Übersetzung von Curry nach Java

Bastian Kirchmayr

Masterarbeit
eingereicht im Juni 2017

Christian-Albrechts-Universität zu Kiel
Programmiersprachen und Übersetzerkonstruktion

Betreuung: Prof. Dr. Michael Hanus und Dipl. Inf. Jan Tikovsky

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Kiel, den 14. Juni 2017

_____
(Name des Kandidaten)

Zusammenfassung

Wir beschreiben wie Curry in Java implementiert werden kann. Dabei werden Berechnungen nicht direkt in Java durchgeführt, stattdessen erzeugt der Java-Code eine Graphstruktur, die den zu berechnenden Ausdruck beschreibt. Diese Art der Darstellung erlaubt es Sharing als Teilen von Knoten im Graphen zu implementieren. Nicht-Determinismus wird dabei als Datenstruktur dargestellt und erlaubt auch Sharing zwischen verschiedenen nicht-deterministischen Auswahlzweigen.

Dazu wird Java-Code generiert, der zu jeder Curry-Funktion den Graphen in einer Art transformiert, die der Funktion entspricht. Damit ist ein Curry-Programm in Java bzw. allgemeiner auf der Java Virtual Machine (JVM) ausführbar.

Wir geben eine Implementierung an, die in der Lage ist dieses praktisch durchzuführen. Sie ist in Curry geschrieben und kann selbst nach Java übersetzt und dort ausgeführt werden. Weiterhin erlaubt sie es, aus natürlichem Java-Code heraus Curry-Funktionen nutzen.

Inhaltsverzeichnis

1. Einführung

Curry wurde entwickelt, um die funktionalen Konzepte einer Sprache wie Haskell um logische Aspekte zu erweitern. Diese erlauben es, Probleme, für die ein Suchraum nach einem Ergebnis durchsucht werden muss, zu formulieren. Das ist etwa für Constraint-Programmierung nützlich, bei der Lösungen gesucht werden, die bestimmte Beschränkungen erfüllen. Dies kann die beste Lösung sein, aber auch alle möglichen Lösungen.

Bestehende Implementierungen von Curry sind etwa PAKCS und KiCS2[He16], die Prolog bzw. Haskell nutzen. (`http://curry-language.org/`)

1.1. Motivation

Eine Implementierung in Java bietet sich an, da die JVM auf vielen Plattformen installiert ist. Diese würden damit auch eine Ausführung von Curry-Programmen erlauben.

Eine Implementierung von Curry in Java kann außerdem dazu genutzt werden, eine Brücke zwischen den beiden Sprachen zu schaffen. Probleme, die sich leicht in Curry lösen lassen können dann in Curry formuliert und von einem Java-Programm genutzt werden.

Auch eine Nutzung von Java-Code innerhalb von Curry-Programmen ist denkbar. Damit könnte Curry von der großen Anzahl existierender Java-Bibliotheken profitieren.

1.2. Ziele

Ziel der Arbeit ist es, einen Übersetzer zu entwickeln, der es erlaubt Curry-Programme in einer JVM auszuführen.

Die Curry-Bibliothek enthält eine Reihe von extern definierten Funktionen. Von diesen sollten die Wichtigsten implementiert werden. Insbesondere sollten alle Funktionalitäten der Curry-Prelude funktionieren. Weiterhin sollte der in Curry implementierte Übersetzer selbst in der JVM laufen können.

Es sollte für Java-Programme auf einfache und natürliche Weise möglich sein, Curry-Funktionen auszuführen. Dabei sollten wichtige Curry-Datenstrukturen automatisch zwischen ihren Curry-Darstellungen und passenden Java-Darstellungen

konvertiert werden.

Nicht-Determinismus und Datentypen aus der Prelude sollten in bekannte Java-Strukturen konvertiert werden, sofern solche existieren. Ansonsten sollten passende Strukturen für Java angeboten werden.

Zielsprache sollte Java bis Version 7 sein.

1.3. Verwandte Arbeiten

Es gibt bereits einen Versuch Curry in Java zu implementieren [HS97]. Dieser basiert auf einer abstrakten Maschine. Curry-Code wird in Code dieser Maschine übersetzt und dieser ausgeführt.

Nicht-Determinismus wird nicht als Datenstruktur explizit dargestellt, sondern verschiedene Ergebnisse werden nebenläufig berechnet.

Die Implementierung verwendet keinen reinen Java-Code sondern die Java-Erweiterung Pizza [OW97].

SPRITE [AJ16] übersetzt Curry-Programme in Assembler-Programme. Im Unterschied zu anderen Implementierungen ist eine Suchstrategie für Ergebnisse vorgegeben. Diese Fair Scheme [AJ13] genannte Strategie soll die Nachteile anderer Strategien vermeiden. Sie ist vollständig und versucht verschiedene Ergebnisse gleichberechtigt zu finden.

[Han07] beschreibt einen Ansatz Curry-Code mit HTML und JavaScript zu verbinden. Dies soll dazu dienen, Aspekte der Web-Programmierung in Curry zu beschreiben. Trotz seines Titels versucht die Arbeit nicht Curry-Code in JavaScript zu übersetzen. Der Ansatz Brücken zwischen Curry und JavaScript zu bauen ähnelt aber unserem Vorgehen Curry und Java zu verbinden.

1.4. Struktur

Nach dieser Einführung gehen wir im zweiten Kapitel auf die Grundlagen ein, die nötig sind, um unseren Ansatz zu verstehen. Diese bestehen aus einer kurzen Einführung in Java und Curry, aber auch aus Beschreibungen von Unifikation und Suchstrategien. Diese sind notwendig, um Ergebnisse in Curry berechnen zu können.

Im dritten Kapitel *Ideen* gehen wir auf die verwendeten Ideen ein, wie die Übersetzung und Ausführung von Curry funktionieren soll. Insbesondere beschreiben wir die Darstellung von Nicht-Determinismus als Datenstruktur und darauf aufbauende Konzepte, wie Pull-Tabbing.

Im Kapitel *Implementierung* beschreiben wir die Zwischensprache *ICurry*, in die wir Curry zunächst übersetzen. Danach gehen wir auf die Übersetzung von *ICurry* in Java-Code ein. Weiterhin beschreiben wir, wie die Graphdarstellung in Java behandelt wird, um konkrete Ergebnisse zu produzieren.

Danach zeigen wir im Kapitel *Evaluation*, wie sich die Implementation im Vergleich zu den anderen Implementierungen PAKCS und KiCS2 verhält.

Am Ende fassen wir die Ergebnisse zusammen und geben einen Ausblick, welche Möglichkeiten sich aus dieser Arbeit ergeben können bzw. welche Probleme noch bestehen.

Im Anhang findet sich die Dokumentation zu unserer Implementierung *Cam*.

2. Grundlagen

2.1. Java

Java ist eine imperative, objektorientierte Sprache. Sie ist 1995 erschienen und wurde von Sun Microsystems entwickelt. Java-Programme werden in Bytecode übersetzt und in einer virtuellen Maschine ausgeführt, der Java Virtual Machine (JVM).

Sie ist auf vielen Platformen verbreitet, unter anderem durch den Erfolg des Smartphone-Betriebssystems *Android* auf vielen Mobilgeräten. Aber auch im Server- und Desktopbereich ist sie gängig.

Java hat einen durchgehend objektorientierten Ansatz. Programme sind in Klassen organisiert, die wiederum in hierarchisch angeordneten Paketen liegen. Aus diese Klassen werden durch Instanziierung Objekte erzeugt. In den Klassen werden die Daten gekapselt und es werden Methoden auf den Objekten angeboten. Diese Methoden arbeiten auf den Daten des Objekts.

Das Typsystem von Java ist statisch und stark typisiert.

```
1  public class Hello {
2
3    private String name;
4
5    public Hello(String n) {
6      name = n;
7    }
8
9    public void say() {
10     System.out.println("Hello " + name);
11   }
12
13   public static void main(String[] args) {
14     Hello h = new Hello("world");
15     h.say();
16   }
17 }
```

Das angegebene Programm ist eine Klasse *Hello*, deren Aufgabe es ist, jemanden zu grüßen. Der Name des zu Grüßenden wird im Attribut *name* gespeichert.

Um eine Instanz der Klasse zu erstellen, wird der Konstruktor in Zeile 5 aufgerufen, der den übergebenen Namen im Attribut *name* speichert. Die Instanz hat also den Namen als Zustand.

Zusätzlich bietet sie in Zeile 9 bis 11 eine Methode *say* an, die einen Gruß an den Namen ausgibt.

Die Methode *main* ist keine Methode der Instanz, sondernd der Klasse selbst. Dies wird durch das Schlüsselwort *static* ausgedrückt. Die Methode *main* ist die Hauptmethode, die aufgerufen wird, wenn das Programm ausgeführt wird. In Zeile 14 und 15 erzeugt *main* eine Instanz der Klasse *Hello* mit Parameter "world" und führt die Methode *say* aus.

Ausführung der Klasse *Hello* führt also zu der Ausgabe.

```
> Hello world
```

2.2. Curry

Curry ist eine funktional-logische Sprache mit verzögerter Auswertung (Laziness). [He16] Sie basiert syntaktisch und inhaltlich vor allem auf Haskell.

Programme sind in Module aufgeteilt und bestehen aus Datentypen mit Konstruktoren und Funktionen. Datentypen werden verwendet, um Daten darzustellen und in Funktionen zu verwenden.

Funktionen erlauben Pattern-Matching auf Konstruktoren. Dabei können Funktionen mehrere Regeln enthalten. Abhängig davon, welche Pattern zu den übergebenen Argumenten passen, werden Funktionsaufrufe durch rechte Seiten der Regeln ersetzt. Ein Programm auszuführen heißt also vor allem, Funktionensaufrufe zu ersetzen.

Das folgende Beispielmodul *Test* enthält einen Datentyp **Maybe** a, der es erlaubt einen oder keinen Wert zu enthalten. Aufbauend darauf gibt die Funktion *isNothing* zurück, ob das übergebene Argument keinen Wert enthält. Dabei besteht *isNothing* aus zwei Regeln, die jeweils mit Pattern-Matching überprüfen, ob das Argument *Nothing* oder *Just* ist.

```
module Test where

data Maybe a = Nothing | Just a

isNothing :: Maybe a -> Bool
isNothing Nothing = True
isNothing (Just _) = False
```

2.2.1. Ausdrücke

Ausdrücke sind ein Grundelement von Curry-Programmen. Alle Reduktionen arbeiten auf Ausdrücken und die rechten Seiten von Funktionen sind Ausdrücke. Daher beschreiben wir hier Ausdrücke einmal explizit als BNF.

Expr	::=	*ConsSymb* {*Expr*}	(Konstruktoraufruf)
	|	*FuncSymb* {*Expr*}	(Funktionsaufruf)
	|	*Var*	(Parameter oder lokale Variable)
	|	*Literal*	(Literalwert)
	|	let {*Def*} in *Expr*	(lokale Definitionen)
	|	free {*Var*} in *Expr*	(freie Variablen)
	|	case *Expr* of {*Branch*}	(Fallunterscheidung)
	|	*Expr* ? *Expr*	(nicht-deterministische Wahl)
Def	::=	*Var Expr*	(lokale Definition)
Branch	::=	*Pattern Expr*	(Fall in Fallunterscheidung)
Pattern	::=	*ConsSymb* {*Pattern*}	(Konstruktorpattern)
	|	*Literal*	(Literalpattern)

Konstruktoraufrufe bestehen aus einem Konstruktorsymbol mit einer Stelligkeit. Ebenso haben Funktionssymbole aus Funktionsaufrufen eine Stelligkeit. Konstruktorsymbole und Funktionssymbole stammen aus einer Menge von Symbolen.

Variablen aus der Menge der Variablen Var haben jeweils Gültigkeit innerhalb einer Funktion oder einer lokalen Definition *let* oder $free$.

Literale sind Grundwerte wie Zahlen und werden hier nicht explizit definiert.

Typausdrücke beschreiben Typen von Elementen in Curry. So haben beispielsweise Funktionen einen Typ, der oft explizit angegeben wird.

TypeExpr	::=	*Var*	(Typvariable)
	|	*TypeSymb* {*TypeExpr*}	(Typ eines Datentyps)
	|	*TypeExpr* → *TypeExpr*	(Funktionstyp)

Typvariablen werden in Funktionen oder Datentypen als Typparameter eingeführt. Diese stehen für noch nicht festgelegte Typausdrücke.

Ähnlich wie Konstruktorsymbole für Ausdrücke, sind Typsymbole aus einer Menge der Typsymbole und haben eine Stelligkeit. Hier steht die Stelligkeit allerdings für die Anzahl der Typparameter.

Ein Funktionstyp steht für eine Funktion mit einem Eingabe- und Rückgabetypen.

Notation : Wir geben im folgenden mehrmals Gleichungen von Ausdrücken der Art id 42 = 42 an. Dies ist die Gleichheit bezüglich Zusammenführbarkeit durch Reduktion und ist kein Teil der Syntax von Curry.

2.2.2. Logische Anteile

Zusätzlich zu den funktionalen Eigenschaften, enthält Curry auch noch Elemente der logischen Programmierung. Ein Wert in Curry kann nicht-deterministisch sein, also mehrere mögliche Werte enthalten. Eine einfache Möglichkeit dies auszudrücken ist der ?-Operator.

? ist eine Funktion, die zwei Parameter bekommt und nicht-deterministisch einen der beiden Werte zurückgibt.

```
coin :: Int
coin = 0 ? 1
```

coin steht für die Werte 0 und 1, kann also als eine Funktion verstanden werden, die einen Münzwurf modelliert.

Abgesehen von ? gibt es zwei Wege in Curry Nicht-Determinismus auszudrücken.

Überlappende Regeln

Der erste Weg sind überlappende Regeln. Bei einer Funktion mit mehr als einer Regel kann es Parameterkombinationen geben, die zu mehreren linken Seiten passen.

```
oneOf :: [Maybe a] -> a
oneOf (Just x : _)  = x
oneOf (_ : xs)      = oneOf xs
```

Die Funktion *oneOf* soll aus einer Liste von *Maybe*-Werten nicht-deterministisch einen Wert zurückgeben. So soll etwa gelten `oneOf [Just 0, Nothing, Just 1] = 0 ? 1`.

Für ein Listenelement *Nothing* kann nur die zweite Regel angewandt werden. Für eine Eingabe `Just 0 : xs` können dagegen beide Regeln verwendet werden. In Haskell wird nur die erste passende Regel angewandt, aber in Curry können beide Regeln nicht-deterministisch genutzt werden. Das heißt beide Ergebnisse sind gültig, es gilt `oneOf (Just 0 : xs) = Just 0 ? oneOf xs`.

Diese Art Nicht-Determinismus zu formulieren, erlaubt es Fälle mit mehreren Auswahlmöglichkeiten direkt auszudrücken. So lässt sich die Funktion *coin* auch auf diese Weise implementieren.

```
coin :: Int
coin = 0
coin = 1
```

Da *coin* gar keine Parameter hat, sind automatisch beide Regeln anwendbar und sowohl 0, als auch 1 sind gültige Ergebnisse.

Freie Variablen

Die zweite Möglichkeit Nicht-Determinismus auszudrücken sind freie Variablen. Eine freie Variable hat einen Typ, wie jede andere Variable, kann aber jeden möglichen Wert dieses Typs annehmen.

```
coin :: Bool
coin = b
  where b free
```

Die Funktion ist wieder eine Implementierung von *coin*. Allerdings wird hier ein boolscher Wert als Darstellung eines Münzwurfs gewählt, statt einer Zahl.

b wird als freie Variable vom Typ *Bool* mit dem Schlüsselwort *free* eingeführt. Damit kann *b* nicht-deterministisch jedem Wert vom Typ *Bool* annehmen, nämlich *True* und *False*. Daraus ergibt sich für Wahrheitswerte `b where b free = True ? False`.

Constraints

Die möglichen Werte von freien Variablen lassen sich durch Constraints beschränken. Das Gleichheits-Constraint =:= vergleicht beide Seiten.

Für Ausdrücke ohne Nicht-Determinismus bedeutet ein Gleichheits-Constraint, dass beide Seiten aus denselben Konstruktoren bestehen müssen.

So gilt das Constraint `Just 42 =:= Just 42`, das Constraint `Left 42 =:= Left 23` aber nicht. Anders als ein boolscher Vergleich ist das Ergebnis eines fehlgeschlagenen Constraints nicht *False*, sondernd ein Fehlschlag. Ein Fehlschlag bedeutet, dass das gesamte Ergebnis, das von dem Constraint abhängt, ungültig ist. Constraints werden in Curry meist in Guards genutzt.

```
assertTrue :: Bool -> Bool
assertTrue b | b =:= True = b
```

Der Ausdruck `map assertTrue [True, True]` ergibt `[True, True]`, da das Constraint `True =:= True` jeweils gültig ist.

Dagegen ergibt `map assertTrue [True, False]` kein Ergebnis, da das Constraint `False =:= True` fehlschlägt und somit das gesamte Ergebnis ungültig ist.

Ein ungültiges Ergebnis bedeutet allerdings nicht, dass ein Ausdruck gar kein Ergebnis hat. Bei mehreren nicht-deteministischen Ergebnissen, können auch nur einzelne fehlschlagen. So produziert etwa der Ausdruck `assertTrue (True ? False)` ein Ergebnis *True*. *False* dagegen ist fehlgeschlagen, da `False =:= True` fehlschlägt.

Freie Variablen werden in einem Gleichheits-Constraints nicht verglichen, sondern gebunden. Das Binden einer freien Variable an einen Konstruktorausdruck bedeutet, dass die freie Variable den Wert des Konstruktors annimmt. Das Binden einer freien Variable an eine andere freie Variable bedeutet, dass beide freie Variablen denselben Wert annehmen müssen. Die Berechnung der Werte von freien Variablen erfolgt durch Unifikation.

Am Beispiel von *last* zeigen wir eine Einsatzmöglichkeit von freien Variablen. Es soll das letzte Element einer Liste berechnet werden, ohne die Liste explizit zu durchsuchen. Dazu wird der Parameter *xs* an den Ausdruck `pre ++ [x]` gebunden,

muss also dieselbe Form annehmen. Da *pre* und x freie Variablen sind, können sie beliebige Werte (ihres Typs) annehmen.

