Generische Übersetzung von Curry-Programmen in imperative Programme

Marc André Wittorf

Masterarbeit
Oktober 2018

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

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

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Es existieren bereits einige Compiler, die von der funktional-logischen Sprache *Curry* in eine imperative Sprache übersetzen können. Diese nutzen in der Übersetzung zwar ein Zwischenformat, dieses unterscheidet sich aber zwischen den Compilern. In dieser Arbeit wird ein solches Zwischenformat *ICurry* (für *imperatives Curry*) vorgestellt, das von einer Zielsprache unabhängig ist und so die Übersetzung in verschiedene Zielsprachen erlaubt. Für die Übersetzung wird das *Fair Scheme* als Ausführungsmodell zugrunde gelegt, das eine vollständige und korrekte Ausführung funktional-logischer Programme beschreibt. Das Zwischenformat wird dabei einfach gehalten, damit ein Backend für eine Zielsprache mit wenig Aufwand entwickelt werden kann. Um die Korrektheit und Allgemeinheit des Formats nachzuweisen, werden gleich zwei Backends, `curry2python` zur Übersetzung nach *Python* sowie `curry2c` zur Übersetzung nach *C* vorgestellt. Diese erzeugen zwar nur durchschnittlich schnelle (`curry2c`) oder sogar eher langsame (`curry2python`) Zielprogramme, sind dafür aber kleiner als die Backends der existierenden Compiler von Curry in imperative Sprachen.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziele . . . 1
- 1.3 Nicht-Ziele . . . 2
- 1.4 Verwandte Arbeiten . . . 2
- 1.5 Struktur . . . 3

2 Einführung Curry **5**
- 2.1 Syntax . . . 5
- 2.2 Seiteneffektfreiheit und referenzielle Transparenz . . . 7
- 2.3 Laziness . . . 7
- 2.4 Nichtdeterminismus . . . 8
- 2.5 Call-Time Choice . . . 9
- 2.6 FlatCurry . . . 10

3 The Fair Scheme **13**
- 3.1 Einschub: Graphnotation . . . 13
- 3.2 Programmzustand als Graph . . . 14
- 3.3 Choice-Nichtdeterminismus durch Pull-Tabbing . . . 15
- 3.4 Pull-Tab Schritte und Call-Time Choice . . . 17
- 3.5 Nichtdeterminismus durch freie Variablen . . . 18
- 3.6 Generatoren, Literale und Unifikation . . . 19
- 3.7 Generatoren und Call-Time Choice . . . 19

4 Das Zwischenformat ICurry **23**
- 4.1 Grammatik und Beschreibung von ICurry . . . 23
- 4.2 Transformationen von FlatCurry zu ICurry . . . 27
  - 4.2.1 Let-Ausdrücke . . . 27
  - 4.2.2 Case-Ausdrücke . . . 28
  - 4.2.3 Funktionsweise . . . 29
- 4.3 Typparameter für Generatoren . . . 30
- 4.4 Semantik für ICurry-Laufzeitsystem . . . 34
  - 4.4.1 Ausgangszustand, Endzustand . . . 35
  - 4.4.2 Regeln . . . 35
- 4.5 Semantik für übersetzte ICurry-Funktionen, in Umgangssprache . . . 38
- 4.6 Semantik für übersetzte ICurry-Funktionen, annähernd formell . . . 39
- 4.7 Semantik für ICurry-Typen . . . 41

4.8 Extended ICurry 41
4.9 Ninja als Buildsystem 43

5 Das Python-Backend **45**
5.1 Graphdarstellung 45
5.2 Übersetzung von ICurry nach Python 46
5.2.1 Übersetzung unterscheidender Blöcke 47
5.2.2 Übersetzung unbedingter Blöcke 48
5.2.3 Rettung unverarbeiteter Argumente 49
5.2.4 Übersetzung von Datentypen 49
5.3 Organisation des Laufzeitsystems 50
5.4 Anbindung externer Funktionen 51
5.5 Hilfsfunktionen für Interoperabilität 52

6 Das C-Backend **55**
6.1 Graphdarstellung 55
6.2 Speicherverwaltung 57
6.3 Umgang mit Einschränkungen durch die Sprache C 58
6.4 Organisation des Laufzeitsystems 59
6.5 Hilfsmakros für Interoperabilität 60

7 Evaluation **63**
7.1 Geschwindigkeit 63
7.1.1 Funktionale Benchmarks erster Ordnung 64
7.1.2 Funktionale Benchmarks höherer Ordnung 65
7.1.3 Logisch-Funktionale Benchmarks 66
7.2 Vollständigkeit ohne Terminierung 66
7.3 Implementationsaufwand eines Backends 67

