken Regelseite:

```
hasRule :: TRS f -> Term f -> Bool
hasRule trs t = any ((eqConsPattern t) . fst) trs
```

Zuletzt wollen wir noch einige Eigenschaften von Termersetzungssystemen in Curry umsetzen. Dazu gehören insbesondere die Begriffe eines Musters und eines konstruktorbasierten TES. Die formale Definition dazu haben wir bereits in Definition 2.12 formuliert. Die beschriebenen Eigenschaften lassen sich damit in der folgenden Form definieren:

```
isPattern :: TRS f -> Term f -> Bool

isConsBased :: TRS f -> Bool
```

Weiterhin wollen wir ermitteln, ob ein Termersetzungssystem links-linear (links-normal) ist. Dies ist der Fall, falls jede linke Regelseite linear (normal) ist. Dazu stellen wir die beiden Prädikate `isLeftLinear` und `isLeftNormal` bereit. Auch der Vergleich von Regeln in der Hinsicht auf Varianten (gleich bis auf Umbenennung von Variablen) wird mit der Funktion `isVariantOf` unterstützt. Zu guter Letzt ermöglichen `showRule` und `showTRS` das *Pretty-Printing* von Regeln und Termersetzungssystemen.

3.6 Kritische Paare

In den vorherigen Abschnitten haben wir alle notwendigen Voraussetzungen für die Berechnung sogenannter kritischer Paare in Curry umgesetzt. Wir können also ein Modul

`Rewriting.CriticalPairs` implementieren, das im Kern aus der Funktion `cPairs` besteht. Dabei erwartet `cPairs` eine Liste von Regeln, also ein Termersetzungssystem, und liefert eine Liste aller kritischen Paare. Für eine ausdrucksstärkere Signatur definieren wir zusätzlich ein Typsynonym für ein kritisches Paar als Paar von zwei Termen. Die Umsetzung erfolgt nun schließlich gemäß Definition 2.7 und der Anwendung von *List Comprehension*. An dieser Stelle wollen wir allerdings nur die Signatur der Funktion betrachten:

```
type CPair f = (Term f, Term f)

cPairs :: TRS f -> [CPair f]
```

Neben der reinen Berechnung kritischer Paare, können wir damit auch Eigenschaften von Termersetzungssystemen nachweisen. So lässt sich etwa nach Definition 2.8 zeigen, ob ein Termersetzungssystem orthogonal oder schwach orthogonal ist. Mit der Funktion `cPairs` können wir diese Bedingungen nun in Curry formulieren. Dabei erhalten wir zwei Prädikate, die wie folgt implementiert sind:

```
isOrthogonal :: TRS f -> Bool
isOrthogonal trs = (isLeftLinear trs) && (null (cPairs trs))

isWeakOrthogonal :: TRS f -> Bool
isWeakOrthogonal trs = (isLeftLinear trs) &&
                           (all (uncurry (==)) (cPairs trs))
```

Die Berechnung kritischer Paare wollen wir nun am Beispiel des Termersetzungssystems aus Abschnitt 3.5 demonstrieren:

```
kics2> putStr . (showCPairs id) . cPairs peanoAddTRS
⟨0, 0⟩
⟨s(x), s(0 + x)⟩
⟨s(x + 0), s(x)⟩
⟨s(x + s(y)), s(s(x) + y)⟩
⟨s(x), s(x + 0)⟩
⟨s(0 + x), s(x)⟩
⟨s(s(x) + y), s(x + s(y))⟩
```

Die Ausgabe wird dabei mit den Funktionen `showCPair` und `showCPairs` übersichtlich dargestellt. Die genaue Funktionsweise ist an dieser Stelle jedoch nicht weiter von besonderem Interesse.

3.7 Reduktion von Termen

In diesem Abschnitt wollen wir ein Modul `Rewriting.Strategy` realisieren. Bestandteil dieses Moduls sind die Darstellung von Reduktionsstrategien, die Reduktion eines Terms

und das *Pretty-Printing* von Reduktionsschritten. Beginnen werden wir mit der allgemeinen Darstellung einer Reduktionsstrategie S. Mit Definition 2.9 wissen wir bereits, dass S für ein gegebenes Termersetzungssystem und einen Term t, eine Menge von Redex-Positionen in t berechnet. Wir können deshalb eine Reduktionsstrategie als Funktion definieren, die ein Termersetzungssystem und einen Term erwartet und eine Liste von Positionen liefert:

```
type RStrategy f = TRS f -> Term f -> [Pos]
```

Mit der allgemeinen Darstellung ist es nun möglich, konkrete Reduktionsstrategien zu entwickeln. Zunächst werden wir allerdings zwei nützliche Hilfsfunktionen `seqRStrategy` und `parRStrategy` einführen, die anhand einer Ordnungsrelation eine sequenzielle beziehungsweise eine parallele Reduktionsstrategie erzeugen:

```
seqRStrategy :: (Pos -> Pos -> Ordering) -> RStrategy f

parRStrategy :: (Pos -> Pos -> Ordering) -> RStrategy f
```

Dabei werden die übergebenen Relationen dazu genutzt, die Menge der Redex-Positionen so zu sortieren, dass am Ende eine beziehungsweise mehrere (kleinste) Positionen selektiert sind. Die Menge der Redex-Positionen kann dabei mit der Funktion `redexes` für ein Termersetzungssystem und einen Term abgerufen werden. Mit der Hilfe dieser Funktionen definieren wir nun die sechs aus Definition 2.10 bekannten Reduktionsstrategien:

```
liRStrategy :: RStrategy f

loRStrategy :: RStrategy f

riRStrategy :: RStrategy f

roRStrategy :: RStrategy f

piRStrategy :: RStrategy f

poRStrategy :: RStrategy f
```

Mit der Deklaration der Reduktionsstrategien fehlt uns lediglich eine Funktion, die einen Term zu einer Normalform reduziert. Bevor wir allerdings die vollständige Reduktion ermöglichen, wollen wir das Reduzieren an ausgewählten Positionen unterstützen. Dazu definieren wir die Funktionen `reduceAt` und `reduceAtL`. Für eine vollständige Reduktion reicht es daher aus, `reduceAtL` wiederholt mit der von einer Strategie berechneten Liste von Positionen hintereinander aufzurufen. Dies führt uns zur Funktion `reduce`, die wie folgt in Curry umgesetzt ist:

```
reduce :: RStrategy f -> TRS f -> Term f -> Term f
```

```
reduce s trs t = case s trs t of
                   []       -> t
                   ps@(_:_) -> reduce s trs (reduceAtL trs ps t)
```

Für den Fall, dass ein Term mit mehreren Termersetzungssystemen reduziert werden soll, bieten wir außerdem die Operation `reduceL` an. Der Unterschied gegenüber `reduce` besteht hier darin, dass nicht nur ein Termersetzungssystem sondern eine Liste von Termersetzungssystemen übergeben wird. Die Implementierung erfolgt allerdings mit `reduce`, da die Liste zu einem einzigen Termersetzungssystem konkateniert wird. Da jedoch nicht jeder Term mit jeder Strategie zu einer Normalform reduziert werden kann, benötigen wir auch eine Möglichkeit, die Anzahl der Reduktionsschritte zu begrenzen. Hierzu ergänzen wir die Funktionen `reduce` sowie `reduceL` um ein ganzzahliges Argument und erhalten:

```
reduceBy :: RStrategy f -> TRS f -> Int -> Term f -> Term f

reduceByL :: RStrategy f -> [TRS f] -> Int -> Term f -> Term f
```