Nur wenn pre = **init** xs und x das letzte Element von xs ist, kann das Constraint erfüllt sein. Somit ist das einzig mögliche Ergebnis das letze Element von xs.

```
last :: [a] -> a
last xs | xs =:= pre ++ [x] = x
  where pre,x free
```

2.2.3. Unifikation

Es sei eine Menge von Variablen V und eine Menge von Konstruktorsymbolen K gegeben. Dabei habe jedes Konstruktorsymbol $k \in K$ eine Stelligkeit $n_k \in \mathbb{N}$.

Ein Ausdruck sei von der Form

$$\begin{array}{lll} E & ::= \text{ v} & v \in V \\ & \mid \text{ k } \{E\} & k \in K, n_k \text{ Anzahl von Parametern} \end{array}$$

Die Menge aller Ausdrücke sei E.

Eine Substitution sei eine Funktion $\sigma : E \to E$ geschrieben als

$$\sigma = \{v_1 \to e_1, \cdots, v_n \to e_n\}$$

mit $e_1, \cdots, e_n \in E, v_1, \cdots, v_n \in V, n \in \mathbb{N}$.

Dabei gilt für σ:

$$\sigma(v \in V) = \begin{cases} e_i & \text{falls ein } i \in \mathbb{N} \text{ existiert mit } v = v_i \\ v & \text{sonst} \end{cases}$$

$$\sigma(k\ e_1 \cdots e_{n_k}) = k\ \sigma(e_1) \cdots \sigma(e_{n_k})$$

mit $k \in K, e_1, \cdots, e_{n_k} \in E$.

Eine Substitution ist ein Unifikator für zwei Ausdrücke $e_1, e_2 \in E$, falls gilt

$$\sigma(e_1) = \sigma(e_2)$$

e_1 und e_2 heißen dann unifizierbar.

Diese Definiton lässt sich auf Mengen von Ausdrucksgleichungen erweitern. Eine Substitution ist ein Unifikator für eine Menge von Ausdrucksgleichungen $M = \{e_l \doteq e_r | e_l, e_r \in E\}$, falls gilt

$$\sigma(e_l) = \sigma(e_r)$$

für alle $e_l \doteq e_r \in M$.

σ ist der allgemeinster Unifikator einer Menge von Ausdrucksgleichungen, falls es für jeden Unifikator σ' dieser Menge einen Unifikator σ'' gibt, so dass gilt

$$\sigma' = \sigma'' \circ \sigma$$

Nach [Rob65] existiert für alle unifizierbaren Ausdrücke $e_1, e_2 \in E$ ein effektiv berechenbarer, allgemeinster Unifikator.

2.2.4. Sharing von Nicht-Determinismus

Eine nicht-deterministische Wahl kann durch Binden an Variablen in einen Ausdruck mehrfach vorkommen.

```
coin :: Int
coin = 0 ? 1

twoCoins :: Int
twoCoins = v + v
  where
    v = coin
```

Die Funktion *twoCoins* bindet den Wert des Münzwurfs *coin* an v. Mögliche Ergebnisse eines Aufrufs von *twoCoins* sind 0 und 2. Aber ist auch 1 ein mögliches Ergebnis?

Das hängt von der Art ab, wie die nicht-deterministische Wahl in *coin* verstanden wird. Wir unterscheiden zwischen zwei Arten: der Run-Time-Choice und der Call-Time-Choice [Hus88].

Bei der Run-Time-Choice wird die Wahl einer Alternative möglichst lange hinausgezögert. Jedesmal, wenn der Wert tatsächlich genutzt wird, wird eine Entscheidung getroffen, nicht früher. Insofern passt diese Vorgehensweise am ehesten zur Strategie der verzögerten Auswertung.

Allerdings ist sie nicht sehr intuitiv, wie am Beispiel von *twoCoins* zu erkennen ist. Da die Entscheidung verzögert wird, reduziert sich *twoCoins* zu `(0 ? 1) + (0 ? 1)`. Die beiden Wahlen sind damit voneinander unabhängig und 1 ist ein weiteres mögliches Ergebnis. Jemand, der die Definition von *twoCoins* liest, hat das vermutlich nicht erwartet.

Bei Wahlen mit Call-Time-Choice-Semantik wird die Entscheidung dagegen sofort getroffen, sobald ein Wert, der Nicht-Determinismus enhält, gebunden wird. Damit werden beim Sharing auch Entscheidungen geteilt.

Im Beispiel *twoCoins* heißt das, dass die Entscheidung `0 ? 1` schon beim Binden `v = coin` getroffen wird. Somit sind 0 und 1 die einzig möglichen Ergebnisse.

Ein Vorgehen genau wie beschrieben hat aber den Nachteil, dass verzögerte Auswertung kaum noch möglich ist. Wenn alle Werte auf zu treffende Entscheidungen durchsucht werden müssen, müssten alle Werte sofort ausgerechnet werden.

Daher wird nicht wirklich sofort eine Entscheidung getroffen, stattdessen müssen alle geteilten Wahlen als gleich markiert werden. Dazu bekommt jede Wahl einen Identifikator, der beim Sharing kopiert wird. Sobald eine Entscheidung benötigt wird, muss darauf geachtet werden, dass für alle Wahlen desselben Identifikators dieselbe Entscheidung getroffen wird.

Somit ließe sich *twoCoins* auch als $(0\ ?_1\ 1) + (0\ ?_1\ 1)$ beschreiben. Hier steht $?_1$ für eine Wahl mit Identifikator 1.

Dabei ist zu beachten, dass die referenzielle Transparenz nicht mehr vollständig gilt. Denn *twoCoins* produziert kein Ergebnis 1, aber die alternative Implementierung *twoCoins′* durchaus. Denn in *twoCoins′* handelt es sich um zwei unabhängige Wahlen zwischen 0 und 1.

```
twoCoins' :: Int
twoCoins' = (0 ? 1) + (0 ? 1)
```

Curry nutzt Call-Time-Choice, allerdings werden dabei Entscheidungen nur in 0-stelligen, nicht-globalen Funktionen geteilt. Während also *twoCoins* 1 nicht als Ergebnis produziert, tun es *twoCoins*2 und *twoCoins*3 durchaus.

```
twoCoins2 :: Int
twoCoins2 = coin + coin

twoCoins3 :: Int
twoCoins3 = v () + v ()
  where
    v () = 0 ? 1
```

2.2.5. Suchstrategien

In unserer Implementierung werden wir Nicht-Determinismus in einer Graphstruktur darstellen. Das Suchen nach Ergebnissen entspricht dann dem Durchsuchen eines Graphen.

Ein Graph sei ein Tupel $G = (V, E)$ mit einer endlichen Menge von Knoten V und und einer Menge von Kanten $E \subseteq V^2$.

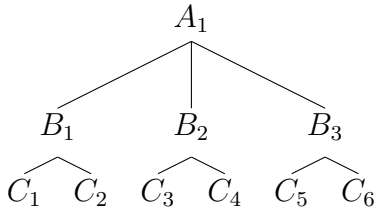
Um einen Graphen zu durchsuchen, gibt es verschiedene Vorgehensweisen. Wir stellen hier zwei vor, nämlich Tiefensuche und Breitensuche.

Tiefensuche

Tiefensuche folgt dem Prinzip, einem Pfad zu folgen, bis dieser endet. Dann auf demselben Pfad möglichst kurz zurückzulaufen und bei der ersten Gelegenheit einen

anderen Pfad zu wählen.

Als Beispiel sei ein Baum mit Knotenbeschriftungen gegeben:



Der erste gewählt Pfad im Graph sei A_1, B_1, C_1. Damit sind die Knoten A_1, B_1 und C_1 besucht. Danach folgt ein Rückschritt und statt C_1 kann C_2 als Pfadelement besucht werden.

Nun muss wieder zurückgeschritten werden, bis eine neue Entscheidung möglich wird. Dies ist erst bei A_1 der Fall. Somit wird als nächstes der Pfad A_1, B_2, C_3 berücksichtigt.

Bis zum Ende durchgeführt ergibt sich für die Reihenfolge der besuchten Knoten $A_1, B_1, C_1, C_2, B_2, C_3, C_4, B_3, C_5, C_6$. Das ist die Reihenfolge, in der Tiefensuche den Beispielbaum durchgehen würde.

Bei Graphen, die nicht kreisfrei sind, ist zusätzlich zu beachten, dass Pfade immer kreisfrei sein müssen und kein Knoten zweimal besucht werden darf. Dies kann in einem Algorithmus *visitor* formuliert werden.

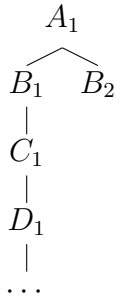
```
procedure visitor(N, G)
begin
  next := empty();
  enqueue(N, next);
  visited := ∅;

  while not empty(next) do
    cur := take(next);
    visit (cur);
    visited := visited ∪ {cur};
    for c in children_G(cur) do
      if c ∉ visited then
        enqueue(c, next);
      fi
    od
  od
end
```

Die Parameter von *visitor* N und G stehen für den Anfangsknoten und den Gesamtgraphen. Die Funktion children_G steht für die Kindknoten eines Knotens in Graph G.

Die Funktionen *empty*, *take* und *enqueue* stehen dabei für Operationen eines Stacks. *empty* ist ein leerer Stack, *take* entspricht *pop* und *enqueue* entspricht *push*.

Tiefensuche ist eine einfache Möglichkeit einen Graphen zu durchsuchen, sie ist allerdings nicht vollständig. Im folgenden Graph wird Tiefensuche nicht alle Knoten besuchen.

Wir gehen hier davon aus, dass Tiefensuche Kindknoten von links nach rechts besucht, für andere, feste Vorgehensweisen gibt es entsprechende Graphen, die die Nicht-Vollständigkeit zeigen.

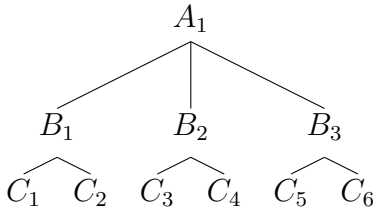


Tiefensuche besucht hier folgende Knoten $A_1, B_1, C_1, D_1, \cdots$, aber niemals den Knoten B_2.

Breitensuche

Breitensuche folgt einem Pfad nicht beliebig lang, sondern folgt ihm im Gegenteil nur ein Element weit. Nachdem ein Pfad um ein Element erweitert wurde, wird ein anderer Pfad gewählt. Dadurch werden alle Knoten in Pfaden von der Wurzel aus besucht, die genau einen Knoten von der Wurzel entfernt sind, dann zwei usw.

Für den Graphen von oben

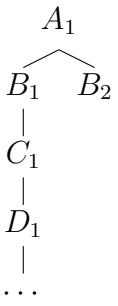


ergibt sich die Besuchsreihenfolge $A_1, B_1, B_2, B_3, C_1, C_2, C_3, C_4, C_5, C_6$.

Als Algorithmus lässt sich Breitensuche exakt so formulieren, wie Tiefensuche. Es müssen nur die Bedeutungen einiger Funktionen verändert werden.

Die Funktion *visitor* beschreibt Breitensuche, wenn wir die Operationen *empty*, *take* und *enqueue* als Operationen einer Warteschlange verstehen. *empty* ist dabei eine leere Warteschlange, *take* entspricht einem *pop* und *enqueue* entspricht *add*. *take* nimmt also Elemente von vorne heraus und *add* fügt Elemente hinten an.

Mit Breitensuche wird der problematische Baum von oben problemlos durchlaufen.



Die Besuchsreihenfolge ist $A_1, B_1, B_2, C_1, D_1, \cdots$.

Ein Nachteil ist, dass sich der Pfad von der Wurzel zum aktuellen Element ständig ändert. Im ersten Beispiel ist der erste genutzte Pfad A_1, der nächste ist A_1, B_1, dann aber A_1, B_2. Dann wird der Pfad A_1, B_1, C_1 genutzt.

Für Anwendungen, in denen der genutzt Pfad den Kontext bestimmt, bedeutet dies, dass sich der Kontext sehr oft ändert. Dies tritt bei Tiefensuche nicht auf.

2.3. FlatCurry

Um Curry-Programme in einem Curry-Programm darzustellen, bietet es sich an, eine Datenstruktur dafür zu definieren. Dazu bietet die Curry-Bibliothek zwei Optionen an: *AbstractCurry* und *FlatCurry*.

AbstractCurry repräsentiert ein vollständiges Quellprogramm, welches dann meist wieder als Quellprogramm ausgegeben wird.

FlatCurry repräsentiert ein Curry-Programm in „flacher“ Form. Das heißt, es hat das Originalprogramm schon vereinfacht, etwa syntaktischen Zucker entfernt. Diese vereinfachte Form bietet sich an, um ein Curry-Programm in eine andere, äquivalente Form zu bringen, etwa um es zu übersetzen.

Das vorhandene Frontend von Curry kann *FlatCurry*-Strukturen generieren. Dieses wird von mehreren Curry-Implementationen genutzt. Verschiedene Backends übersetzen das *FlatCurry*-Programm dann in ihre entsprechenden Zielformate.

Eine Grammatik, die *FlatCurry* beschreibt, ist hier in BNF gegeben.

Prog	::=	*ModName* {*ModName*} {*TypeDecl*} {*FuncDecl*}	
TypeDecl	::=	*TypeName* {*VarIndex*} {*ConsDecl*}	(Datentyp)
	|	*TypeName* {*VarIndex*} *TypeExpr*	(Typsynonym)
ConsDecl	::=	*CombName Arity* {*TypeExpr*}	(Wertkonstruktor)
FuncDecl	::=	*CombName Arity TypeExpr Rule*	(exportierte Funktion)
	|	*CombName Arity Rule*	(private Funktion)

Rule	::=	*ExternName*	(extern definierte Regel)
	|	{*VarIndex*} *Expr*	(einfache Regel)
Expr	::=	*VarIndex*	(Variable)
	|	*Literal*	(Literal)
	|	fCall *QName* {*Expr*}	(Funktionsaufruf)
	|	cCall *QName* {*Expr*}	(Konstruktoraufruf)
	|	fPart *Arity QName* {*Expr*}	(partieller Funktionsaufruf)
	|	cPart *Arity QName* {*Expr*}	(partieller Konstruktoraufruf)
	|	let {*LocalDecl*} *Expr*	(lokale Definition)
	|	free {*VarIndex*} *Expr*	(freie Variablen)
	|	or *Expr Expr*	(Disjunktion)
	|	case *Expr* {*BranchExpr*}	(Fallunterscheidung)
	|	*Expr TypeExpr*	(explizite Typisierung)
BranchExpr	::=	*Pattern Expr*	(Branch eines Cases)
LocalDecl	::=	*VarIndex Expr*	(lokale Definition)
Pattern	::=	*QName* {*VarIndex*}	(Konstruktor-Match)
	|	*Literal*	(Literal-Match)
QName	::=	(*ModName*, *CombName*)	

Verschiedene Nicht-Terminalsymbol sind hier nicht definiert.

- $ModName$ bezeichnet einen Modulnamen.
- $TypeName$ bezeichnet den Namen eines Curry-Typen.
- $CombName$ ist der Name eines Konstruktor- oder Funktionssymbols.
- $ExternName$ ist ein String zur Identifizierung einer externen Funktion.
- *Arity* ist eine Stelligkeit einer Funktion oder eines Konstruktors.
- *Literal* ist eine konstanter Wert, etwa eine Ganzzahl, eine Fließkommazahl oder ein Zeichen.
- Für einen Typausdruck wurde bereits eine Grammatik angegeben, diese entspricht einer $TypeExpr$.

Die Struktur und der Aufbau von $FlatCurry$ entspricht in etwa dem eines Curry-Programms. Die Unterschiede sind gerade die Vereinfachungen, die bei der Transformation eines Curry-Programms vorgenommen werden.

Die wesentlichen Vereinfachungen sind die folgenden:

- Liften von Funktionsdefinitionen: Lokal definierte Funktionen und Lambda-Funktionen werden in globale Funktionen umgewandelt. Dazu müssen die Funktionsnamen eindeutig gemacht werden und genutzte lokale Definitionen als Parameter an die neue Funktion übergeben werden.
- Vereinfachen von Funktionsregeln: Eine Funktionsregel besteht nur noch aus einer Liste von Parametern und einem Ausdruck. Einfaches Pattern-Matching wird in Fallunterscheidungen übersetzt.
- Nicht-lineare Pattern und funktionale Pattern werden durch Guards mit Bindungen (=:<=) ersetzt. Die nicht-strikte Bindung =:<= bindet eine freie Variable lazy an einen Ausdruck.
- Nicht-Determinismus der durch mehrere gültige Regeln eingeführt wurde, wird durch explizite nicht-deterministische *or*-Ausdrücke ersetzt.
- Guards von Funktionen werden durch Aufrufe von *Prelude.cond* ersetzt. *cond* schlägt immer fehl, wenn sein erstes Argument *False* ergibt, sonst verhält es sich wie **flip const**.

Sei eine Funktion *last* im Modul *Last* gegeben.

```
module Last where
  last :: [a] -> a
  last (_ ++ [x]) = x
```

Für dieses Programm sieht die Darstellung in *FlatCurry*-Grammatik wie folgt aus.

```
-- Prog
"Last"
["Prelude"]
[]
[
 --Func
 ("Last","last")
 1
 -- Typparameter a (VarIndex)
 (("Prelude", "[]") a → a)
 -- Rule, Parameter xs (VarIndex)
 ([xs]
  -- freie Variable x (VarIndex)
  (free x
   ( fCall ("Prelude", "cond")
   [ fCall ("Prelude", "=:<=")
     [ fCall ("Prelude", "++")
```

```
        [ fCall ("Prelude", "unknown") []
        , cCall ("Prelude", ":")
          [ x
          , cCall ("Prelude"," []") []
          ]
        ]
      , xs
      ]
    , x
    ]))))]
```

Für das funktionale Pattern wurde eine freie Variable *xs* eingeführt. An diese wurde nicht-strikt der Ausdruck _++[x] gebunden.

Schließlich wurde der Guard in den Aufruf von *cond* verschoben.

3. Ideen

3.1. Termersetzung auf Graphen

Bei der Übersetzung eines Programms von einer Sprache mit Laziness in eine Sprache ohne Laziness muss diese nachgebildet werden. Eine Möglichkeit ist die Darstellung eines auszurechnenden Ausdrucks als Graphstruktur.