8 Ausblick **71**

A Nutzungsanleitung **73**
A.1 Installation 73
A.2 Schnellstart 74
A.3 Nutzung von Curry-Funktionen aus Python 74
A.4 Nutzung von Curry-Funktionen aus C 74
A.5 Anbindung eigener externer Funktionen 74
A.5.1 Python 75
A.5.2 C 75

B Implementation Guide **77**
B.1 Runtime System . . . 77
B.1.1 Data Structures . . . 77
B.1.2 Memory Management . . . 79
B.1.3 Procedures . . . 80
B.2 The Backend . . . 83
B.2.1 (Extended) ICurry Structure . . . 84
B.2.2 Data Types . . . 84
B.2.3 Functions . . . 84
B.3 Input/Output . . . 85
B.3.1 `catch` and the World . . . 86
B.3.2 Starting an IO Action . . . 86
B.4 External Functions . . . 86
B.4.1 `Prelude.unshare :: a -> a` . . . 86
B.4.2 `Prelude.apply :: (a -> b) -> a -> b` and similar . . . 87
B.4.3 `(Prelude.=:=) :: a -> a -> Bool` . . . 87
B.4.4 `(Prelude.$!!) :: (a -> b) -> a -> b` and `toNF :: a -> a` . . . 88
B.4.5 `Prelude.readFile :: String -> IO String` . . . 88
B.4.6 The `Global` Module . . . 88
B.4.7 `IO.Handle` . . . 89

Bibliografie **91**

Abbildungsverzeichnis

3.1 Sharing im Aufruf von `double 21` 14
3.2 Zyklus im Graph von `fortytwos = 42:fortytwos` 14
3.3 Beispiel Pull-Tab Schritt 16
3.4 Pull-Tab ergeben Run-Time Choice Semantik 17

Tabellenverzeichnis

7.1 Ergebnisse funktionaler Benchmarks erster Ordnung 64
7.2 Ergebnisse funktionaler Benchmarks höherer Ordnung 65
7.3 Ergebnisse logisch-funktionaler Benchmarks 66
7.4 Anzahl Codezeilen verschiedener Backends 68

Kapitel 1

Einleitung

Curry ist eine deklarative Programmiersprache. Sie verbindet funktionale und logische Aspekte und erlaubt es, diese in einer Syntax darzustellen, die großteilig identisch mit der Haskells ist.

Als deklarative Programmiersprache, die gleich mehrere Paradigmen (funktional *und* logisch) unterstützt, bietet Curry eine hohe Ausdrucksstärke. Damit ist es einerseits möglich, Programme sehr kurz und nah an ihrer Problemstellung zu formulieren, wodurch die Entwicklung schneller und die Wartbarkeit leichter sein soll als mit weniger ausdrucksstarken Sprachen. Andererseits vermeiden eine automatische Speicherverwaltung oder ein starkes Typsystem ganze Klassen von Programmfehlern, indem der Programmierer sich nicht selbst um Speicherfreigabe kümmern muss oder Fehler bereits zum Zeitpunkt des Kompilierens gefunden werden können.

1.1. Motivation

Curry bietet beim Entwickeln eines Programms einige Vorteile. Dennoch wäre es unrealistisch zu erwarten, dass alle Programme in Curry entwickelt werden. Gerade bei der Verfügbarkeit von Bibliotheken oder Werkzeugen sind viele imperative Sprachen besser aufgestellt. Daher sind viele Programme ebenfalls in einer imperativen Sprache geschrieben, sodass eine Weiterentwicklung auch fortgehend in dieser Sprache erfolgt. Nicht zuletzt gibt es Anwendungsbereiche, in denen eine imperative Programmiersprache zu bevorzugen ist. Systemprogrammierung oder Programme zum Hochleistungsrechnen benötigen oder wünschen direkten Zugriff auf die verfügbaren Ressourcen, von denen deklarative Sprachen zu abstrahieren versuchen.

Die Möglichkeit, Curry-Programme oder -teilprogramme in eine imperative Sprache zu übersetzen, erlaubt es, die Vorteile beider Seiten zu verknüpfen. Gerade auch bestehende Programme, deren Neuentwicklung in Curry aus verschiedenen Gründen nicht möglich ist, können so in neuen Programmteilen von der Ausdrucksstärke profitieren.

1.2. Ziele

Das Ziel dieser Arbeit ist es, einen Rahmen zum Übersetzen von Curry-Programmen in imperative Programme zu schaffen.