Neben der reinen Berechnung einer Normalform, ist es in manchen Situationen wünschenswert, alle Reduktionsschritte einzeln nachzuvollziehen. Die aktuelle Implementierung erlaubt dies jedoch noch nicht. Daher benötigen wir eine geeignete Darstellung, die alle Zwischenschritte speichert. Dies führt uns zu einem Datentyp `Reduction`, den wir im Folgenden einführen:

```
data Reduction f = NormalForm (Term f)
                 | RStep (Term f) [Pos] (Reduction f)
```

Mit dem Konstruktor `RStep` ist es dabei möglich die Zwischenschritte zu verschachteln, bis der Konstruktor `NormalForm` die Reduktion terminiert. Die Funktion `reduction` ist analog zu `reduce` implementiert, liefert jedoch eine Instanz des eben eingeführten Datentyps `Reduction`:

```
reduction :: RStrategy f -> TRS f -> Term f -> Reduction f
reduction s trs t
  = case s trs t of
      []       -> NormalForm t
      ps@(_:_) -> RStep t ps (reduction s trs (reduceAtL trs ps t))
```

Auch hier wollen wir zusätzlich weitere Funktionen für Listen von Termersetzungssystemen und das Reduzieren mit einer Anzahl an Schritten bereitstellen. Dazu definieren wir `reductionL`, `reductionBy` und die Funktion `reductionByL`. Abrunden werden wir die Schnittstelle mit der Operation `showReduction`, die eine Reduktion mit allen Zwischenschritten anschaulich auf dem Terminal ausgibt.

3.8 Definierende Bäume

Neben den im letzten Abschnitt eingeführten Reduktionsstrategien wollen wir hier eine weitere Strategie φ implementieren. Dazu benötigen wir allerdings zunächst eine geeignete Darstellung sogenannter definierender Bäume. In Definition 2.13 haben wir dazu einen definierenden Baum bereits formal eingeführt. Dabei haben wir festgestellt, dass ein solcher Baum entweder ein Blatt mit einer Regel oder ein Verzweigungsknoten mit einem Muster, einer induktiven Position und einer Liste von definierenden Bäumen ist. Damit wir aber für die Narrowing-Strategie λ keine Erweiterung dieser Darstellung umsetzen müssen, wollen wir auch den Oder-Knoten aus Definition 2.24 übernehmen. Wir erhalten somit folgende Definition eines Datentyps:

```
data DefTree f = Leaf (Rule f)
               | Branch (Term f) Pos [DefTree f]
               | Or (Term f) [DefTree f]
```

Aufbauend auf diesem Datentyp wollen wir nun zwei Operationen bereitstellen, um das Muster und das oberste Symbol (also Variable oder Konstruktor) eines definierenden Baumes zu berechnen. Dazu betrachten wir die folgenden Signaturen:

```
dtPattern :: DefTree f -> Term f

dtRoot :: DefTree f -> Either VarIdx f
```

Für ein gegebenes Termersetzungssystem soll es weiterhin möglich sein, alle definierenden Bäume zu ermitteln. Wir benötigen also eine geeignete Funktionalität. Zuvor wollen wir allerdings alle induktiven Positionen eines Termersetzungssystems errechnen. Eine Position ist dabei induktiv, wenn alle Regeln an dieser Stelle einen Konstruktorterm aufweisen. Die folgende Funktion liefert die gewünschte Liste induktiver Positionen:

```
idtPositions :: TRS f -> [Pos]
```

Mit Hilfe dieser Operation können wir nun die definierenden Bäume eines Termersetzungssystems berechnen. Dabei nutzen wir die Permutationen der mit `idtPositions` generierten Liste, um alle Bäume zu ermitteln. Neben der Variante, die ein Termersetzungssystem erwartet, bieten wir auch eine Funktion mit einer Liste als Argument an:

```
defTrees :: TRS f -> [DefTree f]

defTreesL :: [TRS f] -> [DefTree f]
```

Zur besseren Darstellung, wollen wir nun eine Möglichkeit anbieten, einen definierenden Baum in die Graph-Beschreibungssprache DOT [GKN15] zu überführen. Anschließend ist es dann möglich, aus der entsprechenden Repräsentation zum Beispiel eine anschauliche PDF-Datei zu erstellen. Wir definieren also eine Funktion `dotifyDefTree` mit der folgenden Signatur:

```
dotifyDefTree :: (f -> String) -> DefTree f -> String
```

Um das Kopieren der damit generierten Zeichenkette und das anschließende Speichern in eine Datei zu vereinfachen, stellen wir auch eine Operation `writeDefTree` bereit. Diese erwartet neben dem definierenden Baum auch den Namen einer Datei, in die die Darstellung im DOT-Format geschrieben werden soll. Die Funktion ist definiert als:

```
writeDefTree :: (f -> String) -> DefTree f -> String -> IO ()
```

Für ein Beispiel wollen wir die Addition von Peano-Zahlen betrachten. Dabei bezeichne `peanoAddTRS` das Termersetzungssystem, das folgender formalen Definition entspricht:

$$
\begin{aligned}
0 + 0 &\to 0 \\
0 + s(y) &\to s(y) \\
s(x) + 0 &\to s(x) \\
s(x) + s(y) &\to s(s(x + y))
\end{aligned}
$$

Für `peanoAddTRS` können wir dann den ersten definierenden Baum in eine Datei mit dem Namen `defTree.dot` speichern.

```
defTree :: DefTree String
defTree = (defTrees peanoAddTRS) !! 0

kics2> writeDefTree id defTree "defTree.dot"
```

Anschließend dient das Programm `dot` dazu, die Datei `defTree.dot` beispielsweise in eine PDF-Datei oder PNG-Grafik umzuwandeln. Für eine ausführliche Anleitung dieses Programmes sei jedoch das entsprechende Handbuch [GKN15] zu empfehlen. Die resultierende Grafik ist in Abbildung 3.1 dargestellt.

Als nächstes wollen wir die zu Beginn angesprochene Reduktionsstrategie φ umsetzen. Dabei orientieren wir uns an Definition 2.15 und erhalten folgende Funktionen:

```
loDefTrees :: [DefTree f] -> Term f -> Maybe (Pos, [DefTree f])

phiRStrategy :: Int -> RStrategy f
```

Mit `loDefTrees` erhalten wir die Position und die definierenden Bäume für das linkeste äußerste definierte Funktionssymbol des übergebenen Terms. Über die bei `phiRStrategy` von Null startende ganze Zahl wird der definierende Baum ausgewählt, der für die Reduktion genutzt wird. Ist die Zahl undefiniert, so wird standardmäßig der erste Baum für eine Funktion genutzt.