Wir bezeichnen die verwendeten Graphen korrekterweise nicht als Baum, da sie nicht immer kreisfrei sind. Trotzdem ist immer ein ausgezeichneter Knoten gegeben, den wir wie die Wurzel eines Baumes darstellen werden.

Jeder Knoten steht für einen Ausdruck, der von anderen Ausdrücken abhängen kann. Dabei kann dieser Knoten beliebig weit ausgerechnet sein, oder gar nicht.

Sei etwa eine Funktion f mit einem Parameter gegeben und ein Ausdruck `g ()`, der zu 42 ausgewertet werden kann. Dabei gilt `f 42` = `21`.

Dieser Graph steht für den Ausdruck `f (g ())`.

Da `g ()` zu 42 ausgewertet werden kann, ist das Ergebnis des Ausdrucks 21. Mit diese Graphstruktur lässt sich nun auch Sharing darstellen. Da Ausdrücke als Graphen dargestellt sind, bedeutet Sharing der Ausdrücke, dass mehrere Programmteile denselben (identitätsgleichen) Graphen nutzen.

Damit ist es möglich, dass der Ausdruck `g ()` an anderer Stelle zu 42 ausgewertet wird und damit der Untergraph

transformiert wird zu

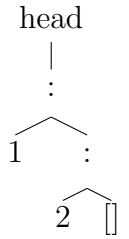
Da dieser Graph auch vom Aufruf von f genutzt wird, hat sich auch der Graph des Aufrufs verändert.

Diese Darstellung führt dazu, dass Funktionsaufrufe bei ihrere Reduktion den Graphen transformieren.

Sei etwa die Funktion *head* und der Ausdruck **head** [1,2] gegeben.

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

Der Ausdruck als Graph g dargestellt sieht wie folgt aus:



Um diesen Ausdruck zu reduzieren, muss er durch die Funktion *head* transformiert werden.

Das heißt nach der Transformation ist $g =$
1

3.2. Übersetzungskonzept

3.2.1. Nicht-Determinismus als Datenstruktur

Programme mit nicht-deterministischen Werten produzieren auch nicht-deterministische Ergebnisse. Die Ausgabe dieser Ergebnisse entweder durch Ausdrucken oder durch Übergabe an nicht-deterministische Programmteile kann damit nicht direkt erfolgen. Um mögliche Ergebnisse zu finden, muss der Suchraum durchsucht werden.

Je nachdem, wie gesucht wird, kann dies zu Ergebnissen in unterschiedlicher Reihenfolge führen oder dazu, dass bestimmte Ergebnisse nie gefunden werden. Ergebnisse werden etwa nicht gefunden, wenn ein unendlich großer Suchraum durchsucht werden muss, bevor ein Ergebnis gefunden werden kann. Dies ist vergleichbar mit Suchen von -1 in einer Curryliste `[1..] ++ [-1]`. -1 kann nie gefunden werden, da eine unendlich lange Liste durchsucht werden muss, um es zu finden.

Damit ist die Wahl der Suchstrategie für eine Programmiersprache mit nicht-deterministischen Werten wichtig. In der Praxis ist Tiefensuche oftmals günstig, allerdings ist diese bekanntermaßen nicht vollständig. Breitensuche ist vollständig, aber nicht immer sehr performant. Andere Suchstrategien existieren, um die Vor- und Nachteile der beiden Strategien zu kombinieren bzw. auszugleichen.

Gerade der Unterschied zwischen Tiefen- und Breitensuche macht aber deutlich, dass die Wahl der Suchstrategie großen Einfluss auf die interne Struktur der Implementierung einer Sprache hat. Eine Implementierung, die Tiefensuche verwendet,

muss sich nur mit den letzten getroffenen Entscheidungen beschäftigen. Wenn eine Entscheidung durch Backtracking verändert wird, ist dies stets die zuletzt getroffene Entscheidung. Dies entspricht einer klassischen Stapelstruktur.

Breitensuche dagegen springt zwischen unterschiedlichen Entscheidungen hin und her. Dabei werden alle Ergebnisse langsam gleichzeitig aufgebaut. Diese Struktur entspricht eher einer Warteschlange, wobei aber, anders als bei Tiefensuche, der gesamte Kontext gespeichert werden muss. Kontext bedeutet hier alle bisher getroffenen Entscheidungen, die sich naturgemäß an unterschiedlichen Stellen im Baum unterscheiden.

Andere Strategien haben andere Eigenheiten, dies wirft die Frage auf, welche Strategie eine Implementierung unterstützen soll. Ohne eine klar beste Strategie ist dies eine schwierige Entscheidung. Eine Idee, wie dies zu umgehen ist, wäre es, gar keine Strategie direkt zu integrieren, sondern stattdessen sämtliche Ergebnisse in einer einheitlichen Datenstruktur darzustellen. Eine Implementierung führt alle Berechnungen auf dieser Datenstruktur aus und produziert eine solche Datenstruktur der Ergebnisse.

Erst dann wird eine Suchstrategie angewendet, um konkrete Ergebnisse zu produzieren. Die Strategie trifft für eine konkrete Entscheidung eine Wahl, welcher Teil der Datenstruktur als nächstes auszuwerten ist. Die Auswertung bis zur nächsten Entscheidung übernimmt wieder der Rest der Implementierung. Dadurch ist die Suchstrategie von der restlichen Implementierung getrennt und ist leicht austauschbar und implementierbar.

In einer Sprache mit Laziness ist diese Idee direkt umsetzbar, indem nicht-deterministische Wahlen selbst Werte sein können. Etwa durch direkte Erweiterung der Datentypen durch nicht-deterministische Wahlen *Choice* und Fehlschläge *Fail*. Der Datentyp *Bool*

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

wird dann erweitert zu

```
data Bool = True | False | Choice Bool Bool | Fail
```

Damit steht etwa `Choice True (Choice False Fail)` für die Werte $True$ und $False$. Dabei ist nicht ausgeschlossen, dass Werte doppelt vorkommen.

Die folgende Funktion soll das Ergebnis eines Münzwurfs als Wahrheitwert zweimal zurückgeben.

```
f :: Bool
f = let x = coin in (x, x)
```

Die rechte Seite dieser Funktion wird mit dem obigen Schema wie folgt reduziert.

```
f = (Choice True False, Choice True False)
```

Damit ist aber eine Run-Time-Choice implementiert, keine Call-Time-Choice. (**True**, **False**) ist auch ein mögliches Ergebnis.

Um Call-Time-Choice zu implementieren, fügen wir zur erweiterten Datenstruktur noch einen Identifier hinzu. Der Datentyp sieht dann wie folgt aus:

data Bool = **True** | **False** | Choice ChoiceId **Bool Bool** | Fail

Sei D ein Datentyp-Bezeichner mit $t_1, \cdots, t_m$ Typparametern, $C_1, \cdots, C_n$ Konstruktorbezeichnern, $p_{1,1}, \cdots, p_{1,n_1}, \cdots, p_{n,1}, \cdots, p_{n,n_n}$ Typausdrücken und $m, n, n_1, \cdots, n_n \in \mathbb{N}$.

data D $t_1 \cdots t_m = C_1\ p_{1,1}\ p_{1,n_1} \mid \cdots \mid C_n\ p_{n,1}\ p_{n,n_n}$

Sei d : ChoiceId $\rightarrow \{L, R\}$ eine Funktion, die jedem Identifier eine Entscheidung zuordnet.

Nun legen wir fest: Seien c_1 = Choice $id_1\ val_{1,1}\ val_{1,2}$ und c_2 = Choice $id_1\ val_{2,1}\ val_{2,2}$ zwei Werte mit $val_{1,1}, val_{1,2}$ Werte eines Typs und $val_{2,1}, val_{2,2}$ Werte eines Typs und id_1 und id_2 $ChoiceId$s.

Die Entscheidung, für einen Wert $val_{1,1}$ oder $val_{1,2}$ hängt von $d(id_1)$ ab. Falls $d(id_1) = L$, so ist ist $val_{1,1}$ der gewählte Wert von c_1, sonst $val_{1,2}$.

Somit wählen c_1 und c_2 beide den linken oder beide den rechten Wert, falls $id_1 = id_2$.

Die obige Funktion f kann nun zu folgender Form mit $id \in ChoiceId$ reduziert werden.

f = (Choice id **True False**, Choice id **True False**)

Somit sind nur die Ergebnisse (**True**, **True**) und (**False**, **False**) möglich.

3.2.2. Pull-Tabbing

Nicht-deterministische Werte können überall vorkommen. Funktionen müssen also mit dem Auftauchen von *Choice* an beliebiger Stelle zurechtkommen.

```
coin :: Bool
coin = True ? False

not :: Bool -> Bool
not True = False
not False = True

notCoin :: Bool
notCcoin = not coin
```

Beim Pattern-Matching ist nicht direkt klar, wie das funktioniert. Welche Regel wird bei dem Ausdruck **not** coin angewendet? Die Lösung liegt in einem Pull-Tab-Schritt, das heißt in einer Umwandlung, die nicht-deterministische Wahlen in Richtung der Wurzel verschiebt.

not coin reduziert sich zu **not** (**True** ? **False**). Intuitiv würde man das Ergebnis **False** ? **True** erwarten. Dieses könnte sich aus dem Ausdruck **not True** ? **not False** ergeben. Der Schritt von **not** (**True** ? **False**) zu **not True** ? **not False** ist genau ein Pull-Tab-Schritt.

Sei für einen Funktionsaufruf einer Funktion f mit Parameterausdrücken $p_1, p_n, n \in \mathbb{N}$ der Aufruf gegeben.

$$\mathsf{f}\ p_1 \cdots p_n = \mathsf{f}\ p_1 \cdots p_{i-1}\ (q_1\ ?_{id}\ q_2)\ p_{i+1} \cdots p_n$$

Sei weiterhin bei der Reduzierung des Funktionsaufrufs die Kopfnormalform des Arguments $p_i, i \in \mathbb{N}$ gefordert. Falls $p_i = q_1\ ?_{id}\ q_2$, wobei q_1, q_2 Ausdrücke sind und $?_{id}$ eine nicht-deterministische Wahl mit Identifier id ist, dann muss ein Pull-Tab-Schritt ausgeführt werden. Dieser wandelt den Funktionsaufruf in die folgende Form um.

$$\begin{aligned}&\mathsf{f}\ p_1 \cdots p_{i-1}\ (q_1\ ?_{id}\ q_2)\ p_{i+1} \cdots p_n\\ =\ &(\mathsf{f}\ p_1 \cdots p_{i-1}\ q_1\ p_{i+1} \cdots p_n)\ ?_{id}\ (\mathsf{f}\ p_1 \cdots p_{i-1}\ q_2\ p_{i+1} \cdots p_n)\end{aligned}$$

Dadurch wird die zu treffende Wahl $?_{id}$ vorgezogen. Abhängig von der gewählen Suchstrategie wird eine Wahl für id getroffen und damit ein auszuwertender Ausdruck $(\mathsf{f}\ p_1 \cdots p_{i-1}\ q_j\ p_{i+1} \cdots p_n)$, $j \in \mathbb{N}$ ausgewählt. Zur weiteren Reduktion des Ausdrucks können weitere Pull-Tab-Schritte oder Reduktionen notwendig sein, bis die benötigten Positionen in Kopfnormalform sind. Danach kann der Funktionsausdruck wie eine deterministische Funktion reduziert werden.

Freie Variablen wurden hier ignoriert, da sie im nächsten Abschnitt behandelt werden.

3.2.3. Freie Variablen

Freie Variablen stehen für unbekannte Werte. Wenn die Typen von freien Variablen bekannt sind, lassen sie sich auch als Disjunktion ihrer möglichen Werte verstehen.

Das führt zu der Idee, freie Variablen tatsächlich als Disjunktionen von Werten zu implementieren. Dazu wird jedem Datentyp eine Funktion zugeordnet, die nicht-deterministisch alle seine Werte respräsentiert. Eine solche Funktion wird auch als Generatorfunktion bezeichnet.

Für den obigen Typ *Bool* sähe die Generatorfunktion wie folgt aus:

```
genBool :: Bool
genBool = True ? False
```

Sei etwa ein Aufruf einer Funktion mit einer freien Variable gegeben.

```
f :: Bool
f = not b
  where b free
```

Durch Ersetzen der freien Variable durch einen Generator ergibt sich folgende Funktion.

```
f :: Bool
f = not genBool
```

Diese kann wie normale nicht-deterministische Wahlen mit Pull-Tab-Schritten transformiert werden.

4. Implementierung

4.1. Struktur

Die Implementierung unterteilt sich in mehrere Abschnitte.

Zunächst muss der Curry-Code in Java-Code übersetzt werden. Das geschieht durch den in Curry implementierten Übersetzer, der im ersten Abschnitt dieses Kapitels beschrieben wird. Dieser Vorgang ist wiederum in unterschiedliche Phasen unterteilt.

Der Curry-Code wird mit dem zu Curry gehörendem Frontend in *FlatCurry*-Code übersetzt. Daraus erzeugt unser Übersetzer zunächst das Zwischenformat *ICurry*. Basierend darauf wird der eigentliche Java-Code erzeugt.

Dieser Java-Code ist zwar das übersetzte Curry-Programm, benötigt zur Ausführung aber noch weiteren Java-Code. Dieser wird im zweiten Abschnitt dieses Kapitels beschreiben.

4.2. Übersetzung

4.2.1. ICurry

Die Übersetzung eines Curry-Programms nach Java soll durch Repräsentation des Programmablaufs als Graphstruktur gelöst werden. Dazu erzeugt der Übersetzer ein Java-Programm, das diesen Graph zur Laufzeit erzeugt und transformiert. Die Ausführung des Programms erfolgt dann durch Graphreduktion. Bei der Umsetzung eines Curry-Programms in diese Graphstruktur müssen eine Reihe von Problemen gelöst werden.

Die Auswertung einer Fallunterscheidung in einer Funktion muss dazu führen, dass die Kopfnormalform eines Ausdrucks berechnet wird. So muss, in der gegebenen Funktion *not*, die Kopfnormalform von x berechnet werden, um die Fallunterscheidung durchführen zu können.

```
not :: Bool -> Bool
not x = case x of
  True -> False
  False -> True
```

Die Berechnung dieser Kopfnormalform erfolgt durch den Interpreter. Jede Kopfnormalform kann zur Berechnung wieder eine andere Kopfnormalform benötigen. Falls der Interpreter jedesmal in der Funktion selbst aufgerufen wird, baut sich auf dem Call-Stack eine große Struktur auf. Für jede Kopfnormalform sind das der Aufruf der übersetzten Curry-Funktion, der Aufruf des Interpreters darin und mögliche Hilfsfunktion im Interpreter. Das führt schnell zu einem Stack-Überlauf insbesondere in einer Sprache, wie Java, die keine Endrekursionsoptimierung besitzt.

Um das zu verhindern, muss die Berechnung der Kopfnormalform vor die Funktion gezogen werden. Eine Funktion muss also im Voraus wissen, ob sie eine Kopfnormalform benötigt und welche. Dazu muss jede Funktion so umgeformt werden, dass Fallunterscheidungen nur am Anfang der Funktion auftreten können. Da nur Fallunterscheidungen zur Berechnung der Kopfnormalform führen können, steht damit für jede Funktion fest, welche Kopfnormalform sie benötigt.

Fallunterscheidungen an anderen Stellen, insbesondere in lokalen Definitionen, werden durch Funktionsaufrufe von Hilfsfunktionen ersetzt. Bei der Übersetzung muss eine Funktion also möglicherweise in mehrere Funktionen unterteilt werden. Dabei müssen lokale Definitionen zwischen diesen Aufrufen geteilt werden, ohne dass die Laziness verloren geht. Die Hilfsfunktionen können also neue Parameter haben, die die Originalfunktion nicht hatte.

Das genutzte Übersetzungsschema sieht vor, dass freie Variablen in Generatoren, also nicht-deterministische Aufzählungen aller Werte, umgewandelt werden. Dazu muss zu jedem Datentyp eine Generatorfunktion eingeführt werden, die dieses leistet. Freie Variablen werden dann als Aufruf dieser Funktionen umgesetzt.

Der Aufruf einer Generatorfunktion für eine freie Variable hängt nur von ihrem Typ ab, allerdings kann dieser Typvariablen enthalten. Diese Typvariablen müssen als Parameter an die Generatorfunktion übergeben werden.

Wenn eine freie Variable eine Generatorfunktion aufruft, müssen die Parameter durch Typinferenz bestimmt werden. Diese bestehen aus expliziten Aufrufen von Generatoren, falls der Typ feststeht, sonst aus Generatoren von Typvariablen. Auch normale Funktionen müssen deswegen für jeden ihrer Typparameter einen weiteren Parameter bekommen, der den Generator des Typparameters übergibt.

Zusätzlich ergeben sich bei der Umsetzung der Graphstuktur speziell in Java weitere Probleme. Um diese Probleme nicht alle gleichzeitig lösen zu müssen, unterteilen wir den Übersetzungprozess in zwei Teile. Aus dem Curry-Programm, dargestellt als FlatCurry-Struktur, wird zunächst das Zwischenformat ICurry. Dieses wird später weiter in Java-Code übersetzt.

Generatoren

Freie Variablen werden in ICurry durch Generatoren dargestellt. Diese sind nicht-deterministische Disjunktionen von allen möglichen Werten. Da Curry statisch getypt ist, hat jede (freie) Variable einen Typ. Von Literalen abgesehen sind Typen

in Curry definierte Datentypen mit endlich vielen, bekannten Konstruktoren und Parametern. Eine Disjunktion aller Werte lässt sich also für Datentypen leicht realisieren.

```
data Bool = True | False
data MaybeBool = Nothing | Just Bool

genBool = True ? False
genMaybeBool = Nothing ? Just genBool
```

Die Funktion *genBool* kann entweder *Nothing* oder *Just genMaybeBool* sein, wobei *genBool* wiederum alle Werte von *Bool* aufzählt. Damit lässt sich zu jedem Datentypen leicht ein Generator erhalten.

```
data Maybe a = Nothing | Just a

genMaybe = Nothing ? Just (genA)
```

Datentypen können aber Typparameter haben, zu diesen lässt sich nicht direkt ein Generator finden, da der Typ, für den ein Typparameter steht, zur Übersetzungzeit nicht bekannt ist. Im Beispiel müsste *genA* bekannt sein, aber der konkrete Typ von *a* ist nicht bekannt, daher auch sein Generator nicht.