Abschließend stellen wir mit `hasDefTree` und `selectDefTrees` zwei Operationen für Terme bereit. Zum einen kann für einen Term überprüft werden, ob dessen Konstruktor einen definierenden Baum in einer Liste enthält, zum anderen können für Terme alle passenden definierenden Bäume aus einer Liste selektiert werden.

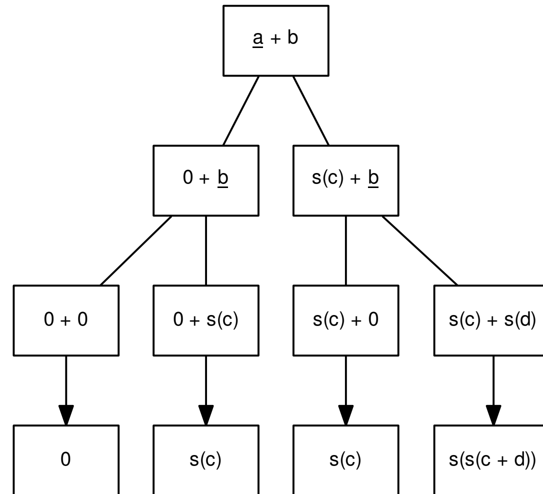

Abbildung 3.1: Definierender Baum der Peano-Additionsfunktion

3.9 Narrowing

In den vorherigen Abschnitten haben wir die Reduktion von Termen implementiert. Als nächstes wollen wir deshalb das *Narrowing* auf Termen und insbesondere das Lösen von Gleichungen umsetzen. Ähnlich der Reduktion nutzen wir auch hier definierte Strategien für einzelne Narrowing-Schritte. Zuerst definieren wir uns deshalb nach Definition 2.18 einen Typ für eine Narrowing-Strategie:

```
type NStrategy f = TRS f -> Term f -> [(Pos, Rule f, Subst f)]
```

Eine Narrowing-Strategie ist dabei eine Funktion, die ein Termersetzungssystem und einen Term erwartet und entsprechend alle möglichen Narrowing-Schritte, in der Form einer Position, einer Regel und einer Substitution, zurückgibt. Auf der Grundlage dieser Deklaration können wir die in Abschnitt 2.7 eingeführten Narrowing-Strategien umsetzen:

```
stdNStrategy :: NStrategy f

imNStrategy :: NStrategy f

omNStrategy :: NStrategy f

loNStrategy :: NStrategy f

lazyNStrategy :: NStrategy f

wnNStrategy :: NStrategy f
```

Die Strategien sind zwar nun definiert, wir benötigen jedoch noch eine Funktion, um die resultierenden Narrowing-Schritte auf einen Term anzuwenden. Dafür dient uns die Operation `narrowBy`, die auf einen Term die angegebene Anzahl an Narrowing-Schritten anwendet. Als Ergebnis wird eine Liste von Paaren zurückgeliefert, wobei ein Paar aus der Substitution und dem resultierenden Term besteht. Die Signatur entspricht:

```
narrowBy :: NStrategy f -> TRS f -> Int -> Term f
         -> [(Subst f, Term f)]
```

Da diese Funktion jedoch nur das Resultat aller angewendeten Narrowing-Schritte liefert, wollen wir auch eine Darstellung ermöglichen, mit der jeder einzelne Schritt sichtbar ist. Wir nutzen dabei die Idee nach der wir den Datentyp `Reduction` umgesetzt haben und erhalten folgende Deklaration:

```
data Narrowing f = NTerm (Term f)
                 | NStep (Term f) Pos (Subst f) (Narrowing f)
```

Damit erhalten wir also eine kompakte Darstellung, die insbesondere für die Ausgabe im Terminal sehr nützlich ist. Daher wollen wir neben eines Äquivalents zu `narrowBy` auch eine Funktion `showNarrowing` für das *Pretty-Printing* bereitstellen:

```
narrowingBy :: NStrategy f -> TRS f -> Int -> Term f -> [Narrowing f]

showNarrowing :: (f -> String) -> Narrowing f -> String
```

Bisher sind nur einzelne Narrowing-Ableitungen möglich. Das Ziel ist es folglich, eine Übersicht aller Narrowing-Ableitungen eines Terms mit einer bestimmten Strategie zu ermöglichen. Eine Graph-Darstellung scheint dabei am sinnvollsten. Wir definieren also einen entsprechenden Datentyp `NarrowingGraph` und eine Funktion, die das Berechnen eines solchen Graphen umsetzt:

```
data NarrowingGraph f
  = NGraph (Term f) [(Pos, Subst f, NarrowingGraph f)]

narrowingGraphBy :: NStrategy f -> TRS f -> Int -> Term f
                 -> NarrowingGraph f
```

Ein derartiger Graph ist dabei so zu begreifen, dass ein Term jeden aus einem Narrowing-Schritt resultierenden Term als Nachfolger hat. Diese haben jeweils wieder Nachfolger, bis eine bestimmte Tiefe erreicht ist. Es stellt sich jedoch auch hier die Frage nach einer anschaulichen grafischen Aufbereitung der gewählten Darstellung. Die Lösung finden wir bei den definierenden Bäumen, wo wir bereits eine Transformation mit `dot` bereitstellen. Analog definieren wir daher folgende Operationen:

```
dotifyNarrowingGraph :: (f -> String) -> NarrowingGraph f -> String
```

```
writeNarrowingGraph :: (f -> String) -> NarrowingGraph f -> String
                    -> IO ()
```

Nachdem wir nun das *Narrowing* auf Termen umfänglich implementiert haben, benötigen wir zuletzt eine Möglichkeit, Gleichungen zu lösen. Der Ansatz ist hier, eine Substitution σ zu finden, sodass eine Anwendung von σ auf beide Seiten zu zwei gleichen Termen führt. Insbesondere nutzen wir Unifikation, um eine solche Substitution zu finden, falls zwei Terme bereits unifizierbar sind. Ist dies nicht der Fall, so wenden wir gemäß der gewählten Strategie einen Narrowing-Schritt auf eine der beiden Seiten an. Jetzt beginnen wir erneut von vorne und verknüpfen die Substitution des Narrowing-Schrittes mit der nächsten Substitution. Wir erhalten also nach mehreren Durchläufen eine vollständige Substitution als Lösung der Gleichung, falls eine solche Lösung überhaupt existiert. Diesen Algorithmus fassen wir in eine Funktion `solveEq` zusammen, wobei zusätzlich über bestimmte Optionen die Berechnung gesteuert werden kann:

```
data NOptions f = NOptions { normalize :: Bool,
                             rStrategy :: RStrategy f }

solveEq :: NOptions f -> NStrategy f -> TRS f -> Term f -> [Subst f]
```

Die Optionen dienen dabei dazu, den Term vor der Berechnung weiterer Narrowing-Schritte zu normalisieren (`normalize = True`). Für die Normalisierung wird die mit `rStrategy` angegebene Reduktionsstrategie genutzt. In der Default-Konfiguration, die als `defaultNOptions` implementiert wurde, ist dies die *Parallel Outermost* Strategie, wobei das Normalisieren hier standardmäßig deaktiviert ist.

3.10 Transformation von Curry-Programmen

Termersetzungssysteme bilden die Grundlage für das Rechnen mit Termen und Funktionen. Bis jetzt ist es allerdings nicht möglich, die Funktionsdefinitionen aus Curry-Programmen direkt in Termersetzungssysteme zu überführen. Daher wollen wir in diesem Abschnitt ein Modul `Rewriting.Files` implementieren, sodass eine Transformation zur Verfügung gestellt wird. Um dabei den Inhalt eines Curry-Programmes zunächst in eine zu verarbeitende Form zu bringen, greifen wir auf die in KiCS2 enthaltene Bibliothek `AbstractCurry` zurück. In diesem Zusammenhang sind insbesondere folgende Deklarationen von Bedeutung:

```
type MName = String

type QName = (MName, String)

data CurryProg
  = CurryProg MName [MName] [CTypeDecl] [CFuncDecl] [COpDecl]
```

```
tryReadCurryFile :: String -> IO (Either String CurryProg)
```

Mit der Funktion `tryReadCurryFile` ist es möglich, den Inhalt einer Curry-Datei in den Datentyp `CurryProg` zu transformieren. Der Datentyp besteht dabei aus einem Modulnamen und vier Listen mit Importen, Typdeklarationen, Funktionsdeklarationen und Operatordeklarationen. Die Typsynonyme `MName` und `QName` bezeichnen Modulnamen und qualifizierte Bezeichner. Für die Umwandlung in Termersetzungssysteme sind allerdings nur die Typ- und Funktionsdeklarationen von Interesse.

Zunächst definieren wir uns eigene Typen, die die umgewandelten Daten repräsentieren. Dabei speichern wir die Typinformationen und die Daten der Termersetzungssysteme in einem Tupel mit dem Typ `RWData` ab. Die Typdeklarationen übernehmen wir ohne Änderung aus der `CurryProg`-Darstellung. Für den zweiten Eintrag wählen wir mit `FiniteMap` eine Darstellung, die jedem qualifizierten Funktionsnamen das entsprechende Termersetzungssystem zuordnet. Die Typen lauten insgesamt wie folgt:

```
type RWData = (TRSData, TypeData)

type TRSData = FM QName (TRS QName)

type TypeData = [CTypeDecl]
```

Für die Transformation einer Curry-Datei nach `RWData` definieren wir eine Operation `readCurryProgram` mit folgender Signatur:

```
readCurryProgram :: String -> IO (Either String RWData)
```

Der Rückgabewert ist also entweder die Fehlermeldung, die durch das Einlesen mit `tryReadCurryFile` entsteht, oder ein Tupel der transformierten Daten. Die Umwandlung von `CurryProg` nach `RWData` ist intern mit der folgenden Operation umgesetzt:

```
fromCurryProg :: CurryProg -> RWData
```

Die Funktion wandelt dabei jede Funktionsdeklaration in ein Termersetzungssystem um und ordnet dieses dem qualifizierenden Funktionsbezeicher zu. Für die Umwandlung müssen wir die Datentypen des Moduls `AbstractCurry.Types` entsprechend parsen. Dazu existieren Operationen nach dem Muster `fromXXX`, wobei `XXX` jeweils den Typen `CExpr`, `CFuncDecl`, `CLiteral`, `CPattern`, `CRhs` und `CRule` ohne das vorangestellte C entspricht. Für eine genaue Übersicht dieser Funktionen sei jedoch Abschnitt A.3 zu empfehlen.

4 Fazit

Dieses Kapitel beinhaltet eine Zusammenfassung der erarbeiteten Ergebnisse und eine Bewertung hinsichtlich der ursprünglichen Zielsetzung. Im zweiten und letzten Abschnitt folgt abschließend ein Ausblick auf mögliche Erweiterungen und Verbesserungen der in dieser Arbeit entwickelten Bibliotheken.

4.1 Zusammenfassung und Bewertung

Das Ziel dieser Arbeit war die Implementierung und Bereitstellung von Bibliotheken für das Rechnen mit Termen, Gleichungen und Termersetzungssystemen in der funktional-logischen Programmiersprache Curry. Insbesondere aber waren hier das Reduzieren von Termen mit Reduktionsstrategien, das Berechnen kritischer Paare sowie definierender Bäume und das Lösen von Gleichungen mittels *Narrowing* ein wesentlicher Bestandteil der Aufgabenstellung. Weiterhin war auch die Transformation von Curry-Programmen in Termersetzungssysteme von besonderem Interesse.

Dementsprechend haben wir in Kapitel 2 zunächst die wichtigsten Grundlagen für die Implementierung erläutert. Dazu gehören etwa die Programmiersprache Curry sowie die formalen Definitionen im Zusammenhang mit Termersetzungssystemen. Letztere haben wir im Wesentlichen aus den bereits sehr umfangreichen und detaillierten Notizen zur Vorlesung *Deklarative Programmiersprachen* [Han16b] entnommen.

In Kapitel 3 haben wir dann zunächst die Darstellung für Variablen und Terme eingeführt und darauf aufbauend Regeln sowie Termersetzungssysteme und Positionen definiert. Basierend auf dieser Grundlage konnten dann die Berechnung kritischer Paare, die Reduktion von Termen, Reduktionsstrategien aber auch definierende Bäume und das *Narrowing* implementiert werden. Die formalen Definitionen konnten dabei in einer Vielzahl der Fälle, ohne große Ergänzungen, leicht in Curry-Code überführt werden.

Während der Entwicklung wurden jedoch nicht nur die zuvor definierten Vorgaben umgesetzt, sondern auch bereits kleinere Erweiterungen, wie etwa die anschauliche Ausgabe (*Pretty-Printing*) von zum Beispiel Termen, kritischen Paaren oder Reduktionsschritten, erarbeitet und integriert. Auch die Transformation von definierenden Bäumen und *Narrowing*-Graphen in die Graph-Beschreibungssprache DOT war zu Beginn nicht Bestandteil der ursprünglichen Zielsetzung. Die Ergänzungen, die insgesamt in kurzer Zeit entwickelt wurden, ermöglichen jedoch eine grundlegende Verbesserung der Darstellung der zu verarbeitenden und resultierenden Daten.

4.2 Ausblick

Zuletzt wollen wir einige Erweiterungen und Verbesserungen der Bibliotheken diskutieren. Eine Funktionalität, die eine wesentliche Erweiterung benötigt, ist dabei die Transformation von Curry-Programmen in Termersetzungssysteme. Derzeit wird nur ein kleiner Teil der Curry-Syntax unterstützt. Nicht enthalten ist zum Beispiel die Umformung von `case`-Ausdrücken oder lokaler Deklarationen. Damit wird jedoch, aufgrund der Relevanz in Curry-Programmen, die Nutzbarkeit der Bibliothek `Rewriting.Files` wesentlich eingeschränkt. Die größte Hürde der genannten syntaktischen Elemente ist dabei die notwendige Aufwertung zu eigenständigen und unabhängigen Funktionen (*Lambda Lifting*). Eine weitere Schwierigkeit zeigt sich auch bei einem Vorkommen mehrerer verschiedener `case`-Ausdrücke innerhalb eines Terms, denn es bedarf zur Unterscheidung einer Indizierung beziehungsweise einer geschickten Namensgebung der aufgewerteten `case`-Funktionen. Aufgrund der Komplexität des Themas, insbesondere aber wegen des notwendigen *Lambda Lifting*, war es jedoch nicht möglich diese Erweiterung im zeitlichen Rahmen dieser Arbeit zu implementieren.