Die Lösung dieses Problems besteht in ICurry darin, dass Typparameter von Generatoren in formale Parameter von Funktionen übersetzt werden. Diese formalen Parameter enthalten jeweils den Generator des Typs der Typvariable.

```
data Maybe a = Nothing | Just a
data MyList a = MyList [a]

genMaybe genA = Nothing ? Just (genA)
genMyList genA = MyList (genList genA)
```

Im Fall von *genMyList* kann man sehen, dass die formalen Parameter, genau wie die Typparameter, auch von einem zum anderen Generator übergeben werden können. Hier wird der Parameter *genA* in *genMyList* an den Listengenerator *genList* übergeben.

Da wir zur Übersetzungzeit den Typen jeder freien Variable kennen, kann der gewünschte Generator gefunden werden.

```
f :: Maybe Bool
f = x
  where x free
```

wird zu

```
f = genMaybe genBool
```

Allerdings können auch Funktionen Typparameter haben, bei denen ergibt sich dasselbe Problem, wie bei Datentypen. Wieder ist der Typ nicht vollständig bekannt,

sondern hängt von Typparametern ab. Hier ist die Lösung offensichtlich dieselbe: auch Funktionen müssen für jeden ihrer Typparameter zusätzlich einen formalen Parameter für Generatoren bekommen.

```
f :: Maybe a
f = x
  where x free
```

wird zu

```
f :: a -> Maybe a
f genA = genMaybe genA
```

In Curry gilt die referenzielle Transparenz bei Auftreten von nichtdeterministischen Werten nicht mehr vollständig.

```
x = 1 ? 2
f = (x, x)

g = let y = 1 ? 2 in (y, y)
```

f hat die Ergebnisse (1,1), (1,2), (2,1), (2,2), g aber nur (1,1), (2,2). Das liegt daran, dass Entscheidungen der Call-time-Semantik unterliegen. Beide y haben die gleiche Identität, da y eine lokale Definition und ein Wert ist. Die x sind unterschiedliche Entscheidungen, da 0-stellige Funktionen auf Top-Ebene als unterschiedlich angesehen werden.

Dieses Problem tritt auch bei Generatoren auf:

```
f :: a
f = (x, x)
  where x free

test = f :: Bool
```

wird mit dem obigen Schema übersetzt zu

```
f :: a -> a
f genA = (genA, genA)

test = f genBool
```

Da *genA* eine 0-stellige Funktion ist, sich aber nicht auf Top-Ebene befindet, haben beide *genA*'s die gleiche Identität. Das widerspricht der Semantik von freien Variablen, die unabhängig sein müssen. Daher wird in ICurry unterschieden zwischen der Nutzung eines normalen Parameters und der Nutzung eines Generatorparameters. Wenn auf eine Variable als Generatorparameter zugegriffen wird, sind die getroffenen Entscheidungen unabhängig von anderen Entscheidungen desselben Wertes. Damit ergibt sich als Pseudocode, wobei Generatorwerte mit spitzen Klammern gekennzeichnet werden:

```
f :: a -> a
f genA = (<genA>, <genA>)

test = f genBool
```

Beachte, dass *genBool* kein Generatorparameter ist, denn *genBool* ist eine Top-Level-Funktion. Daraus ergibt sich für *test* korrekterweise:

```
> test
(True, True)
(True, False)
(False, True)
(False, False)
```

Definition

Die Struktur von ICurry entspricht zu großen Teilen der von FlatCurry. Ein ICurry-Programm setzt sich aus einem oder mehreren Modulen zusammen, in denen die Programmelemente enthalten sind. Alle Bezeichner in ICurry, bis auf Modulnamen, befinden sich im selben Namensraum, sind also modulweit eindeutig.

IProg	::=	*ModName* {*ModName*} {*IData*} {*IFunc*}	
IData	::=	*DataName* {*ICons*}	(Datentyp)
ICons	::=	*CombName Arity OriName*	(Wertkonstruktor)
IFunc	::=	*CombName Arity TypeExpr IRule*	(exportierte Funktion)
	|	*CombName Arity IRule*	(private Funktion)
IRule	::=	*Arity ExternName*	(extern definierte Regel)
	|	{*IParam*} {*ILocal*} *IExpr*	(einfache Regel)
	|	*CombName* {*IParam*} {*ILocal*} {*VarIndex*} *VarIndex* [*Literal*] {*IBranch*}	(Case-Regel)
IBranch	::=	*IPattern* {*ILocal*} *IExpr*	(Branch eines Cases)
ILocal	::=	*VarIndex IExpr*	(lokale Definition)
IExpr	::=	*VarIndex*	(Variable)
	|	*Literal*	(Literal)
	|	*String*	(String)
	|	{*IExpr*}	(Liste)
	|	gen *VarIndex*	(Generator einer Typvariable)
	|	noGen *String*	(nicht-existenter Generator)

	| fCall *QName* {*IExpr*}	(Funktionsaufruf)
	| cCall *QName* {*IExpr*}	(Konstruktoraufruf)
	| fPart *Arity QName* {*IExpr*}	(partieller Funktionsaufruf)
	| cPart *Arity QName* {*IExpr*}	(partieller Konstruktoraufruf)
	| or {*IExpr*}	(Disjunktion)
	| free {*IExpr*}	(Disjunktion einer freien Variable)
IPattern	::= *QName* {*IParam*}	(Konstruktor-Match)
	| *Literal*	(Literal-Match)
IParam	::= *VarIndex* | _	

Ein Modul $IProg$ besteht aus einem Namen, einer Menge von importierten Modulen, einer Menge von Datentypen und einer Menge von Funktionen. Importierte Module können im aktuellen Modul verwendet werden.

$IData$ ist ein benannter Datentyp und enthält eine Menge von Wertkonstruktoren. Mit Hilfe dieser Datenstrukturen werden alle Informationen eines ICurry-Programms gespeichert. Eine Information besteht darin, welcher Konstruktor verwendet wird, um ein $IData$ darzustellen.

Ein Konstruktor $ICons$ setzt sich aus einem Namen, der Stelligkeit und einem Originalnamen zusammen. Der Name des Konstruktors dient, wie auch bei Datentypen, zur Identifikation des Konstruktors selbst und kann vom Originalnamen im Quelltext abweichen. Der Originalname dagegen ist der Name, der auch im Original (Curry-)Programm verwendet wurde. Er wird unter anderem bei der Ausgabe eines Konstruktors benötigt. Die Stelligkeit eines Konstruktors gibt an, wie viele Parameter dieser bekommt. Jeder dieser Parameter ist wiederum ein Datum eines bestimmten Datentyps. Im Gegensatz zu FlatCurry wird dieser Typ aber in ICurry nicht mehr angegeben.

Funktionen $IFunc$ kommen in zwei Varianten vor, als exportierte und private Funktion. Eine exportierte Funktion kann von außen aufgerufen werden. Um (getypten) Code für diesen Aufruf generieren zu können, enthält eine exportierte Funktion zusätzlich noch eine Typinformation $TypeExpr$. $TypeExpr$ ist in FlatCurry definiert und bestimmt hier den Typ der Funktion, also die Stelligkeit, Typen der Parameter und Rückgabetyp. Ansonsten stimmen exportierte und private Funktionen überein. Weiterhin enthält einen Funktion einen Namen, eine explizite Angabe der Stelligkeit und ihre Regel. Die Stelligkeit gibt an, wie viele Parameter die Funktion konkret verarbeitet, d.h. sie kann niedriger sein, als die Stelligkeit, die sich aus der Typinformation ergibt.

```
inc :: Int -> Int
inc = (+1)
```

So hat die Funktion *inc* eine Stelligkeit von 0, die Typinformation ist aber eine einstellige Funktion auf Zahlen.

Eine Regel *IRule* enthält den eigentlichen Inhalt einer Funktion, sie gibt also an, durch welchen Ausdruck die linke Seite ersetzt wird. Es gibt drei Typen von Regeln. Eine externe Regel wird außerhalb des ICurry-Programms definiert. Wie ihre Funktionalität ausgedrückt wird, hängt davon ab, in welche Ausgabe das ICurry-Programm später transformiert wird. Eine externe Funktion enthält nochmals die Stelligkeit und einen Exportnamen. Dieser Exportname entspricht dem in FlatCurry und kann zusammen mit dem Namen der Funktion verwendet werden, um die Funktionalität zu identifizieren.

Eine einfache Regel gibt an, dass ein Funktionsaufruf durch einen einfachen Ausdruck ersetzt werden kann. Eine solche Regel besteht aus einer Liste von formalen Parametern, einer Menge von lokalen Definitionen und dem Ausdruck der rechten Seite. Die Länge der Liste von Parametern entspricht der Stelligkeit der Funktion. Die formalen Parameter bestimmen, welche Argumente die Funktion erwartet und werden durch ihren Wert ersetzt, falls sie auf der rechten Seite vorkommen. Lokale Definitionen definieren weitere Variablen, die auf der rechten Seite oder in anderen lokalen Definitonen verwendet werden können. Dabei sind auch Zyklen erlaubt.

Eine Case-Regel enthält ebenfalls Parameter und lokale Definitonen, zusätzlich aber auch einen weiteren Funktionsnamen, eine Liste von Variablenidentifikatoren, eine Fallunterscheidungsvariable, ein optionales Literal und eine Liste von *IBranch*es. Eine Case-Regel unterscheidet sich vor allem dadurch von einer einfachen Regel, dass sie eine Fallunterscheidung erlaubt. Die rechte Seite unterteilt sich hier daher in verschiedene Fälle, beschrieben als *IBranch*. Der zusätzliche Funktionsname dient dazu dem Nutzer von ICurry die Möglichkeit zu geben, eine Case-Regel in zwei Funktionen aufzuteilen. Eine solche Aufteilung kann dazu genutzt werden, vor der Ausführung der Fallunterscheidung zu erfahren, auf welchem Ausdruck die Fallunterscheidung stattfindet. Der zusätzliche Funktionsname ist dabei lediglich ein neuer, eindeutiger Funktionsbezeichner. Die Liste von Variablenidentifikatoren dient ebenfalls einer solchen Aufteilung. Sie gibt an welche Parameter und lokalen Variablen in der eigentlichen Fallunterscheidung verwendet werden. Die Fallunterscheidungsvariable ist die Variable, auf der die tatsächliche Fallunterscheidung stattfindet. Das optionale Literal ist ein letzter Rest Typinformation. ICurry ist im Wesentlichen ungetypt, da aber eine Fallunterscheidung auf Literalen oftmals anders abläuft als auf Konstruktoren, gibt dieses optionale Literal an, ob und auf was für einem Literal die Fallunterscheidung stattfindet. Der Wert des Literals selbst ist ohne Bedeutung. Die Liste von *IBranch*es bestimmt, zu welchem Ausdruck die Funktion ausgewertet wird, abhängig von der Fallunterscheidung.

Ein *IBranch* besteht aus einem Pattern, der Fallunterscheidung und dem Ausdruck, zu dem reduziert wird, falls das Pattern auf die Eingabe passt. Zusätzlich kann jeder Fall auch noch eigene lokale Definitionen enthalten.

Lokale Definitionen werden als $ILocal$ dargestellt. Diese bestehen einfach aus dem Identifikator der definierten Variable und ihrem Ausdruck.

Das Kernstück eines ICurry-Programms bilden die Ausdrücke $IExpr$. Sie beschreiben die berechneten Werte.

Eine Variable steht für den Wert einer lokalen Variable oder eines Parameters. $VarIndex$ stammt aus FlatCurry und ist die ID einer Variable.

Ein Literal steht für den Wert des Literals selbst, etwa 42 oder $'*'$.

Ein String ist die Kurzschreibweise für eine Liste von Zeichen. Eine Liste ist eine Schreibweise für Prelude-Listen, Listen könnten auch als $cCall$ von $Prelude.:$ dargestellt werden.

Der Generator einer Typvariable verhält sich wie eine Variable, mit dem Unterschied, dass Entscheidungen zwischen gleichen Variablen nicht geteilt werden. Die Ausdrücke `gen 2` und `gen 2` produzieren also Werte, die sich in ihren nichtdeterministischen Entscheidungen unterscheiden können. Dies wird zur Realisierung von Generatorparametern verwendet.

In ein paar Situationen ist es nicht möglich einen Generator zu bestimmen, obwohl einer benötigt würde. Diese Fälle von nicht-existenten Generatoren werden hier mit noGen und einem String gekennzeichnet. Dieser String gibt die Art von fehlendem Generator an, mögliche Werte sind:

- "Int": Generatoren für Literale können nicht direkt in ICurry ausgedrückt werden, da es quasi unendlich viele Werte gibt. Allerdings kann eine Zielsprache Generatoren für Ganzzahlen durch Darstellung als Binärrepräsentation realisieren.
- "Char": Für Zeichen gilt dasselbe wie für Zahlen, da Zeichen nur uminterpretierte, nicht negative Zahlen sind
- "Float": Fließkommazahlen sind schwierig aufzuzählen, aber es ist prinzipiell möglich
- "IO": Für IO-Aktionen können keine Generatoren erzeugt werden
- "Function": Für Funktionen können ebenfalls keine Generatoren erzeugt werden
- "Unknown" unterscheidet sich von den anderen durch den Grund, warum ein Generator nicht direkt erzeugt werden kann. In den anderen Fällen waren es Werte, die schwierig aufzuzählen waren, hier ist der Grund, dass es Fälle gibt, in denen Informationen fehlen. Falls der Typ einer freien Variable oder eines Generatorparameters nicht bestimmt werden kann, wird ein `noGen "unknown"` erzeugt.

Der letzte Fall tritt in folgenden Situationen auf:

```
f :: String
f = show x
  where
    x free

g :: String
g = show (error "")
```

Im Fall von f ist der Typ der freien Variable x nicht bestimmbar, da die einzige Verwendung in *show* den Typ **show** :: a −> **String** hat. Dieser Fall muss zu einem Fehler führen, da das Ergebnis von show sonst alle (typunabhängig) möglichen Werte anzeigen müsste.

Im zweiten Fall g ist das Problem etwas subtiler. Aber sowohl **show** :: a −> **String**, als auch **error** :: **String** −> a benötigen einen Generatorparameter. Dieses ist der Generator für a, der allerdings wieder jeder mögliche Typ sein könnte. Der Generator wird allerdings weder von *show* noch von *error* benutzt, somit genügt es einen Dummygenerator einzufügen, der einen Fehler wirft, sobald er Werte generieren soll.

Ein Funktionsaufruf $fCall$ unterscheided sich nicht weiter von einem Funktionsaufruf in FlatCurry. Es wird ein $QName$ verwendet, um die aufzurufende Funktion zu identifizieren, die Argumente sind dabei weitere Ausdrücke.

Generatorparameter werden hier als normale Parameter behandelt.

Ein Konstruktoraufruf $cCall$ funktioniert genauso, wie ein Funktionsaufruf. Generatorparameter werden hier nicht benötigt, da Konstruktoren nie freie Variablen erzeugen.

Ein partieller Funktionsaufruf $fPart$ unterscheidet sich von einem normalen Funktionsaufruf dadurch, dass nicht alle von der Funktion benötigten Parameter übergeben werden. Die Anzahl der fehlenden Parameter wird in der Arity notiert.

Die Generatorparameter müssen hier bereits alle vollständig vorhanden sein.

Ein partieller Konstruktoraufruf $cPart$ entspricht einem partiellem Funktionsaufruf ohne Generatorparameter.

Eine Disjunktion *or* ist ein Ausdruck für die nicht-deterministische Auswahl eines Wertes. Die Menge von Ausdrücken $IExpr$ gibt alle möglichen Werte an.

Die Disjunktion einer freien Variable $free$ verhält sich genauso wie eine normale Disjunktion *or*. Der einzige Unterschied ist die Information, dass die Disjunktion eine freie Variable bildet.

$free$ wird nicht direkt für freie Variablen verwendet, die ja durch Generatoraufrufe ersetzt werden. Stattdessen werden Disjunktionen innerhalb von Generatorfunktionen damit markiert.

$IPattern$ werden in Fallunterscheidungen verwendet. Dabei wird unterschieden zwischen einem Konstruktor-Match und einem Literal-Match. Ein Konstruktor

matcht, falls der QName zu dem übergebenen Konstruktor passt. Dabei können $IParam$s als Argumente angegeben werden. Die Variablen dieser $IParam$s werden an die entsprechenden Werte gebunden.

Ein Literal matcht, falls der Wert des Literals dem Wert der Fallunterscheidung entspricht.

Ein $IParam$ entspricht einem $VarIndex$, mit dem Zusatz, dass ein $IParam$ auch _ sein kann. Ein _ steht für eine frische Variable, die niemals verwendet wird.

Nicht definiert in dieser Grammatik sind einige Nichtterminalsymbole aus FlatCurry:

- *Arity* ist ein Zahl, die Stelligkeiten angibt
- $TypeExpr$ respräsentiert einen Typausdruck
- $VarIndex$ ist eine Zahl, genutzt zur Identifikation von lokalen Variablen und Parametern
- *Literal* kann eine ganze Zahl, eine Fließkommazahl oder ein Zeichen sein
- $QName$ ist ein Tupel aus einem Modulbezeichner und einem Namen, es steht für ein Top-Level-Element eines Moduls. Etwa für eine Funktion f im Modul $Module1$: `("Module1", "f")`

Ebenfalls nicht definiert sind diverse Bezeichner, dies sind alles Zeichenketten in unterschiedlichen Kontexten. Bis auf $OriName$ und $ExternName$ sind alles Bezeichner, die sich von ihren Originalnamen im Curry-Quelltext unterscheiden können.

- $ModName$ ist der Name eines Moduls
- $DataName$ ist der Name eines Datentyps
- $CombName$ ist der Name einer Funktion oder eines Konstruktors
- $OriName$ ist der originale Name eines Konstruktors, wie er im Quelltext auftaucht.
- $ExternName$ ist eine Zusatzinformation zu externen Funktionen, um ihre Implementierung zu finden

Übersetzungsprozess

Bei der Übersetzung von FlatCurry nach ICurry müssen im Wesentlichen drei Probleme gelöst werden.