Ein weiterer Vorschlag zur Verbesserung, ist die Unterstützung von *Sharing* für das *Narrowing* von Termen. Dabei werden mehrfach auftretende gleiche Teilterme als nur ein einziges Vorkommen angesehen. Der Vorteil ist dabei, dass sich die Anzahl der möglichen *Narrowing*-Ableitungen im besten Fall verringert, da eine Veränderung des geteilten Terms eine Auswirkung auf alle Vorkommen hat. Inwieweit der *Sharing*-Ansatz in die erarbeitete *Narrowing*-Implementierung integriert werden kann, bleibt zu klären.

Abschließend wollen wir noch auf die Tatsache eingehen, dass in Curry im Gegensatz zu Haskell derzeit keine Typklassen existieren. Dieser Umstand hatte zwar keine wesentlichen Auswirkungen auf die Entwicklung der Schnittstellen, dennoch sind damit insbesondere die Funktionalitäten für das *Pretty-Printing*, aufgrund eines weiteren Argumentes, etwas komplizierter gestaltet. Wir benötigen daher immer eine zusätzliche Funktion, die den Konstruktor eines polymorphen Terms in eine Zeichenkette umwandelt. Mit der Einführung von Typklassen und der Verwendung einer einheitlichen `show`-Funktion kann die Implementierung demnach weiter vereinfacht werden.

Zusammenfassend lässt sich feststellen, dass die bisherigen Implementierungen bereits einen großen Umfang an Funktionen bereitstellen, durchaus aber einen Raum für zukünftige Erweiterungen bieten.

A Übersicht der Module

Dieser Anhang bietet eine Übersicht über die exportierten Typen und Funktionen der entwickelten Module. Aufgelistet werden jeweils die Datentyp-Deklarationen und die Typsynonyme aber auch die Signaturen der Funktionen mit den zugehörigen englischen *CurryDoc*-Kommentaren[1] in alphabetischer Reihenfolge. Die exportierten Typen eines Moduls werden dabei stets zuerst aufgeführt.

A.1 Rewriting.CriticalPairs

```
--- A critical pair represented as a pair of terms and parameterized over
--- the kind of function symbols, e.g., strings.
type CPair f = (Term f, Term f)

--- Returns the critical pairs of a term rewriting system.
cPairs :: TRS f -> [CPair f]

--- Checks whether a term rewriting system is orthogonal.
isOrthogonal :: TRS f -> Bool

--- Checks whether a term rewriting system is weak-orthogonal.
isWeakOrthogonal :: TRS f -> Bool

--- Transforms a critical pair into a string representation.
showCPair :: (f -> String) -> CPair f -> String

--- Transforms a list of critical pairs into a string representation.
showCPairs :: (f -> String) -> [CPair f] -> String
```

A.2 Rewriting.DefinitionalTree

```
--- Representation of a definitional tree, parameterized over the kind of
--- function symbols, e.g., strings.
---
--- @cons Leaf r            - The leaf with rule ‘r‘.
--- @cons Branch pat p dts - The branch with pattern ‘pat‘, inductive
---                          position ‘p‘ and definitional subtrees ‘dts‘.
```

[1] https://www-ps.informatik.uni-kiel.de/currywiki/tools/currydoc/

```
--- @cons Or pat dts        - The or node with pattern ‘pat‘ and
---                           definitional subtrees ‘dts‘.
data DefTree f = Leaf (Rule f)
               | Branch (Term f) Pos [DefTree f]
               | Or (Term f) [DefTree f]

--- Returns a list of definitional trees for a term rewriting system.
defTrees :: TRS f -> [DefTree f]

--- Returns a list of definitional trees for a list of term rewriting
--- systems.
defTreesL :: [TRS f] -> [DefTree f]

--- Transforms a definitional tree into a graphical representation by
--- using the *DOT graph description language*.
dotifyDefTree :: (f -> String) -> DefTree f -> String

--- Returns the pattern of a definitional tree.
dtPattern :: DefTree f -> Term f

--- Returns the root symbol (variable or constructor) of a definitional
--- tree.
dtRoot :: DefTree f -> Either VarIdx f

--- Returns the definitional tree with the given index from the given
--- list of definitional trees or the provided default definitional tree
--- if the given index is not in the given list of definitional trees.
fromDefTrees :: DefTree f -> Int -> [DefTree f] -> DefTree f

--- Checks whether a term has a definitional tree with the same
--- constructor pattern in the given list of definitional trees.
hasDefTree :: [DefTree f] -> Term f -> Bool

--- Returns a list of all inductive positions in a term rewriting system.
idtPositions :: TRS f -> [Pos]

--- Returns the position and the definitional trees from the given list
--- of definitional trees for the leftmost outermost defined constructor
--- in a term or ‘Nothing‘ if no such pair exists.
loDefTrees :: [DefTree f] -> Term f -> Maybe (Pos, [DefTree f])

--- The reduction strategy phi. It uses the definitional tree with the
--- given index for all constructor terms if possible or at least the
--- first one.
```

```
phiRStrategy :: Int -> RStrategy f

--- Returns a list of definitional trees with the same constructor
--- pattern for a term from the given list of definitional trees.
selectDefTrees :: [DefTree f] -> Term f -> [DefTree f]

--- Writes the graphical representation of a definitional tree with the
--- *DOT graph description language* to a file with the given filename.
writeDefTree :: (f -> String) -> DefTree f -> String -> IO ()
```

A.3 Rewriting.Files

```
--- Representation of term rewriting system data and type data as a pair.
type RWData = (TRSData, TypeData)

--- Mappings from a function name to the corresponding term rewriting
--- system represented as a finite map from qualified names to term
--- rewriting systems.
type TRSData = FM QName (TRS QName)

--- Information about types represented as a list of type declarations.
type TypeData = [CTypeDecl]

--- Returns the qualified name for an `if-then-else`-constructor.
condQName :: QName

--- Returns the term rewriting system for an `if-then-else`-function.
condTRS :: TRS QName

--- Transforms an abstract curry program into an equivalent
--- representation, where every function gets assigned the corresponding
--- term rewriting system and every type has a corresponding type
--- declaration.
fromCurryProg :: CurryProg -> RWData

--- Transforms an abstract curry expression for the function with the
--- given name into a tuple of a term, a substitution and a term
--- rewriting system.
fromExpr :: QName -> CExpr -> (Term QName, Subst QName, TRS QName)

--- Transforms an abstract curry function declaration into a pair with
--- function name and corresponding term rewriting system.
fromFuncDecl :: CFuncDecl -> (QName, TRS QName)
```

```
--- Transforms an abstract curry literal into a term.
fromLiteral :: CLiteral -> Term QName

--- Transforms an abstract curry pattern for the function with the given
--- name into a pair of a term and a substitution.
fromPattern :: QName -> CPattern -> (Term QName, Subst QName)

--- Transforms an abstract curry right-hand side of a rule for the
--- function with the given name into a tuple of a term, a substitution
--- and a term rewriting system.
fromRhs :: QName -> CRhs -> (Term QName, Subst QName, TRS QName)

--- Transforms an abstract curry rule for the function with the given
--- name into a pair of a rule and a term rewriting system.
fromRule :: QName -> CRule -> (Rule QName, TRS QName)

--- Tries to read and transform a curry program into an equivalent
--- representation, where every function gets assigned the corresponding
--- term rewriting system and every type has a corresponding type
--- declaration.
readCurryProgram :: String -> IO (Either String RWData)

--- Transforms a string into a qualified name.
readQName :: String -> QName

--- Transforms a qualified name into a string representation.
showQName :: QName -> String
```