1. Freie Variablen müssen durch Generatoren ersetzt werden. Zu jedem Datentyp müssen Generatorfunktionen erzeugt werden.

2. Funktionen müssen so umgeformt werden, dass Fallunterscheidungen nur ganz am Anfang von Funktionen auftreten. Dabei kann jede Funktion nur eine Fallunterscheidung durchführen.

3. Lokale Definitionen müssen an den Anfang von Funktionen oder Branches verschoben werden.

Generatoren Statt direkt FlatCurry zu verwenden, basiert unsere Übersetzung tatsächlich auf Annotated-FlatCurry. Annotated-FlatCurry stellt zusätzlich zu den FlatCurry-Informationen noch Typinformationen bereit. Damit ist zu jeder freien Variable und jedem Funktionsaufruf bekannt, welche Typen sie verwenden. Diese Typen können noch von den Typparametern der umgebenden Funktion abhängen.

Jeder Funktionsaufruf bekommt also noch Generatorparameter dazu, dann kann jede freie Variable durch einen Generatoraufruf ersetzt werden.

```
f :: [a]
f = g

g :: [a]
g = x where x free
```

wird zu

```
f :: a -> [a]
f genA = g genA

g :: a -> [a]
g genA = genList <genA>
```

Dazu muss ein *TypeExpr*-Ausdruck in einen *IExpr*-Ausdruck transformiert werden.

TypeExpr hat die Form:

TypeExpr	::=	*TVarIndex*	(Typvariable)
	|	*TypeExpr TypeExpr*	(Funktionstyp)
	|	*QName* {*TypeExpr*}	(Konstruktortyp)

Eine Typvariable t wird durch den Generatorparameter der umgebenden Funktion $genT$ ersetzt. Falls die umgebende Funktion einen solchen Typparameter nicht hat, ist es ein unbekannter Generator noGen "Unknown".

Im Beispiel wird sowohl in f, als auch in $g\ a$ durch $genA$ ersetzt.

Ein Funktionstyp wird durch noGen "Function" ersetzt, da es keinen Generator für Funktionen gibt.

Falls der $QName$ ein Literal oder IO ist, wird er wieder durch den passenden $noGen$-Ausdruck ersetzt. Ein gewöhnlicher Konstruktortyp wird durch Aufruf des Generators des entsprechenden Konstruktors ersetzt. Alle Argumente werden genauso ersetzt.

Im Beispiel wird der Typ [a] durch genList <genA> ersetzt.

Damit sind sämtliche Generatorausdrücke beschrieben und ICurry braucht keine expliziten freien Variablen mehr.

Zu jedem Datentypen muss noch eine Generatorfunktion erzeugt werden. Die Konstruktoren werden dazu nicht-deterministisch aufgezählt. Jeder Konstruktor erhält für jeden seiner Typparameter einen Generatorausdruck. Die Erzeugung der Generatorfunktionen für Datentypen nutzt das gleiche Schema für Typparameter.

Disjunktionen werden in diesem Beispiel als '?' markiert, das steht hier für die Disjunktion einer freien Variable, nicht für eine einfache Disjunktion.

```
data Bool = True | False
data Maybe a = Nothing | Just a

genBool = True ? False
genMaybe genA = Nothing ? Just <genA>
```

Der Generator zu $Bool$, $genBool$ ist eine Disjunktion seiner Konstruktoren. Genauso $genMaybe$, aber mit $genA$ als zusätzlichem Parameter.

Fallunterscheidungen und lokale Definitionen Fallunterscheidungen können in FlatCurry überall in Ausdrücken vorkommen, in ICurry aber nur am Anfang von Funktionen. Um aus einem Programm in FlatCurry eines in ICurry zu erstellen, müssen Fallunterscheidungen ersetzt werden.

Dazu wird jede Fallunterscheidung durch einen Funktionsaufruf ersetzt und eine neue Funktion mit frischem Namen eingeführt, die diese Fallunterscheidung durchführt. Da Parameter und lokale Definitionen an mehreren Stellen benutzt werden können, kann es passieren, dass die neue Funktion andere Argumente benötigt, als die ursprüngliche Funktion.

Weiterhin können lokale Definitionen in ICurry nur am Anfang einer Funktion oder einer Verzweigung einer Fallunterscheidung stehen. Um das ICurry-Programm korrekt generieren zu können, ist es daher nötig zu wissen, welche Variablen ein

bestimmter Ausdruck benötigt und welche Variablen er zur Verfügung stellt. Dazu werden beim Aufbau der Ausdrücke diese Informationen bottom-up berechnet.

```
fromJust :: a -> a -> Maybe a -> a
fromJust genA d x = case x of
  Nothing -> d
  Just y  -> y
```

In der angegebenen Funktion $fromJust$ sind die untersten Ausdrücke d und y. Beide Ausdrücke definieren keine neuen Variablen und benötigen die Werte genau einer Variablen, jeweils d und y.

Die erste Verzweigung **Nothing** -> d benötigt weiterhin d und definiert nichts weiter.

Die zweite Verzweigung **Just** y -> y definiert die Variable y und benötigt keine weiteren Variablen. Definitionen sind an Verzweigungen erlaubt, also hat der Teilausdruck keine weiteren Anforderungen.

Die Fallunterscheidung als ganzes benötigt die Variable d durch die erste Verzweigung und die Variable x. Damit kann die Fallunterscheidung in eine neue Funktion ausgelagert werden. Diese benötigt genau die Variablen, die die Fallunterscheidung benötigt, bekommt diese also als Parameter.

```
fromJust :: a -> a -> Maybe a -> a
fromJust genA d x = fromJust' d x

fromJust' :: a -> Maybe a -> a
fromJust' d x = case x of
  Nothing -> d
  Just y  -> y
```

Der Generatorparameter $genA$ wird von $fromJust'$ nicht benötigt, muss also auch nicht übergeben werden. Dagegen muss $fromJust$ diesen Parameter weiterhin bekommen, da sie von außen aufgerufen werden kann und äußere Funktionen diesen Parameter immer mitliefern.

Der Ausdruck **fromJust'** d x benötigt schließlich d und x, die von $fromJust$ direkt bereitgestellt werden. In diesem einfachen Beispiel könnte die Aufspaltung in $fromJust$ und $fromJust'$ natürlich auch wegoptimiert werden, da die rechte Seite von $fromJust'$ genau der ursprünglichen rechten Seite von $fromJust$ entspricht.

4.2.2. Java

Im Weiteren wird die Darstellung von Programmen und die Übersetzung von ICurry nach AbstractJava beschrieben. Zur Repräsentation von Curryprogrammen in Java verwenden wir eine direkte Darstellung von Ausdrücken als Graph. Das Programm zur Laufzeit hat also eine Graphstruktur und Funktionen werden in Java-Code über-

setzt, der diesen Graphen transformiert. Alle durchgeführten Berechnungen erfolgen als Operationen auf dem Programmgraphen.

AbstractJava

Java-Code wird nicht direkt ausgegeben, sondernd in einer eigenen Datenstruktur AbstractJava dargestellt. AbstractJava unterstützt nicht die vollständige Java-Syntax sondern nur ausgewählte Strukturen, die für die Darstellung eines Curryprogramms mit *Cam* notwendig sind.

Knoten

Der Ausführungsgraph besteht aus verschiedenen Typen von Knoten. Diese werden in Java als Instanz einer einzigen Klasse dargestellt, der Klasse *Node*. Knoten haben einen von mehrerer Typen:

- **Funktionsknoten** stehen für den Aufruf einer bestimmten Funktion mit Parametern. Diese Parameter sind wiederum Knoten. Die Reduktion eines solchen Funktionsknotens ist die Umwandlung des Knotens in einen anderen Typen, der der rechten Seite der Funktion entspricht. Damit ist dies der einzige Knotentyp, der kein Endpunkt ist.

  Funktionsknoten teilen sich noch einmal in zwei Arten auf, einfache Funktionsknoten und GHnf-Funktionsknoten. GHnf-Funktionsknoten können erst weiter reduziert werden, wenn sich ein bestimmter Parameter in Grund-Kopfnormalform befindet. Welcher Parameter das ist, speichert der Knoten ebenfalls.

- **Konstruktorknoten** stehen für ein Datum eines bestimmten Konstruktors. Konstruktorknoten können nicht weiter reduziert werden, die enthaltenen Kindsknoten aber schon.

- **Partielle Funktions- und Konstruktorknoten** sind Funktionen bzw. Konstruktoren, denen noch mindestens ein Parameter fehlt. Anders als reguläre Funktionsknoten können sie nicht weiter reduziert werden und können auch den fehlenden Parameter nicht aufnehmen. Um einen weiteren Parameter aufzunehmen müssen die partiellen Informationen in einen anderen Knoten hinzugefügt werden. Der Grund dafür ist, dass eine partielle Funktion an unterschiedlichen Stellen des Programms genutzt werden kann und dort unterschiedliche Parameter bekommen kann.

- **Auswahl- und freie Knoten** enthalten ein Menge von Knoten und repräsentieren nicht-deterministisch diese Werte. Freie Knoten können zusätzlich auch gebunden werden und entstehen nur aus freien Variablen.

- **Fehlschlagsknoten** respräsentieren fehlgeschlagene Berechnungen, sie verhalten sich wie Auswahlknoten mit leeren Wertmengen.
- **Guard-Knoten** enthalten einen Unterknoten und einen oder mehrere Constraints. Der Knoten steht für seinen Unterknoten, falls die Constraints erfüllt sind, sonst für einen Fehlschlag.
- **Identitätsknoten** enthalten einen weiteren Knoten und verhalten sich wie dieser Unterknoten. Sie sind notwendig für Abbildungsfunktionen, in denen der Wert einer Funktion einem ihrer Parameter entspricht.
- **IO-Aktions-Knoten** stehen für eine IO-Aktion. Ähnlich wie partielle Funktionen werden sie selber nicht reduziert, wenn die IO-Aktion ausgeführt wird.
- Knoten aus **Literalen** Int, Char und Float stehen für diese Werte.

Knoten können im Graphen mehrfach vorkommen (Sharing). Gleichzeitig können sich Funktionsknoten in andere Knotentypen umwandeln. Sobald ein Knoten an einer Stelle umgewandelt (reduziert) wurde, soll diese Veränderung überall wo der Knoten auftaucht stattfinden.

Um dies zu gewährleisten, ohne dass jeder Knoten alle Stellen kennt, an denen er auftaucht, wird der Typ in der Knotenklasse *Node* selbst gespeichert. Verschiedene Knotentypen sind also keine Unterklassen von *Node*, sondern *Node* speichert seinen Typen als Attribut.

Damit teilen sich alle Knotentypen dieselben Attribute, um ihre Information darzustellen. Diese Attribute sind die folgenden:

- Ein **Funktions- bzw. Konstruktoridentifikator**, der in allen Knotentypen genutzt wird, die eine bestimmte Funktion bzw. einen bestimmten Konstruktor repräsentieren. Er besteht eigentlich aus zwei Teilen: einer Referenz auf eine Modulinstanz und einer Ganzzahl Jedes Curry-Modul wird in Java in eine Modulklasse übersetzt, dadurch ist das Modul eindeutig bestimmt. Die Ganzzahl identifiziert die Funktion bzw. den Konstruktor innerhalb dieses Moduls eindeutig.
- Eine Liste von **Kindsknoten** enthält die Parameter des Knotens. Für einen Funktions- bzw. Konstruktorknoten und ihre partiellen Versionen sind es die normalen Parameter, für Auswahl- und freie Knoten sind es die möglichen nicht-deterministischen Werte. Guard- und Identitätsknoten speichern hier ihren einzigen Unterknoten, für dessen Wert sie stehen. IO-Aktions-Knoten können ebenfalls Parameter enthalten.
- Jeder Knoten kann zusätzlich noch eine **Knoteninformation** enthalten. Diese enthält zusätzliche Informationen, speziell für diesen Knotentyp. Für Auswahl-

und freie Knoten ist dies ein Identifikator für die Wahl. Knoten mit gleichem Identifikator müssen die gleiche Wahl treffen.

Literale von Fließkommazahlen speichern hier ihren Wert.

Für Guard-Knoten werden hier die Constraints gespeichert, die der Guard erfüllen muss, um nicht fehlzuschlagen.

- Schließlich enthält jede *Node* noch einen ganzzahligen Wert, der für Zeichenliterale und Ganzzahlliterale ihren Wert enthält. Außerdem speichert dieser Wert für partielle Funktionen und Konstruktoren, wie viele Parameter noch benötigt werden.

Durch Ändern dieser Attribute, inklusive des Attributs, das den Knotentypen bestimmt, kann ein Knoten sämtliche in Curry möglichen Daten darstellen. Zum Verändern der Attribute bietet *Node* Hilfsfunktionen zu jedem Knotentyp an, etwa toCons(Module m, **int** fld), um einen Knoten in einen Konstruktorknoten zu verwandeln.

Unterknoten werden mit der Methode setChilds(Node... cs) neu gesetzt, bzw. mit initChilds (Node... cs) zum ersten Mal gesetzt.

Module

Jedes Curry-Modul wird in eine einzige Java-Klasse übersetzt. Diese enthält Informationen darüber, welche Funktionen das Modul enthält, wie diese auszuführen sind und welche Konstruktoren das Modul enthält.

Es sei folgendes Curry-Modul gegeben:

```
module List where
  data List a = Empty | Cons a (List a)

  head :: List a -> a
  head xs = case xs of
    Cons x _ -> x
```

Eine erzeugte Modul-Klasse leitet sich von *Module* ab und hat stets die unten angegebene Struktur.

Es enthält eine Instanz dieser Modulklasse, in der Variable *INSTANCE*, die in Knoten abgelegt werden kann, um auf dieses Modul zu verweisen. Damit sind die genutzten Klassen de facto Singletons, auch wenn das Anlegen neuer Instanzen möglich ist. Veränderliche Attribute haben Modulklassen nicht, die tatsächlichen Daten werden in den Knoten gehalten.

Die weiteren Konstanten sind Zahlwerte, die als modulweit eindeutige Identifikatoren dienen. Ein Knoten der einen Konstruktor repräsentiert ist mit einer Instanz der Modulklasse und dem Identifikator des Konstruktors eindeutig identifizierbar. Im Beispiel sind die Identifikatoren der Konstruktoren *Empty* und *Cons*.

Da Typen in Knoten nicht dargestellt werden, ist kein Feld für den Typ *List* nötig. Allerdings wird ein Generator für *List* benötigt. Dieser wurde in *ICurry* als normale Funktion *genList* erzeugt.

head ist der Funktionsidentifikator der gleichnamigen Curry-Funktion.

h0_head ist ein anderer Funktionsidentifikator für eine Hilfsfunktion von *head* Diese wird aufgerufen, nachdem die für *head* notwendige Kopfnormalform berechnet wurde.

Alle Funktionsidentifikatoren in Modulen dienen dazu die Funktion in einem Knoten zu repräsentieren. Um die Funktion tatsächlich zu reduzieren, d.h. durch ihre linke Seite zu ersetzen, muss die Methode *callFunction* aufgerufen werden.

callFunction kommt in allen Modulen vor und dient dazu, zu einem Funktionsknoten die passende Methode auszuführen. Um einen Funktionsknoten zu reduzieren, wird also *callFunction* auf der Modulinstanz aufgerufen und der Knoten selbst als Argument übergeben. Die Methode *callFunction* ruft dann die passende Methode auf, abhängig vom Funktionsidentifikator im Knoten. Im Beispiel ist das ein Switch-Statement über `n.id`.

```
public class List implements Module {
  public final List INSTANCE = new List();
  public final int Empty = 1;
  public final int Cons = 2;
  public final int genList = 3;
  public final int head = 4;
  public final int h0_head = 5;

  public void callFunction(Node n) {
    switch(n.id) {
      case genList:
        genList(n);
        break;
      case head:
        head(n);
        break;
      case h0_head:
        h0_head(n);
        break;
    }
  }

  private void genList(Node n) {
    ...
  }

  private void head(Node n) {
```

```
    ...
  }

  private void h0_head(Node n) {
    ...
  }

}
```

Funktionen

Jede Funktion eines Moduls hat die Aufgabe einen Knoten, der diese Funktion repräsentiert entsprechend ihrer Funktionalität zu transformieren. Eine Funktion bekommt als Eingabe also einen einzigen Knoten. *ICurry* kennt drei unterschiedliche Arten von Funktionen:

- Funktionen mit Regeln ohne Fallunterscheidung
- Funktionen mit Regeln mit Fallunterscheidung
- Externe Funktionen

Funktionen ohne Fallunterscheidung Funktionen, deren Regel keine Fallunterscheidung hat, benötigen zur Reduzierung keine Kopfnormalform. Der Funktionsknoten wird direkt transformiert, wobei die Parameter im neuen Teilgraphen verwendet werden können.

Aus dem obigen Beispiel ist die Generatorfunktion *genList* ein Beispiel für eine solche Funktion. Der entsprechende Curry-Code dieser Funktion sehe wie folgt aus:

```
genList :: a -> List a
genList genA = empty ? cons
  where
    empty       = Empty
    genListNode = genList genA
    cons        = Cons <genA> genListNode
```

Die Methode genList(Node n) muss hier den Funktionsknoten n in einen Graphen verwandeln, der die rechte Seite repräsentiert. Dazu müssen zunächst die Argumente der Funktion aus dem Funktionsknoten herausgeholt werden. In diesem Beispiel ist das die Zeile 2, in der der Generatorparameter aus dem Unterknoten-Array geholt wird.

Danach werden lokale Definitionen erzeugt, hier in den Zeilen 4 bis 6. Zunächst werden alle lokale Definitionen als leere Knoten angelegt, da sich jeder Knoten auf jeden anderen Knoten beziehen kann. Die Beziehungen sind also nicht unbedigt

kreisfrei. Die Knoten zunächst als leere (damit ungültige) Knoten zu nutzen ist für die Ausführung kein Problem, da jede Berechnung auf dem Graphen erst nach vollständiger Ausführung der Methode erfolgt.

Erst danach kann der Graph der lokalen Definitionen aufgebaut werden. Dies geschieht im Beispiel in den Zeilen 8 bis 14. Es wird jeweils der Typ des Knoten gesetzt und die Unterknoten festgelegt. So ist *empty* ein Konstruktor *Empty* ohne Unterknoten und *genListNode* eine Funktion *genList*, mit *genA* als einzigem Argument.