A.4 Rewriting.Narrowing

```
--- Representation of a narrowing on first-order terms, parameterized
--- over the kind of function symbols, e.g., strings.
---
--- @cons NTerm t          - The narrowed term `t`.
--- @cons NStep t p sub n - The narrowing of term `t` at position `p`
---                         with substitution `sub` to narrowing `n`.
data Narrowing f = NTerm (Term f)
                 | NStep (Term f) Pos (Subst f) (Narrowing f)

--- Representation of a narrowing graph for first-order terms,
--- parameterized over the kind of function symbols, e.g., strings.
---
--- @cons NGraph t ns - The narrowing of term `t` to a new term with a
```

```
---                     list of narrowing steps ‘ns‘.
data NarrowingGraph f
  = NGraph (Term f) [(Pos, Subst f, NarrowingGraph f)]

--- Representation of narrowing options for solving term equations,
--- parameterized over the kind of function symbols, e.g., strings.
---
--- @field normalize - Indicates whether a term should be normalized
---                    before computing further narrowing steps.
--- @field rStrategy - The reduction strategy to normalize a term.
data NOptions f = NOptions { normalize :: Bool,
                             rStrategy :: RStrategy f }

--- A narrowing strategy represented as a function that takes a term
--- rewriting system and a term and returns a list of triples consisting
--- of a position, a rule and a substitution, parameterized over the kind
--- of function symbols, e.g., strings.
type NStrategy f = TRS f -> Term f -> [(Pos, Rule f, Subst f)]

--- The default narrowing options.
defaultNOptions :: NOptions f

--- Transforms a narrowing graph into a graphical representation by using
--- the *DOT graph description language*.
dotifyNarrowingGraph :: (f -> String) -> NarrowingGraph f -> String

--- The innermost narrowing strategy.
imNStrategy :: NStrategy f

--- The lazy narrowing strategy.
lazyNStrategy :: NStrategy f

--- The leftmost outermost narrowing strategy.
loNStrategy :: NStrategy f

--- Narrows a term with the given strategy and term rewriting system by
--- the given number of steps.
narrowBy :: NStrategy f -> TRS f -> Int -> Term f -> [(Subst f, Term f)]

--- Narrows a term with the given strategy and list of term rewriting
--- systems by the given number of steps.
narrowByL :: NStrategy f -> [TRS f] -> Int -> Term f
          -> [(Subst f, Term f)]
```

```
--- Returns a list of narrowings for a term with the given strategy, the
--- given term rewriting system and the given number of steps.
narrowingBy :: NStrategy f -> TRS f -> Int -> Term f -> [Narrowing f]

--- Returns a list of narrowings for a term with the given strategy, the
--- given list of term rewriting systems and the given number of steps.
narrowingByL :: NStrategy f -> [TRS f] -> Int -> Term f -> [Narrowing f]

--- Returns a narrowing graph for a term with the given strategy, the
--- given term rewriting system and the given number of steps.
narrowingGraphBy :: NStrategy f -> TRS f -> Int -> Term f
                 -> NarrowingGraph f

--- Returns a narrowing graph for a term with the given strategy, the
--- given list of term rewriting systems and the given number of steps.
narrowingGraphByL :: NStrategy f -> [TRS f] -> Int -> Term f
                  -> NarrowingGraph f

--- The outermost narrowing strategy.
omNStrategy :: NStrategy f

--- Transforms a narrowing into a string representation.
showNarrowing :: (f -> String) -> Narrowing f -> String

--- Solves a term equation with the given strategy, the given term
--- rewriting system and the given options. The term has to be of the
--- form `TermCons c [l, r]` with `c` being a constructor like `=`. The
--- term `l` and the term `r` are the left-hand side and the right-hand
--- side of the term equation.
solveEq :: NOptions f -> NStrategy f -> TRS f -> Term f -> [Subst f]

--- Solves a term equation with the given strategy, the given list of
--- term rewriting systems and the given options. The term has to be of
--- the form `TermCons c [l, r]` with `c` being a constructor like `=`.
--- The term `l` and the term `r` are the left-hand side and the
--- right-hand side of the term equation.
solveEqL :: NOptions f -> NStrategy f -> [TRS f] -> Term f -> [Subst f]

--- The standard narrowing strategy.
stdNStrategy :: NStrategy f

--- The weakly needed narrowing strategy.
wnNStrategy :: NStrategy f
```

```
--- Writes the graphical representation of a narrowing graph with the
--- *DOT graph description language* to a file with the given filename.
writeNarrowingGraph :: (f -> String) -> NarrowingGraph f -> String
                    -> IO ()
```

A.5 Rewriting.Position

```
--- A position in a term represented as a list of integers greater than
--- zero.
type Pos = [Int]

--- Concatenates two positions.
(.>) :: Pos -> Pos -> Pos

--- The root position of a term.
eps :: Pos

--- Checks whether the first position is above the second position.
isPosAbove :: Pos -> Pos -> Bool

--- Checks whether the first position is below the second position.
isPosBelow :: Pos -> Pos -> Bool

--- Checks whether two positions are disjunct.
isPosDisjunct :: Pos -> Pos -> Bool

--- Checks whether the first position is left from the second position.
isPosLeft :: Pos -> Pos -> Bool

--- Checks whether the first position is right from the second position.
isPosRight :: Pos -> Pos -> Bool

--- Returns a list of all positions in a term.
positions :: Term f -> [Pos]

--- Replaces the subterm of a term at the given position with the given
--- term if the position exists within the term.
replaceTerm :: Term f -> Pos -> Term f -> Term f

--- Transforms a position into a string representation.
showPos :: Pos -> String

--- Returns the subterm of a term at the given position if the position
```

```
--- exists within the term.
(|>) :: Term f -> Pos -> Term f
```