Schließlich kann die Transformation des eigentlichen Funktionsknoten in den Zeilen 17 und 18 auf dieselbe Weise erfolgen. Da *genList* ein Generator ist, wird der Funktionsknoten mit *toFreeFromGen* in eine freie Variable verwandelt. Als Argumente werden die lokalen Variablen *empty* und *cons* gesetzt.

```
1  private void genList(Node n) {
2    Node genA = n.childs[0];
3
4    Node empty = new Node();
5    Node genListNode = new Node();
6    Node cons = new Node();
7
8    empty.toCons(INSTANCE, Empty);
9    empty. initChilds ();
10
11   genListNode.toFn(INSTANCE, genList);
12   genListNode. initChilds (genA);
13
14   cons.toCons(INSTANCE, Cons);
15   cons. initChilds (genA.toDeepCopy(), genListNode);
16
17   n.toFreeFromGen();
18   n. setChilds (empty, cons);
19 }
```

Funktionen mit Fallunterscheidung Funktionen, deren Regel eine Fallunterscheidung enthält, benötigen zur Reduzierung die Kopfnormalform eines Ausdrucks. Dazu wird die Funktion in zwei Methoden aufgeteilt.

Die Aufgabe der ersten Methode ist es, festzulegen, welche Kopfnormalform benötigt wird, danach terminiert die Methode sofort. Nun darf die zweite Methode erst aufgerufen werden, wenn die Kopfnormalform bestimmt ist. Nachdem diese bestimmt ist, genügt es, wenn die zweite Methode eine Fallunterscheidung auf dem entsprechenden Argument macht.

Im obigen Beispiel ist die Funktion *head* eine Funktion, die eine Fallunterscheidung macht. Die Methode *head* ist von der ersten Art. Wie eine Funktion ohne

Fallunterscheidung packt sie zunächst in den Zeilen 2 und 3 alle Argumente aus.

In Zeile 5 und 6 erfolgt dann die eigentliche Funktionalität, die dazu führt, dass die zweite Methode aufgerufen wird. Die Umformung funktioniert wie ein normaler Funktionsaufruf. Der einzige Unterschied ist, dass der Typ des Knotens mit $toFnGHnf$ gesetzt wird. $toFnGHnf$ sorgt dafür, dass der n-te Unterknoten vor dem Aufruf der Methode in Grund-Kopfnormalform sein muss. n ist durch den letzten Parameter von $toFnGHnf$ bestimmt.

Die Methode $h0\_head$ packt ebenfalls ihre Parameter aus und führt dann die Fallunterscheidung auf dem entsprechenden Argument aus. Dabei genügt eine Unterscheidung anhand des Konstruktor-Identifikators, da der Typ und damit das Modul des Knotens durch das Typsystem von Curry bereits bekannt sind.

```
1  private void head(Node n) {
2    Node genA = n.childs[0];
3    Node list  = n.childs [1];
4
5    n.toFnGHnf(INSTANCE, h0_head, 0);
6    n.setChilds( list );
7  }
8
9  private void h0_head(Node n) {
10   Node list  = n.childs [0];
11   switch(list.id) {
12     case Cons:
13       Node x = list.childs [0];
14       n.toIdentity (x);
15       break;
16     default:
17       n.toFail ();
18   }
19 }
```

Externe Funktionen Die Implementierung einer externen Funktionen wird nicht im Curry-Programm selbst angegeben, sondern muss manuell geschrieben werden. Der generierte Code einer externen Funktion beschränkt sich daher auf den Aufruf dieses manuellen Codes. Implementierungen von externen Funktionen müssen sich in Klassen mit demselben Namen, wie die Modulklasse befinden, im Paket cam.external. Die Signatur der Methode entspricht der, der generierten Methode.

4.2.3. Übersetzung im Detail

Wir beschreiben die Übersetzung von ICurry nach AbstractJava bzw. Java durch die Übersetzungsfunktion $tr_{IProg} :: ICurry \rightarrow AbstractJava$ genauer. Dabei geben

wir das Ergebnis als Pseudo-Java-Code an. Die Eingabe bezieht sich auf die Grammatik von $ICurry$, wobei die Regeln der Grammatik als Tupel ihrer rechten Seiten aufgefasst werden.

Das n-te Element eines Tupels t wird mit t^n bezeichnet.

Listen von Elementen aus der Grammatik werden wie Mengen mit impliziter, fester Reihenfolge behandelt, also M_n für das n-te Element einer Menge M. Für Mengen, deren Reihenfolge nicht durch die Grammatik festgelegt ist, wird eine beliebige, feste Reihenfolge angenommen.

tr_{IProg} übersetzt das Gesamtmodul in eine Java-Klasse mit den nötigen Methoden.

$tr_{\mathsf{IProg}}(modName, iDatas, iFuncs)$ = **class** $modName$ **implements** Module {
  Module INSTANCE = **new** $modName$();
  int $cons_1 = ident(cons_1)$;
  ...
  int $cons_{|cons|} = ident(cons_{|cons|})$;
  int $funcs_1^1 = ident(funcs_1^1)$;
  ...
  int $funcs_{|funcs|}^1 = ident(funcs_{|funcs|}^1)$

  public void callFunction(Node n) {
    switch(n.id) {
     case $funcs_1^1$:
       $funcs_1^1$(n);
       break;
     ...
     case $funcs_{|funcs|}^1$:
       $funcs_{|funcs|}^1$(n);
       break;
     case $hnfFuncs_1^1$:
       $hnfFuncs_1^1$(n);
       break;
     ...
     case $hnfFuncs_{|hnfFuncs|}^1$:
       $hnfFunc_{|hnfFuncs|}$(n);
       break;
    }
  }

  public void $funcs_1^1$(Node n) {
    $tr_{\mathsf{IFunc}}(funcs_1)$;
  }
  ...
  public void $funcs_{|funcs|}^1$(Node n) {

$tr_{\mathsf{IFunc}}(funcs_{|funcs|})$;
}

public void $hnfFuncs^1_1$(Node n) {
$tr_{\mathsf{HnfFunc}}(hnfFuncs^1_1)$;
}
...
public void $hnfFuncs^1_{|hnfFuncs|}$(Node n) {
$tr_{\mathsf{HnfFunc}}(hnfFuncs_{|hnfFuncs|})$;
}
}

Dabei sei

$$hnfRules = \{(hnfName, name, iParams, iLocals, hnfParams, caseParam) \\ |(name, \_, (hnfName, iParams, iLocals, hnfParams, caseParam, \_, \_)) \\ \in funcs \text{ Funktion mit Case-Regel}\}$$

die Menge von Regeln mit Fallunterscheidung. Dies sind die Regeln, für die eine zweite Methode benötigt wird. Diese wird in tr_{HnfFunc} generiert.

Weiterhin sei

$$funcs = \{(name, arity, iRule)|(name, arity, \_, iRule) \in iFuncs\} \\ \cup\{(name, arity, iRule)|(name, arity, iRule) \in iFuncs\}$$

Diese Menge vereinheitlicht die beiden $IFunc$-Varianten einer exportierten und privaten Funktion.

Es sei $cons = \{c|(\_, (c, \_, \_)) \in iDatas\}$, dies ist lediglich zur Lesbarkeit nötig.
Außerdem sei $ident :: String \to \mathbb{N}$ eine Funktion, die jedem Funktions- und Konstruktornamen eine eindeutige Zahl zuordnet.
Für alle Regeln mit Parameter $iParams$ sei außerdem $params = \{p|p \in iParams, p/ = \_\}$.

tr_{IFunc} erzeugt zu jeder $ICurry$-Funktion den Code der entsprechenden Java-Methode. Dieser ist für jede der drei Varianten von $IRule$ etwas anders. Die drei Varianten sind hier in der Reihenfolge: einfache Regel, Case-Regel, extern definierte Regel.

$tr_{\mathsf{IFunc}}(\_, \_, (iParams, iLocals, iExpr)) =$
Node v_{params_1} = n.childs[0];
...
Node $\mathsf{v}_{params_{|params|}}$ = n.childs[$|params| - 1$];
Node $\mathsf{v}_{iLocals^1_1}$ = **new** Node();

$\ldots$
Node $\mathsf{v}_{iLocals^1_{|iLocals|}}$ = **new** Node();
$tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals^1_1}, iLocals^2_1)$
$\ldots$
$tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals^1_{|iLocals|}}, iLocals^2_{|iLocals|})$
$tr_{\mathsf{IExpr}}(\mathsf{n}, iExpr)$

$tr_{\mathsf{IFunc}}(\_, \_, (\_, \_, \_, hnfParams, caseVar, \emptyset, branchs)) =$
Node $\mathsf{v}_{hnfParams_1}$ = n.childs[0];
$\ldots$
Node $\mathsf{v}_{hnfParams_{|params|}}$ = n.childs[$|hnfParams| - 1$];
switch($caseVar$.id) {
$tr_{\mathsf{IBranch}}(branchs_1)$
$\ldots$
$tr_{\mathsf{IBranch}}(branchs_{|branchs|})$
}

$tr_{\mathsf{IFunc}}(\_, \_, (\_, \_, \_, hnfParams, caseVar, \{\_\}, branchs)) =$
Node $\mathsf{v}_{hnfParams_1}$ = n.childs[0];
$\ldots$
Node $\mathsf{v}_{hnfParams_{|params|}}$ = n.childs[$|hnfParams| - 1$];
switch($caseVar$.value) {
$tr_{\mathsf{IBranch}}(branchs_1)$
$\ldots$
$tr_{\mathsf{IBranch}}(branchs_{|branchs|})$
}

$tr_{\mathsf{IFunc}}((mod, combName), \_, (\_, \_)) = \mathsf{external}.mod.combName(\mathsf{n});$

tr_{IBranch} generiert Code, der in den Zweigen der Fallunterscheidungen benötigt wird. Die erste Variante arbeitet mit Konstruktoren, die zweite ist nur für Literale notwendig.

$tr_{\mathsf{IBranch}}(((mod, name), iParams), iLocals, iExpr) =$
case $mod.name$:
Node v_{params_1} = n.childs[0];
$\ldots$
Node $\mathsf{v}_{params_{|params|}}$ = n.childs[$|params| - 1$];
Node $\mathsf{v}_{iLocals^1_1}$ = **new** Node();
$\ldots$
Node $\mathsf{v}_{iLocals^1_{|iLocals|}}$ = **new** Node();
$tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals^1_1}, iLocals^2_1)$
$\ldots$
$tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals^1_{|iLocals|}}, iLocals^2_{|iLocals|})$
$tr_{\mathsf{IExpr}}(\mathsf{n}, iExpr)$
break;

$tr_{\mathsf{IBranch}}(literal, iLocals, iExpr) =$
  case $literal$:
    Node $\mathsf{v}_{iLocals_1^1}$ = **new** Node();
    ...
    Node $\mathsf{v}_{iLocals_{|iLocals|}^1}$ = **new** Node();
    $tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals_1^1}, iLocals_1^2)$
    ...
    $tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals_{|iLocals|}^1}, iLocals_{|iLocals|}^2)$
    $tr_{\mathsf{IExpr}}(\mathsf{n}, iExpr)$
    break;

tr_{HnfFunc} erzeugt Code für die Extramethoden, die für eine Case-Regel notwendig sind.

$tr_{\mathsf{HnfFunc}}(\_, (mod, name), iParams, iLocals, hnfParams, caseParam) =$
  Node v_{params_1} = n.childs[0];
  ...
  Node $\mathsf{v}_{params_{|params|}}$ = n.childs[$|params| - 1$];
  Node $\mathsf{v}_{iLocals_1^1}$ = **new** Node();
  ...
  Node $\mathsf{v}_{iLocals_{|iLocals|}^1}$ = **new** Node();
  $tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals_1^1}, iLocals_1^2)$
  ...
  $tr_{\mathsf{IExpr}}(\mathsf{v}_{iLocals_{|iLocals|}^1}, iLocals_{|iLocals|}^2)$
  n.toFnGHnf(mod.INSTANCE, $mod.name$, elemIndex($caseParam, iParams$));
  n. setChilds ($\mathsf{v}_{hnfParams_1}, \cdots, \mathsf{v}_{hnfParams_{|hnfParams|}}$);

tr_{IExpr} übersetzt Ausdrücke in ihre Java-Äquivalente. Der erste Parameter ist dabei ein Java-Ausdruck für einen Knoten, der umgewandelt werden soll. In der Grammatik sind Variablen, Literale, Strings und Listen jeweils mit nur einem Parameter angegeben. Eine Möglichkeit diese 4 Fälle zu unterscheiden wird vorausgesetzt.

$tr_{\mathsf{IExpr}}(n, varIndex) = n$.toIdentity($\mathsf{v}_{varIndex}$);
$tr_{\mathsf{IExpr}}(n, literal) = n$.toLit($literal$);
$tr_{\mathsf{IExpr}}(n, string) = n$.toStr($string$);
$tr_{\mathsf{IExpr}}(n, listExpr) = n$.toList($listExpr_1, \cdots, listExpr_{|listExpr|}$);
$tr_{\mathsf{IExpr}}(n, (\mathsf{gen}, varIndex)) = n$.toDeepCopy($\mathsf{v}_{varIndex}$);
$tr_{\mathsf{IExpr}}(n, (\mathsf{noGen}, type)) = n$.toGenError($type$);
$tr_{\mathsf{IExpr}}(n, (\mathsf{fCall}, (mod, name), params)) =$
  n.toFn(mod.INSTANCE, $mod.name$)
  . setChilds (
    tr_{IExpr}(**new** Node(), $params_1$),
    ...
    tr_{IExpr}(**new** Node(), $params_{|params|}$),

```
  );
tr_IExpr(n, (cCall, (mod, name), params)) =
  n.toCons(mod.INSTANCE, mod.name)
  . setChilds (
    tr_IExpr(new Node(), params_1),
    ...
    tr_IExpr(new Node(), params_|params|),
  );
tr_IExpr(n, (fPart, arity, (mod, name), params)) =
  n.toPartialFn(mod.INSTANCE, mod.name, arity)
  . setChilds (
    tr_IExpr(new Node(), params_1),
    ...
    tr_IExpr(new Node(), params_|params|),
  );
tr_IExpr(n, (cPart, arity, (mod, name), params)) =
  n.toPartialCons(mod.INSTANCE, mod.name, arity)
  . setChilds (
    tr_IExpr(new Node(), params_1),
    ...
    tr_IExpr(new Node(), params_|params|),
  );
tr_IExpr(n, (or, iExprs)) =
  n.toChoice()
  . setChilds (
    tr_IExpr(new Node(), iExprs_1),
    ...
    tr_IExpr(new Node(), iExprs_|iExprs|),
  );
tr_IExpr(n, (free, iExprs)) =
  n.toFreeFromGen()
  . setChilds (
    tr_IExpr(new Node(), iExprs_1),
    ...
    tr_IExpr(new Node(), iExprs_|iExprs|),
  );
```

4.3. Laufzeit

4.3.1. Kopfnormalform mit Nicht-Determinismus

Alle Berechnungen bzw. Reduktionen basieren auf der Bestimmung der Kopfnormalform eines Ausdrucks. Da alle Daten in einer Graphstruktur dargestellt werden, muss

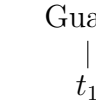
definiert werden, wie eine Kopfnormalform eines solchen Graphen bestimmt wird. Knoten, die Nicht-Determinismus oder Bindungen ausdrücken, werden hier als in Kopfnormalform befindlich verstanden. Diese werden von der Auswertungsstrategie weiterverarbeitet.

Ziel dieses Abschnitts ist es also zu zeigen, wie eine Kopfnormalform an einer beliebigen Stelle im Graphen berechnet wird. Dazu sei ein Graph oder Teilgraph gegeben mit einem ausgezeichneten Knoten, an dem reduziert werden soll. Dieser wird im Folgenden immer als Wurzelknoten dargestellt. $t_1, \cdots, t_n, n \in \mathbb{N}$ seien dazu jeweils beliebige Teilgraphen.

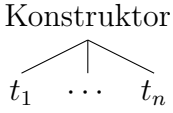
Literal

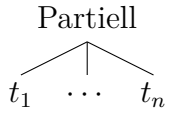
Fehlschlag

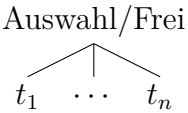
Graphen, die nur aus einem Literal-Knoten oder einem Fail-Knoten bestehen, sind bereits in Kopfnormalform.

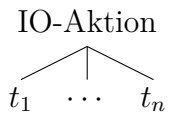


Ein Guard-Knoten befindet sich auch in Kopfnormalform, auch wenn der Untergraph es nicht ist.

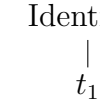






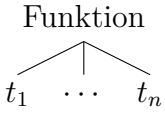


Konstruktorknoten, partielle Funktions- und Konstruktorknoten, Auswahlknoten und IO-Aktionen sind ebenfalls direkt in Kopfnormalform.

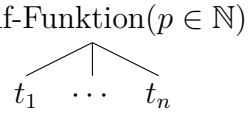


Ein Identitätsknoten ist erst in Kopfnormalform, wenn sich sein Parameter in Kopfnormalform befindet. Damit unterscheidet er sich vom Guard-Knoten. Der

Grund dafür liegt darin, dass der Parameter eines Identitätsknotens immer gilt. Dagegen hängt die Gültigkeit des Parameters eines Guard-Knotens davon ab, ob die Constraints erfüllt sind.



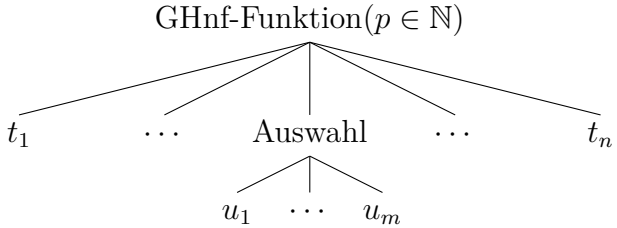
Um einen einfachen Funktionsknoten zu reduzieren, wird er der im Knoten gespeicherten Funktion übergeben. Diese transformiert ihn in einen Graphen, der der rechten Seite der Funktion entspricht. Der resultierende Graph wird wiederum in Kopfnormalform gebracht.



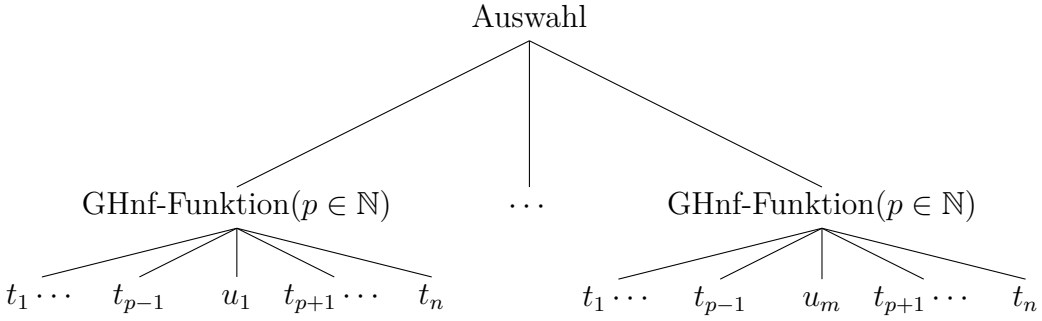
Um einen GHnf-Funktionsknoten zu reduzieren, muss zunächst der Parameter t_p in Kopfnormalform h gebracht werden. Falls p nun kein Fehlschlags-, Guard- oder Auswahlknoten ist, ist der Parameter in Grund-Kopfnormalform. Danch wird wie mit einem einfachem Funktionsknoten weitergemacht.

Andernfalls muss ein Pull-Tab-Schritt auf dem Funktionsknoten ausgeführt werden. Ein Pull-Tab-Schritt auf einem Fehlschlagsknoten bzw. Guardknoten funktioniert dabei so, wie ein Pull-Tab-Schritt auf einem Auswahlknoten ohne, bzw. mit genau einem Unterknoten.

Sei also der Funktionsknoten von der folgenden Form mit $u_1, \cdots, u_m$ Graphen und $m \in \mathbb{N}$ gegeben.



Nach dem Pull-Tab-Schritt sieht der Graph wie folgt aus.



Da sich jetzt ein Auswahlknoten in der Wurzel befindet, ist der Graph in Kopfnormalform.

4.3.2. Suchstrategien

Nach der Berechnung der Kopfnomalform mit Nicht-Determinismus können sich in der Wurzel des Graphen noch nicht-deterministische Wahlen (bzw. Fehlschläge) oder Guards befinden. Um das Ergebnis des Gesamtprogramms ausgeben zu können ist diese Form nicht hilfreich, da eine Durchmischung von Wahlen und Kostruktoren nicht direkt ausgegeben werden kann. Durch Pull-Tabbing werden bei Auswertung des Graphen alle Wahlen und Guards zur Wurzel verschoben. Sobald so ein Element an der Wurzel angekommen ist, muss aber eine Wahl getroffen werden, oder es müssen Constraints überprüft werden.

Um zu einem Ergebnis zu kommen, müssen also Entscheidungen getroffen werden. Dies entspricht einer Suchstrategie in einem Graphen. Abhängig von der Strategie werden Ergebnisse in bestimmter Reihenfolge gefunden, oder auch nicht. Mögliche Strategien sind die oben beschriebene Tiefen- und Breitensuche.

5. Evaluation

In diesem Kapitel vergleichen wir die Laufzeit einiger mit *Cam* erstellter Programme. Dazu verwenden wir die Testmodule aus dem Kapitel *Benchmarks* aus [BHPR11]. Wir vergleichen unsere Implementation *Cam* mit der Haskell-Implementation *KiCS*2 und *PAKCS*, das Prolog nutzt.

Die Tests laufen auf einem Notebook mit Intel® Core ™ i5-4210U statt, auf einem Linux-Mint-System mit 8 GB RAM. Die Module wurden vor dem Test übersetzt, um Übersetzungszeiten möglichst nicht einzurechnen.

Die verwendete Java-Version ist *OpenJDK* 1.8.0_111, 64 bit.

*KiCS*2 wurde in der Version 0.5.1 mit Haskell in der GHC-Version 7.10.3 genutzt.

Für den Test mit *PAKCS* wurde die Version 1.13.1 verwendet, dazu SWI-Prolog in der Version 6.6.4.

Wir gliedern die Test dem Beispiel von [BHPR11] folgend in drei Kategorien.

- Funktionale Programme der ersten Ordnung testen vor allem direkte Funktionsaufrufe.
- Funktionale Programme höherer Ordnung testen auch Funktionsaufrufe, allerdings mit Fokus auf Funktionen höherer Ordnung.
- Nicht-deterministische funktional-logische Programme testen die nicht-deterministischen Aspekte von Curry.

Alle Angaben sind in Sekunden Echtzeit. Alle Implementationen wurden in Standardeinstellungen belassen.

Funktionale Programme der ersten Ordnung

Implementation	ReverseUser	Reverse	Tak	TakPeano
*KiCS*2	0,25	0,245	0,177	0,72
Cam	0,595	0,605	2,254	6,268
PAKCS	12,834	12,697	198,673	291,76

In dieser Kategorie ordnet sich *Cam* zwischen *KiCS*2 und *PAKCS* ein. Die Ergebnisse sind etwa eine Größenordung schlechter, als die von *KiCS*2 und eine Größenordnung besser als die von *PAKCS*. Außreißer sind vor allem die schlechten Ergebnisse von *PAKCS* bei den *Tak*-Tests.

Funktionale Programme der ersten Ordnung

Implementation	ReverseHO	Primes	PrimesPeano	Queens	QueensUser
KiCS2	0,288	0,295	0,585	0,853	0,835
Cam	5,424	1,229	3,179	7,352	7,275
PAKCS	29,298	144,949	196,448	ERROR	ERROR

Hier ergibt sich ein ganz ähnliches Bild wie in der ersten Kategorie. Abgesehen davon, dass *PAKCS* bei *Queens* und *QueensUser* mit einem Fehler abbricht, ist *KiCS2* merklich schneller, als *Cam* und *PAKCS* merklich langsamer.

Nicht-deterministische funktional-logische Programme

Implementation	PermSort	PermSortPeano	Last	RegExp
KiCS2	43,184	45,584	1,789	3,539
Cam	20,967	25,143	out of memory	out of memory
PAKCS	124,349	284,301	14,353	74,594

In dieser Kategorie gibt es gleich zwei Auffälligkeiten.

Einmal schafft *Cam* die Test *Last* und *RegExp* nicht. Das sind gerade die Tests, in denen Variablenbindungen getestet werden. Dies wird in *Cam* durch die Klasse *ConstraintStore* implementiert. Das Ergebnis deutet darauf hin, dass dort noch Potential zur Verbesserung liegt.

Mit kleineren Testgrößen berechnet *Cam* diese Tests korrekt.

Die andere Auffälligkeit sind die Tests *Last* und *RegExp*. Dies sind die einzigen Tests in denen *Cam* die Zeiten von *KiCS2* übertrifft. Aufgrund der direkten Implementation als Graph in *Cam* ist dieses Ergebnis sehr unerwartet. Eine Teil der Erklärung dafür könnte in der verbrauchten CPU-Zeit liegen.

Alle obigen Ergebnisse sind in Echtzeit gemessen und alle Implementationen verwenden keine parallelen Suchstrategien. Allerdings bietet die verwendete CPU 2 Kerne bzw. 4 Threads an.

Implementation	PermSort	PermSortPeano
KiCS2	43,184	45,584
KiCS2 CPU-Zeit	43,006	45,464
Cam	20,967	25,143
Cam CPU-Zeit	36,823	41,644

Zusätzlich sind in der Tabelle jetzt die verbrauchten CPU-Zeiten aufgelistet. Diese unterscheiden sich von den Echtzeiten dadurch, dass sie die tatsächlich verbrauchte

Rechenzeit messen. Durch Nutzung mehrerer CPUs oder Kerne kann diese höher liegen, als die Echtzeit.

Da keine parallelen Suchstrategien genutzt wurden, nutzt $KiCS2$ nur einen Kern aus. Cam verwendet zwar auch keine parallele Strategie, aber Java führt automatisch im Hintergrund verschiedene Threads aus. Insbesondere die automatische Speicherverwaltung (garbage collection) läuft in einem eigenen Thread.

Diese Nutzung könnte ein Grund dafür sein, dass Cam hier erheblich schneller als $KiCS2$ ist. Ein Hinweis darauf ist, dass die verwendete CPU-Zeit von Cam und $KiCS2$ sehr ähnlich ist und erheblich kleiner als die Echtzeit von Cam. Cam hat also in diesem Test sehr viel parallel gearbeitet.

In allen anderen Tests unterscheiden sich die CPU-Zeiten und Echtzeiten nicht wesentlich. Allerdings liegt die CPU-Zeit von Cam immer leicht über der Echtzeit. Dies scheint darauf hinzudeuten, dass in den Tests $PermSort$ und $PermSortPeano$ sehr viel mehr Aufwand für die Speicherverwaltung nötig ist und Cam daher einen Vorteil hat.

Allerdings ist die CPU-Zeit von Cam in den ersten beiden Tests immer noch kleiner als die von $KiCS2$.

6. Ausblick

Cam bietet die Grundfunktionalitäten einer Curry-Implementierung. Verbesserungen sind aber noch an vielen Stellen möglich.

Wie schon die Evaluation gezeigt hat, ist die Implementierung von Bindungen nicht optimal.

Dazu kommen noch andere Probleme. Die Berechnung von Kopfnormalformen ist so ausgelegt, dass Stack-Überläufe möglichst ausgeschlossen werden. Dies gilt aber nicht für die Implementation von Bindungen, insbesondere von Lazy-Bindungen. Hier könnte durch eine ungünstige Kombination von Bindungen ein Stack-Überlauf auftreten.

Dies kann verhindert werden, indem Bindungen, die andere Bindungen bedingen, in einer großen Schleife berechnet werden, ähnlich dem Ansatz in *Interpreter*.

Bei der Übersetzung von *FlatCurry* nach *ICurry* werden neue Funktionen eingeführt. Beim Aufruf einer neuen, internen Funktion wird die Reihenfolge der Parameter beliebig gewählt. Ein Ansatz, bei dem die Parameter in eine günstigere Reihenfolge gebracht werden hätte zwei Vorteile.

Eine Funktion `f v1 v2 v3` muss beim Aufruf einer Hilfsfunktion nicht mehr unnötig Unterknoten umkopieren, etwa beim Aufruf von `f' v2 v3 v1`. Ein Beibehalten der Reihenfolge der Parameter `f' v1 v2 v3` kann dies verhindern.

Der zweite Vorteil wäre eine bessere Lesbarkeit dieser übersetzen Funktionen. Wenn der erste Parameter der Funktion f in der Hilfsfunktion f' als dritter Parameter auftaucht, der zweite Parameter zum ersten wird etc. macht dies das Debuggen schwerer.

In der aktuellen Version enthält *Cam* zahlreiche externe Funktionen, die nicht implementiert sind. Diese führen zu einem Fehler, der mit „ niy “ (Not Implemented Yet) markiert ist. Diese sollten langfristig implementiert werden.

Die Anbindung an Java könnte noch verbessert werden. Momentan basiert die Anbindung an Java auf einer Schnittstelle, die vom Übersetzter generiert wird. Diese unterstützt einige bekannte Datentypen, bietet aber nur rudimentäre Möglichkeiten diese zu erweitern.

Die konkrete Anbindung in `cam.bindings.Bindings` und die Vermittlung zwischen Java-Typen und Curry-Typen in `cam.bindings.types` sind strikt getrennt. Dadurch

ist eine Erweiterung der Vermittlungstypen leicht möglich. Allerdings können diese momentan nur für Typparameter genutzt werden.

Eine Funktion `f :: MyType -> MyType` wird in der Java-Schnittstelle zu einer Methode `Iterator <Showable> f(Showable v1)`. Wobei *Showable* ein sehr allgemeiner Vermittlungstyp ist, der auf Strings basiert. Es ist nicht möglich eine eigene Implementation zur Vermittlung des Typen $MyType$ zu nutzen.

Dies betrifft auch Erweiterungen, wie die Nutzung von $FunctionalInterface$ in Java 8. Diese könnte die Nutzung von Funktionen zwischen Curry und Java verbessern.

Ähnliches gilt für andere Neuerungen von Java 8, wie *Optional* als Java-Pendant zu $Maybe$.

Literaturverzeichnis

[AJ13] Antoy, S. ; Jost, A.: Compiling a Functional Logic Language: The Fair Scheme. In: *23rd Int'nl Symp. on Logic-based Program Synthesis and Transformation (LOPSTR 2013)*. Madrid, Spain : Dpto. de Systems Informaticos y Computation, Universidad Complutense de Madrid, TR-11-13, Sept. 2013, S. 129–143

[AJ16] Antoy, Sergio ; Jost, Andy: A New Functional-Logic Compiler for Curry: Sprite. In: *CoRR* abs/1608.04016 (2016). `http://arxiv.org/abs/1608.04016`

[BHPR11] Brassel, B. ; Hanus, M. ; Peemöller, B. ; Reck, F.: KiCS2: A New Compiler from Curry to Haskell. In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, Springer LNCS 6816, 2011, S. 1–18

[Han07] Hanus, M.: Putting Declarative Programming into the Web: Translating Curry to JavaScript. In: *Proc. of the 9th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'07)*, ACM Press, 2007, S. 155–166

[He16] Hanus (ed.), M.: *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-language.org`, 2016

[HS97] Hanus, M. ; Sadre, R.: A Concurrent Implementation of Curry in Java. In: *Proc. ILPS'97 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*. Port Jefferson (New York), 1997

[Hus88] Hussmann, Heinrich: Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. In: *Proceedings of the International Workshop on Algebraic and Logic Programming*. London, UK, UK : Springer-Verlag, 1988. – ISBN 3–540–50667–5, 31–40

[OW97] Odersky, Martin ; Wadler, Philip: Pizza into Java: Translating theory into practice. In: *In Proc. 24th ACM Symposium on Principles of Programming Languages*, ACM Press, 1997, S. 146–159

[Rob65] Robinson, J. A.: A Machine-Oriented Logic Based on the Resolution Principle. In: *J. ACM* 12 (1965), Januar, Nr. 1, 23–41. http://dx.doi.org/10.1145/321250.321253. – DOI 10.1145/321250.321253. – ISSN 0004–5411

A. Dokumentation

Cam ist eine Implementierung von Curry in Java. Sie übersetzt Curry-Programme in Java-Code, der mit der Laufzeitumgebung von *Cam* ausgeführt werden kann. Die Implementierung basiert darauf, Curry-Ausdrücke zur Laufzeit als Graph darzustellen und Transformationen auf diesem Graphen durchzuführen.

A.1. Voraussetzungen

Cam benötigt eine Java-Installation bestehend aus Laufzeitumgebung (java) und Compiler (javac) der Version 1.7 oder höher.

Da der Übersetzer selbst in Curry geschrieben ist, wird außerdem zum Übersetzen des Übersetzers eine lauffähige Curry-Installation benötigt.

Für die Nutzung werden außerdem Makefiles und Shell-Skripte bereitgestellt, die *make* und eine Shell wie *bash* benötigen. Zur Ausführung von System-Kommandos aus Curry heraus wird ebenfalls ein *bash*-Skript verwendet.

Es wird angenommen, dass *java*, *javac*, *make* und *bash* mit der PATH-Variable zugänglich sind.

A.2. Installation

Vor der Installation muss in der Datei ./runtime/cam/Installation.java die Konstante *installDir* gesetzt werden. Diese muss den absoluten Pfad des *Cam*-Basisordner enthalten. Im Weiteren wird der *Cam*-Basisordner mit . bezeichnet.

Danach kann mit make all die Installation gestartet werden. Dabei wird zusätzlich angenommen, dass das Programm *runcurry* mit der PATH-Variable zugänglich ist. Dieses wird von anderen Curry-Implementierungen, wie *KiCS*2 oder *PAKCS* bereitgestellt.

Der Installationsvorgang sollte je nach Compiler nicht länger als eine halbe Stunde dauern.

Falls die nötigen Curry-Module bereits übersetzt vorliegen, kann die Installation auch mit make bzw. make bootstrap gestartet werden. Dieser Weg nutzt die *Cam*-Implementierung selbst und ist möglicherweise schneller.

Mit make clean werden sämtliche durch die Installation generierten Daten gelöscht.

make shallow_clean löscht nur die durch den Java-Compiler erzeugten *class*-Dateien.

A.3. Übersetzung und Ausführung

Nachdem *Cam* installiert wurde, können Curry-Programme ausgeführt werden. Dazu werden im Ordner ./bin mehrere ausführbare Dateien bereitgestellt.

- *makeCam* übersetzt alle Curry-Module im aktuellen Order und alle Unterordnern. Pfade von Modulen in Unterordnern werden als hierarchische Modulnamen interpretiert.

  Alternativ kann ein einzelnes Curry-Modul im aktuellen Ordner mit ./bin/makeCam curry TARGET=<curry module> übersetzt werden.

  Modulnamen mit Sonderzeichen (Unterstrichen oder einfachen Anführungszeichen) werden von *makeCam* nicht unterstützt, können aber mit dem Übersetzer direkt übersetzt werden.

- *execCam* erwartet den Namen eines übersetzten Moduls im aktuellen Ordner und führt dieses aus. Ein übergebener Modulname sollte nicht maskiert sein und darf keine Endung *.curry* enthalten.

  Hierarchische Module werden unterstützt. Falls der Name nach dem letzten Punkt kein Großbuchstabe ist, wird dieser als Funktion angesehen. Diese Funktion wird dann ausgeführt, ansonsten wird die *main*-Funktion des Moduls ausgeführt.

  Gültige Aufrufe sind folgende:

  - ./bin/execCam Test führt die Funktion *main* im Modul $Test$ aus.
  - ./bin/execCam Test.f führt die Funktion f im Modul $Test$ aus.
  - ./bin/execCam Tests.Test führt die Funktion *main* im Modul $Tests.Test$ aus.
  - ./bin/execCam Tests.Test.f führt die Funktion f im Modul $Tests.Test$ aus.

  Ausgaben werden auf die Standardausgabe geschrieben.