A.6 Rewriting.Rules

```
--- A rule represented as a pair of terms and parameterized over the kind
--- of function symbols, e.g., strings.
type Rule f = (Term f, Term f)

--- A term rewriting system represented as a list of rules and
--- parameterized over the kind of function symbols, e.g., strings.
type TRS f = [Rule f]

--- Checks whether a term has a rule with the same constructor and the
--- same arity in the given term rewriting system.
hasRule :: TRS f -> Term f -> Bool

--- Checks whether a term rewriting system is constructor-based.
isConsBased :: TRS f -> Bool

--- Checks whether the given argument position of a rule is demanded.
--- Returns `True` if the corresponding argument term is a constructor
--- term.
isDemandedAt :: Int -> Rule f -> Bool

--- Checks whether a term rewriting system is left-linear.
isLeftLinear :: TRS f -> Bool

--- Checks whether a term rewriting system is left-normal.
isLeftNormal :: TRS f -> Bool

--- Checks whether a term is a pattern according to the given term
--- rewriting system.
isPattern :: TRS f -> Term f -> Bool

--- Checks whether the first rule is a variant of the second rule.
isVariantOf :: Rule f -> Rule f -> Bool

--- Returns the maximum variable in a rule or `Nothing` if no variable
--- exists.
maxVarInRule :: Rule f -> Maybe VarIdx

--- Returns the maximum variable in a term rewriting system or `Nothing`
```

```
--- if no variable exists.
maxVarInTRS :: TRS f -> Maybe VarIdx

--- Returns the minimum variable in a rule or ‘Nothing‘ if no variable
--- exists.
minVarInRule :: Rule f -> Maybe VarIdx

--- Returns the minimum variable in a term rewriting system or ‘Nothing‘
--- if no variable exists.
minVarInTRS :: TRS f -> Maybe VarIdx

--- Normalizes a rule by renaming all variables with an increasing order,
--- starting from the minimum possible variable.
normalizeRule :: Rule f -> Rule f

--- Normalizes all rules in a term rewriting system by renaming all
--- variables with an increasing order, starting from the minimum
--- possible variable.
normalizeTRS :: TRS f -> TRS f

--- Returns a list without duplicates of all constructors in a rule.
rCons :: Rule f -> [f]

--- Renames the variables in a rule by the given number.
renameRuleVars :: Int -> Rule f -> Rule f

--- Renames the variables in every rule of a term rewriting system by the
--- given number.
renameTRSVars :: Int -> TRS f -> TRS f

--- Returns the root symbol (variable or constructor) of a rule.
rRoot :: Rule f -> Either VarIdx f

--- Returns a list without duplicates of all variables in a rule.
rVars :: Rule f -> [VarIdx]

--- Transforms a rule into a string representation.
showRule :: (f -> String) -> Rule f -> String

--- Transforms a term rewriting system into a string representation.
showTRS :: (f -> String) -> TRS f -> String
```

A.7 Rewriting.Strategy

```
--- Representation of a reduction on first-order terms, parameterized
--- over the kind of function symbols, e.g., strings.
---
--- @cons NormalForm t - The normal form with term ‘t‘.
--- @cons RStep t ps r - The reduction of term ‘t‘ at positions ‘ps‘ to
---                      reduction ‘r‘.
data Reduction f = NormalForm (Term f)
                 | RStep (Term f) [Pos] (Reduction f)

--- A reduction strategy represented as a function that takes a term
--- rewriting system and a term and returns the redex positions where the
--- term should be reduced, parameterized over the kind of function
--- symbols, e.g., strings.
type RStrategy f = TRS f -> Term f -> [Pos]

--- The leftmost innermost reduction strategy.
liRStrategy :: RStrategy f

--- The leftmost outermost reduction strategy.
loRStrategy :: RStrategy f

--- Returns a parallel reduction strategy according to the given ordering
--- function.
parRStrategy :: (Pos -> Pos -> Ordering) -> RStrategy f

--- The parallel innermost reduction strategy.
piRStrategy :: RStrategy f

--- The parallel outermost reduction strategy.
poRStrategy :: RStrategy f

--- Returns the redex positions of a term according to the given term
--- rewriting system.
redexes :: TRS f -> Term f -> [Pos]

--- Reduces a term with the given strategy and term rewriting system.
reduce :: RStrategy f -> TRS f -> Term f -> Term f

--- Reduces a term with the given term rewriting system at the given
--- redex position.
reduceAt :: TRS f -> Pos -> Term f -> Term f
```

```
--- Reduces a term with the given term rewriting system at the given
--- redex positions.
reduceAtL :: TRS f -> [Pos] -> Term f -> Term f

--- Reduces a term with the given strategy and term rewriting system by
--- the given number of steps.
reduceBy :: RStrategy f -> TRS f -> Int -> Term f -> Term f

--- Reduces a term with the given strategy and list of term rewriting
--- systems by the given number of steps.
reduceByL :: RStrategy f -> [TRS f] -> Int -> Term f -> Term f

--- Reduces a term with the given strategy and list of term rewriting
--- systems.
reduceL :: RStrategy f -> [TRS f] -> Term f -> Term f

--- Returns the reduction of a term with the given strategy and term
--- rewriting system.
reduction :: RStrategy f -> TRS f -> Term f -> Reduction f

--- Returns the reduction of a term with the given strategy, the given
--- term rewriting system and the given number of steps.
reductionBy :: RStrategy f -> TRS f -> Int -> Term f -> Reduction f

--- Returns the reduction of a term with the given strategy, the given
--- list of term rewriting systems and the given number of steps.
reductionByL :: RStrategy f -> [TRS f] -> Int -> Term f -> Reduction f

--- Returns the reduction of a term with the given strategy and list of
--- term rewriting systems.
reductionL :: RStrategy f -> [TRS f] -> Term f -> Reduction f

--- The rightmost innermost reduction strategy.
riRStrategy :: RStrategy f

--- The rightmost outermost reduction strategy.
roRStrategy :: RStrategy f

--- Returns a sequential reduction strategy according to the given
--- ordering function.
seqRStrategy :: (Pos -> Pos -> Ordering) -> RStrategy f

--- Transforms a reduction into a string representation.
showReduction :: (f -> String) -> Reduction f -> String
```

A.8 Rewriting.Substitution

```
--- A substitution represented as a finite map from variables to terms
--- and parameterized over the kind of function symbols, e.g., strings.
type Subst f = FM VarIdx (Term f)

--- Applies a substitution to the given term.
applySubst :: Subst f -> Term f -> Term f

--- Applies a substitution to both sides of the given term equation.
applySubstEq :: Subst f -> TermEq f -> TermEq f

--- Applies a substitution to every term equation in the given list.
applySubstEqs :: Subst f -> TermEqs f -> TermEqs f

--- Composes the first substitution `phi` with the second substitution
--- `sigma`. The resulting substitution `sub` fulfills the property
--- `sub(t) = phi(sigma(t))` for a term `t`. Mappings in the first
--- substitution shadow those in the second.
composeSubst :: Subst f -> Subst f -> Subst f

--- The empty substitution.
emptySubst :: Subst f

--- Extends a substitution with a new mapping from the given variable to
--- the given term. An already existing mapping with the same variable
--- will be thrown away.
extendSubst :: Subst f -> VarIdx -> Term f -> Subst f

--- Returns a substitution that contains all the mappings from the given
--- list. For multiple mappings with the same variable, the last
--- corresponding mapping of the given list is taken.
listToSubst :: [(VarIdx, Term f)] -> Subst f

--- Returns the term mapped to the given variable in a substitution or
--- `Nothing` if no such mapping exists.
lookupSubst :: Subst f -> VarIdx -> Maybe (Term f)

--- Returns a new substitution with only those mappings from the given
--- substitution whose variable is in the given list of variables.
restrictSubst :: Subst f -> [VarIdx] -> Subst f

--- Transforms a substitution into a string representation.
showSubst :: (f -> String) -> Subst f -> String
```

A.9 Rewriting.Term

```
--- Representation of a first-order term, parameterized over the kind of
--- function symbols, e.g., strings.
---
--- @cons TermVar v      - The variable term with variable `v`.
--- @cons TermCons c ts - The constructor term with constructor `c` and
---                       argument terms `ts`.
data Term f = TermVar VarIdx | TermCons f [Term f]

--- A term equation represented as a pair of terms and parameterized over
--- the kind of function symbols, e.g., strings.
type TermEq f = (Term f, Term f)

--- Multiple term equations represented as a list of term equations and
--- parameterized over the kind of function symbols, e.g., strings.
type TermEqs f = [TermEq f]

--- A variable represented as an integer greater than or equal to zero.
type VarIdx = Int

--- Checks whether the constructor pattern of the first term is equal to
--- the constructor pattern of the second term. Returns `True` if both
--- terms have the same constructor and the same arity.
eqConsPattern :: Term f -> Term f -> Bool

--- Checks whether a term is a constructor term.
isConsTerm :: Term f -> Bool

--- Checks whether a term is a ground term (contains no variables).
isGround :: Term f -> Bool

--- Checks whether a term is linear (contains no variable more than
--- once).
isLinear :: Term f -> Bool

--- Checks whether a term is normal (behind a variable is not a
--- constructor).
isNormal :: Term f -> Bool

--- Checks whether a term is a variable term.
isVarTerm :: Term f -> Bool

--- Transforms a term by applying a transformation on all constructors.
```

```
mapTerm :: (a -> b) -> Term a -> Term b

--- Returns the maximum variable in a term or `Nothing` if no variable
--- exists.
maxVarInTerm :: Term f -> Maybe VarIdx

--- Returns the minimum variable in a term or `Nothing` if no variable
--- exists.
minVarInTerm :: Term f -> Maybe VarIdx

--- Normalizes a term by renaming all variables with an increasing order,
--- starting from the minimum possible variable.
normalizeTerm :: Term f -> Term f

--- Renames the variables in a term by the given number.
renameTermVars :: Int -> Term f -> Term f

--- Transforms a term into a string representation.
showTerm :: (f -> String) -> Term f -> String

--- Transforms a term equation into a string representation.
showTermEq :: (f -> String) -> TermEq f -> String

--- Transforms a list of term equations into a string representation.
showTermEqs :: (f -> String) -> TermEqs f -> String

--- Transforms a variable into a string representation.
showVarIdx :: VarIdx -> String

--- Returns a list without duplicates of all constructors in a term.
tCons :: Term f -> [f]

--- Returns a list of all constructors in a term. The resulting list may
--- contain duplicates.
tConsAll :: Term f -> [f]

--- Returns a term with the given constructor and no argument terms.
tConst :: f -> Term f

--- Returns an infix operator term with the given constructor and the
--- given left and right argument term.
tOp :: Term f -> f -> Term f -> Term f

--- Returns the root symbol (variable or constructor) of a term.
```

```
tRoot :: Term f -> Either VarIdx f

--- Returns a list without duplicates of all variables in a term.
tVars :: Term f -> [VarIdx]

--- Returns a list of all variables in a term. The resulting list may
--- contain duplicates.
tVarsAll :: Term f -> [VarIdx]
```

A.10 Rewriting.Unification

```
--- Representation of a unification error, parameterized over the kind of
--- function symbols, e.g., strings.
---
--- @cons Clash t1 t2    - The constructor term ‘t1‘ is supposed to be
---                        equal to the constructor term ‘t2‘ but has a
---                        different constructor.
--- @cons OccurCheck v t - The variable ‘v‘ is supposed to be equal to
---                        the term ‘t‘ in which it occurs as a subterm.
data UnificationError f = Clash (Term f) (Term f)
                        | OccurCheck VarIdx (Term f)

--- Transforms a unification error into a string representation.
showUnificationError :: (f -> String) -> UnificationError f -> String

--- Checks whether a list of term equations can be unified.
unifiable :: TermEqs f -> Bool

--- Unifies a list of term equations. Returns either a unification error
--- or a substitution.
unify :: TermEqs f -> Either (UnificationError f) (Subst f)
```

A.11 Rewriting.UnificationSpec

```
--- Representation of a unification error, parameterized over the kind of
--- function symbols, e.g., strings.
---
--- @cons Clash t1 t2    - The constructor term ‘t1‘ is supposed to be
---                        equal to the constructor term ‘t2‘ but has a
---                        different constructor.
--- @cons OccurCheck v t - The variable ‘v‘ is supposed to be equal to
---                        the term ‘t‘ in which it occurs as a subterm.
data UnificationError f = Clash (Term f) (Term f)
```

```
                        | OccurCheck VarIdx (Term f)

--- Transforms a unification error into a string representation.
showUnificationError :: (f -> String) -> UnificationError f -> String

--- Checks whether a list of term equations can be unified.
unifiable :: TermEqs f -> Bool

--- Unifies a list of term equations. Returns either a unification error
--- or a substitution.
unify :: TermEqs f -> Either (UnificationError f) (Subst f)
```

Literaturverzeichnis

[AEH00] ANTOY, Sergio ; ECHAHED, Rachid ; HANUS, Michael: A Needed Narrowing Strategy. In: *Journal of the ACM* 47 (2000), Nr. 4, S. 776–822

[GKN15] GANSNER, Emden R. ; KOUTSOFIOS, Eleftherios ; NORTH, Stephen: *Drawing graphs with dot*. `http://www.graphviz.org/`, 2015. – Version vom 5. Januar 2015

[Han16a] HANUS, Michael (Hrsg.): *Curry: An Integrated Functional Logic Language (Version 0.9.0)*. `http://www.curry-language.org/`, 2016

[Han16b] HANUS, Michael: *Notizen zur Vorlesung Deklarative Programmiersprachen*. `https://www.informatik.uni-kiel.de/~mh/lehre/dps15/`, 2016. – Version vom 21. März 2016 im Wintersemester 2015/2016

[KB70] KNUTH, Donald E. ; BENDIX, Peter B.: Simple Word Problems in Universal Algebras. In: LEECH, John (Hrsg.): *Computational Problems in Abstract Algebra*, Pergamon Press, 1970, S. 263–297

[MNRA92] MORENO-NAVARRO, Juan J. ; RODRÍGUEZ-ARTALEJO, Mario: Logic Programming with Functions and Predicates: The Language BABEL. In: *The Journal of Logic Programming* 12 (1992), Nr. 3, S. 191–223