  Ist die aufgerufene Funktion eine IO-Aktion, wird die Ausgabe der IO-Aktion ausgegeben.

  Ist die aufgerufene Funktion keine IO-Aktion, werden alle berechneten Ergebnisse durch Zeilenumbruch getrennt ausgegeben. Gibt es kein Ergebnis, wird 'No more solutions.' ausgegeben.

  Zusätzlich zum Ergebnis werden alle Bindungen ausgegeben. Für die Funktion f = **let last** (_++[x]) = x **in** [**EQ**,**GT**] ist die Ausgabe dann:

(_0 = [EQ], _2 = [], _7 = GT) GT

Hier wurde _ an $[EQ]$ gebunden und x an GT. Die Zahlen sind interne Identifier. Das letzte GT ist die eigentliche Ausgabe.

Die Ausgabe der Bindungen kann mit dem Parameter *−−dont−show−bindings* an $execCam$ unterdrückt werden.

- *cam* stellt eine interaktive Curry-Umgebung bereit. Ein eingegebener Ausdruck wird als rechte Seite einer Funktion interpretiert und das Ergebnis ausgegeben. Dabei können exportierte Funktionen und Konstruktoren aus $Prelude$, sowie allen geladenen Modulen genutzt werden.

  Zusätzlich erkennt *cam* noch mehrere Kommandos:

  - :help gibt eine kurze Hilfe aus.
  - :load <**module**> übersetzt das übergebene Modul und setzt es als geladenes Modul.
  - :reload übersetzt alle geladenen Module neu.
  - :add <**module**> verhält sich wie : *load*, aber fügt das neue Modul zu anderen geladenen Modulen hinzu.
  - :make übersetzt alle Module im aktuellen Ordner und Unterordnern. Verhält sich, als ob $makeCam$ im aktuellen Ordner aufgerufen worden wäre.
  - :quit beendet die Curry-Umgebung.

- $curryName$ liest Modulnamen aus den übergebenen Argumenten oder der Standardeingabe und gibt diese als Cam-Modulnamen aus. Cam-Namen maskieren alle Java-Schlüsselworte mit einem vorangestelltem _ und escapen alle Sonderzeichen außer ., etwa *_doll* für $.

A.4. Ordnerstruktur

Die Ordner im Cam-Basisorder enthalten folgendes:

- ./benchmarks enthält Benchmarks zum Vergleich der Laufzeit von Cam, $KiCS2$ und $PAKCS$. Vor der Nutzung müssen in ./benchmarks/Makefile die beiden Pfade $KICS2HOME$ und $PAKCSHOME$ gesetzt werden. Diese zeigen auf den Basisordner einer $KiCS2$- bzw. $PAKCS$-Installation. Danach kann make all aufgerufen werden.

  Die Tests werden mit dem Script ./benchmarks/run.sh ausgeführt. Diese führen die Module im Ordner mit $KiCS2$, Cam und $PAKCS$ aus und geben die benötigte Laufzeit aus.

- ./bin enthält die ausführbaren Dateien von *Cam*.
- ./classes enthält die *class*-Dateien, die von *javac* erzeugt wurden. Das gilt nur für die *class*-Dateien, die direkt zu *Cam* gehören. *class*-Dateien aus Benutzercode werden an anderer Stelle gespeichert, s.u.
- ./codegen enthält den in Curry geschriebenen Übersetzer. Er spaltet sich auf in mehrere Teile:
  - *AbstractJava* stellt Java-Code in Curry dar und kann ihn ausgeben.
  - *Cam* enthält das ausführbare Modul für den *Cam*-Übersetzer und den Code um *ICurry*-Strukturen in *AbstractJava*-Strukturen zu übersetzen.
  - *ICurry* stellt *ICurry*-Code in Curry dar und kann ihn ausgeben. Dies ist eine Zwischensprache zur Übersetzung von Curry-Code in Java-Code.
  - *ArgumentParsing* stellt Code zum Parsen von Kommandozeilenargumenten bereit.
- ./doc enthält die Dokumentation zu *Cam*.
- ./examples enthält Beispiele für die Nutzung von *Cam*.
- ./jars enthält notwendige Java-Jar-Dateien. Diese werden entweder zur Ausführung von *Cam* benötigt, oder zum Testen desselben.
- ./lib enthält die Standardbibliothek von Curry. Implementierungen von externen Funktionen sind in ./lib/cam/external enthalten.
- ./misc enthält zusätzlich benötigte Dateien.
- ./runtime enthält den Laufzeitcode, der von *Cam* benötigt wird.
- ./**test** enthält die Testfälle von *Cam*.

Generierter Code aus Modulen, die vom Nutzer erzeugt wurden, wird im jeweiligen Ordner erzeugt. Dazu nutzt *Cam*, wie auch andere Curry-Tools den Ordner .curry im Ordner des Moduls. Dieser enthält unter anderem *FlatCurry*-Dateien. *Cam* erzeugt seine Dateien im Unterordner .curry/cam. Dabei liegen erzeugte Java-Dateien in .curry/cam/curry und .curry/cam/java sowie deren Unterordner, entsprechend den hierarchischen Modulnamen. *class*-Dateien liegen im Ordner .curry/cam/classes und seinen Unterordnern.

A.5. Übersetzer

Anstatt *makeCam* o.Ä. zu nutzen, ist es auch möglich den Übersetzer direkt aufzurufen. Dazu muss lediglich die *main*-Funktion im Modul *Cam.Main* ausgeführt werden. Der direkte Aufruf des Übersetzer hat keine Einschränkungen bezüglich der Modulnamen.

Der Übersetzer erwartet als Eingabe ein zu übersetzendes Modul. Außerdem unterstützt er folgende Eingabeoptionen:

- −o | −−*output*=<*outputFile*>: In diese Datei wird der Java-Code geschrieben. Ohne diese Option wird die Datei in den normalen *Cam*-Pfad geschrieben. Dieser normale Pfad ist vom aktuellen Ordner aus .curry/cam/curry/<module path>. Das erzeugte Java-Interface wird in diesem Fall nach .curry/cam/java/<module path> geschrieben.

  Falls als *outputFile* − angegeben wird, wird die erzeugte Datei auf die Standardausgabe geschrieben.

- −mp | −−*module−path*=<*path*>: In diesem Pfad sucht der Übersetzer nach Curry-Modulen, zusätzlich zum aktuellen Ordner. Diese Option kann mehrfach angegeben werden.

- −q | −−*quiet*: Wenn diese Option vorhanden ist, werden Ausgaben unterdrückt.

- −h | −−*help*: Diese Option führt dazu, dass ein Hilfetext ausgegeben wird.

A.6. Tests

Im Ordner ./**test** werden einige *JUnit*-Tests für *Cam* zur Verfügung gestellt. Diese decken die Grundfunktionalität von *Cam* ab, sowie die Implementierungen von externen Funktionen.

Übersetzt und ausgeführt werden sie durch Aufruf von make **test** im *Cam*-Basisordner.

A.7. Java-Anbindung

Der nach Java übersetzte Curry-Code kann direkt in einem Java-Programm verwendet werden. Dazu werden beim Übersetzen der Curry-Module Java-Schnittstellen generiert. Diese stellen Methoden zum Aufruf aller exportierten Funktionen eines Moduls bereit.

Der Quelltext der Schnittstellen befindet sich vom Curry-Modul aus gesehen im Ordner ./. curry/cam/java/. Die nach Java übersetzten Curry-Module befinden sich in ./. curry/cam/curry/.

Kompilate dieser Java-Module befinden sich in ./. curry/cam/classes/. Die von *Cam* selbst zur Verfügung gestellten, übersetzten Klassen befinden sich im *Cam*-Basisordner ./ classes /. Außerdem nutzt *Cam* verschiedene Jar-Archive im Ordner ./ jars / im *Cam*-Basisordner.

Um Curry in Java zu nutzen, müssen sich diese Pfade alle im Klassenpfad befinden. Beispielaufrufe befinden sich in ./examples/**test**.sh im *Cam*-Basisordner.

Die Java-Schnittstellen befinden sich im Java-Paketbaum in cam.java.<Modulname>_. Dabei kann der Modulname hierarchisch sein, die Curry-Modulhierarchie wird also direkt in die Java-Pakethierarchie eingegliedert. Sämtliche wichtigen Klassen zur Nutzung von Curry befinden sich in den Paketen cam.bindings und cam.bindings.types.

Instanzen der Schnittstellen stellt die Klasse cam.bindings.Bindings bereit. Dazu benötigt die Methode **public static** <T **extends** CurryModule> T getInstance(Class<T> intf) das Klassenobjekt einer Schnittstelle.

Die Methodentypen der Schnittstelle bilden die Funktionstypen des zugehörigen Curry-Moduls nach. Curry-Funktionen können in Ein- und Ausgaben Nicht-Determinismus enthalten. Um diesen in Java nachzubilden, werden Iteratoren genutzt. Etwa steht der Typ Iterator <String> nicht-deterministisch für alle Strings, die der Iterator zurückliefert.

Bekannte Datentypen werden von *Cam* in eine passende Java-Darstellung transformiert. Die folgenden Typen verwenden vordefinierte Java-Pendants:

- Für *Bool*-Werte werden *Boolean*s verwendet.
- Für *Char*-Werte werden *Character*s verwendet.
- Für *Int*-Werte werden *Long*s verwendet.
- Für *Float*-Werte werden *Float*s verwendet.
- Für Listen von Werten werden java.util.**List**en verwendet.
- Für Listen von *Char*s werden Strings verwendet. Dies gilt allerdings nur, falls es sich explizit um Listen von *Char*s handelt. Listen von Typvariablen [a] werden als Java-Listen dargestellt, auch falls *a* ein *Char* ist.

Für die folgenden Typen wird eine Java-Repräsentation verwendet, die *Cam* zur Verfügung stellt. Diese Typen liegen im Paket cam.bindings.

- Für *Either*-Werte steht die abstrakte Java-Klasse *Either* mit ihren Unterklassen *Left* und *Right* bereit. Diese enthalten jeweils den Wert eines Typs.

- Für einstellige Funktionen wird die Klasse $Function < P, R >$ verwendet. Instanzen dieser Klasse stellen eine aufrufbare Funktion dar. P und R sind hierbei die Typen des Parameters bzw. des Rückgabewertes. Die Funktionalität der Funktion steckt dabei in der abstrakten Methode **public abstract** Iterator<R> call(Iterator<P> param). Um die Funktion ohne Nicht-Determinismus in den Argumenten aufzurufen, existiert zusätzlich die Funktion **public** Iterator<R> call(P param).

  Um eine Funktion zu implementieren die keine nicht-deterministischen Rückgaben hat, ist es auch möglich die Unterklasse Function.Simple zu nutzen. Deren abstrakte Methode ist **public abstract** R simpleCall(Iterator<P> param).

  Mehrstellige Funktionen werden durch Currying mit einstelligen Funktionen realisiert. Für zweistellige Funktionen stellt die Klasse $Fn2$ Hilfskonstrukte bereit.

- *IO*-Aktionen werden ähnlich wie Funktionen dargestellt, allerdings können sie keinen Parameter bekommen und keinen Nicht-Determinismus enthalten. Die Klasse *IO* kapselt ihre Aktion dazu in der abstrakten Methode der vordefinierten Schnittstelle $Callable < T >$.

- *Ordering*-Werte werden mit der Java-Klasse *Ordering* dargestellt. Diese stellt genau drei Instanzen von sich zur Verfügung: LT, EQ und GT.

- Bis zu 15-stellige Tupel werden mit den Klassen $Tuple1$ bis $Tuple15$ dargestellt.

- Für ()-Werte stellt die Klasse *Unit* eine Singleton-Instanz *UNIT* bereit.

- Alle anderen Typen in Typsignaturen von Funktionen, werden als *Showable* dargestellt. *Showable* nutzt einfache Strings zur Umwandlung von Curry-Typen in Java-Typen und umgekehrt. *Showable* enthält eine abstrakte Funktion **public** String show().

  Um einen Curry-Wert in einen String zu verwandeln, wird die *Prelude*-Funktion *show* genutzt. Um den String in einen Curry-Wert zu verwandeln, wird die *ReadShowTerm*-Funktion *readQTerm* genutzt.

  Die Klasse *ShowableObject* implementiert *Showable* mit der *toString*-Methode eines Java-Objekts.

Schlüsselwörter und Sonderzeichen in Funktions- und Modulnamen werden bei der Übersetzung nach Java maskiert.

Alleinstehende Schlüsselwörter wie *null* werden maskiert, indem ihnen ein Unterstrich _ vorangestellt wird. Dies betrifft keine Substrings, wie *notnull*. Erkannt werden folgende Schlüsselwörter:

abstract, **continue**, **for**, **new**, **switch**, assert, **default**, **goto**, **package**, **synchronized**, **boolean**, **do**, **if**, **private**, **this**, **break**, **double**, **implements**, **protected**, **throw**, **byte**, **else**, **import**, **public**, **throws**, **case**, enum, **instanceof**, **return**, **transient**, **catch**, **extends**, **int**, **short**, **try**, **char**, **final**, **interface**, **static**, **void**, **class**, **finally**, **long**, strictfp , **volatile**, **const**, **float**, **native**, **super**, **while**, **null**

Sonderzeichen werden maskiert, indem sie ersetzt werden und ihnen ein Unterstrich _ vorangestellt wird. Sie werden wie folgt übersetzt:

```
'$'  -> "_doll"
'\'' -> "_squo"
'~'  -> "_tild"
'!'  -> "_bang"
'@'  -> "_at"
'#'  -> "_hash"
'%'  -> "_perc"
'^'  -> "_circ"
'&'  -> "_amp"
'*'  -> "_time"
'+'  -> "_plus"
'-'  -> "_min"
'='  -> "_eq"
'<'  -> "_lt"
'>'  -> "_gt"
'?'  -> "_qm"
'.'  -> "_dot"
'/'  -> "_div"
'|'  -> "_bar"
'\\' -> "_bs"
':'  -> "_col"
'['  -> "_lbra"
']'  -> "_rbra"
'('  -> "_lpar"
')'  -> "_rpar"
','  -> "_comm"
'_'  -> "___"
```

Mit diesen Informationen lassen sich Curry-Funktionen in Java-Funktionen umwandeln. Beispielsweise erzeugt die Funktion && aus *Prelude* folgende Interface-Methode:

```
public Iterator<Boolean> _amp_amp(Iterator<Boolean> p1, Iterator<Boolean> p2);
```

Zur Vereinfachung wird zusätzlich noch eine Variante konstruiert, die keinen Nicht-Determinismus in Parametern erlaubt.

```
public Iterator<Boolean> _amp_amp(Boolean p1, Boolean p2);
```

Der potentielle Nicht-Determinismus in Rückgabewerten kann nicht verhindert werden.

Typparameter in Curry werden in Java ebenfalls als Typparameter dargestellt. Als Beispiel sei die Schnittstellen-Methode für *map* gegeben:

```
public <T1, T2> Iterator<List<T2>> map(Function<T1, T2> p1, List<T1> p2);
```

Dabei ist zu beachten, dass die Typinformationen zu $T1$ und $T2$ beim Aufruf durch Type-Erasure verloren gehen. Um einen Parameter in seine Curry-Darstellung zu verwandeln, muss *Cam* also versuchen den richtigen Typ der Werte selbst zu erkennen. Da dies fehlschlagen kann, wird für jede Funktion mit Typvariablen eine weitere Alternative angeboten, bei der die Typen von Typvariablen explizit angegeben werden.

Für die Transformation eines Wertes vom Typ t aus Curry-Darstellung in Java-Darstellung und umgekehrt ist die Klasse cam.bindings.types.*t*Type zuständig. Jede dieser Klassen ist selbst vom Typ Type<*t*>.

```
public <T1, T2> Iterator<List<T2>> map( Type<T1> g1
                                      , Type<T2> g2
                                      , Function<T1, T2> p1
                                      , List <T1> p2);
```

Durch Kombination dieser Alternativen können für jede Curry-Funktion also bis zu vier Java-Methoden erstellt werden.

Schließlich ist noch zu beachten, dass niemals Methoden erstellt werden, die direkt eine Funktion zurückgeben. Sämtliche Funktionen werden möglichst weit entcurry't, also deren Parameter aus dem Ergebnis in die Argumente verschoben. So erzeugt die Curry-Funktion . folgende Methode:

```
public <T1, T2, T3> Iterator<T2> _dot( Function<T1, T2> p1
                                     , Function<T3, T1> p2
                                     , T3 p3);
```

A.8. Externe Funktionen

Implementierungen von externen Funktionen werden in *Cam* in einer Klasse im Paket cam.external erwartet. Die Klasse hat denselben Namen wie das Curry-Modul, gefolgt von einem Unterstrich _.

Jede externe Funktion hat die Form **public static void** f__<Funktionsname>(Node n). Dabei ist *Funktionsname* der Name der Funktion nach den oben genannten Regeln maskiert.

Der übergeben Parameter ist der Funktionsknoten. Nach Ausführung der Methode sollte der Knoten in der Form sein, wie es die externe Funktion erfordert.

Das Programm *GenExtSkeleton* aus ./codegen/Cam kann dazu verwendet werden, aus einem Curry-Modul eine Klasse für die Implementierungen von externen Funktionen zu generienen. Dazu muss dem Programm das entsprechend Modul als Parameter übergeben werden.

Die Ausgabe wird nach ./cam/external/<Modulname>_.java.new geschrieben.

A.9. Beispiele

Im Ordner ./examples werden einige Beispielprogramme angegeben. Sowohl reine Curry-Beispiele, als auch Beispiele für Java-Code, der Curry-Code nutzt. Das Skript ./examples/test.sh führt alle Beispiele aus.

A.10. Einschränkungen

Die Übersetzung mit *makeCam* erlaubt keine Sonderzeichen in Modulnamen. Dies funktioniert nur mit direktem Aufruf des Übersetzers.

Freie Variablen werden in diesem Übersetzungsschema als Generatoren übersetzt. Generatoren für Literale werden von *Cam* nicht unterstützt. Somit funktionieren auch keine funktionalen Parameter mit Literaltypen.

Nicht alle externen Funktionen sind in der aktuellen Version implementiert. Beim Aufruf von nicht implementierten Funktionen bricht die Ausführung mit einer Fehlermeldung *niy* (Not Implemented Yet) ab.