Als Kern dieses Übersetzungsvorgangs soll ein Zwischenformat namens *ICurry* spezifiziert werden. Dieses soll soweit vollständig sein, dass jedes korrekte Curry-Programm in ein äquivalentes ICurry-Programm übersetzt werden kann. Die Vollständigkeit bezieht sich dabei auf den Curry-Sprachumfang während der Entwicklung dieses Formats. Insbesondere die in Curry relativ neuen Typklassen sollen unterstützt werden. Dabei soll es unabhängig von der Zielsprache möglichst klein und einfach sein, also nur die nötigsten Konstrukte anbieten. Schließlich soll dieses Format als Schnittstelle zu weiteren Backends, die auf dieser Arbeit aufbauen, funktionieren.

Damit ICurry genutzt werden kann, soll ein Programm entwickelt werden, das Curry nach ICurry übersetzt. Auch soll neben der üblichen Programmdokumentation eine zusätzliche Entwicklerdokumentation verfasst werden, die sich abstrakter mit den definierten Schnittstellen beschäftigt und bei der Entwicklung eines neuen Backends helfen soll, ohne dass man sich zuvor in die Details der generischen ICurry-Implementierung einarbeiten muss.

Um zu demonstrieren, dass das ICurry-Format hinreichend allgemein ist, sollen zwei weitgehend vollständige Backends entwickelt werden. Ein Backend soll nach Python übersetzen, als Beispiel für eine moderne, relativ mächtige imperative Sprache mit vielen unterstützten Sprachkonstrukten und automatischer Speicherverwaltung. Ein weiteres Backend soll nach C übersetzen, um zu zeigen, dass auch ältere Sprachen mit weniger Abstraktionen geeignet sind, Zielsprache einer Curry-Übersetzung zu sein.

Alle entwickelten Teile sollen modular sein und, sofern möglich, das curryeigene Paketsystem *CPM* [Obe16] verwenden, um die Entwicklung eines neuen Backends im Rahmen eines bekannten Ökosystems zu erlauben.

1.3. Nicht-Ziele

In dieser Arbeit soll kein vollständiges Curry-System erarbeitet werden, das etwa mit *PAKCS* [Han18] oder *KiCS2* [BHP+11] konkurrieren könnte. Der Schwerpunkt liegt weniger darauf, Curry-Programme mit diesen Programmen zu erarbeiten, sondern mehr darauf, zuvor erarbeitete Curry-Programme bzw. -bibliotheken im Kontext anderer, nicht in Curry entwickelter Programme einsetzen zu können.

Auch wird bei der Entwicklung kein besonderes Augenmerk auf die Ausführungsgeschwindigkeit der übersetzten Programme gelegt. Zusätzliche Optimierungen sind oft von zusätzlichen Informationen abhängig, die im Zwischenformat abgebildet sein müssten. Da ein primäres Ziel dieser Arbeit ist, die Entwicklung eines neuen Backends möglichst weit zu vereinfachen, indem möglichst wenige Konstrukte benötigt werden, wäre das Einführen neuer Konstrukte zur Leistungssteigerung kontraproduktiv.

1.4. Verwandte Arbeiten

In der jüngeren Vergangenheit sind in einigen Arbeiten Programme zur Übersetzung von Curry in imperative Sprachen (bzw. Strukturen mit imperativem Charakter) entstanden. All

diese Arbeiten bauen auf der gleichen Idee zur Ausführung, dem *Fair Scheme*, [AJ13] auf.

Der Compiler *SPRITE* [AJ16] ist in direkter Folge zur theoretischen Arbeit des Fair Scheme entstanden und behauptet von sich, der erste vollständige Curry-Compiler zu sein, wobei sich diese Vollständigkeit primär auf die vom Fair Scheme gegebene Vollständigkeit beim Nichtdeterminismus bezieht. Er übersetzt Curry in die LLVM Zwischensprache [1]. Diese Zwischensprache ist Teil des LLVM (früher engl. *Low Level Virtual Machine*) -Compilerframeworks und kann bereits in Maschinencode vieler Architekturen übersetzt werden.

Der Compiler *Cam* [Kir17] wurde im Rahmen einer Masterarbeit entwickelt und übersetzt von Curry nach Java. Dabei wird ein Zwischenformat genutzt, das ähnlich zu dem in dieser Arbeit verwendeten ist.

Hurry [Sik17] ist ebenfalls in einer Masterarbeit entstanden und nutzt das gleiche Zwischenformat wie *Cam*, um Curry-Programme nach JavaScript zu übersetzen. Es bietet darüber hinaus Komponenten, um ein interaktives Curry-System, das direkt im Browser arbeitet, bereitzustellen.

1.5. Struktur

Nach dieser Einleitung wird in Kapitel 2 die Sprache Curry vorgestellt und dabei einige Besonderheiten näher erklärt, die bei der Übersetzung eigener Beachtung bedürfen. Dann wird in Kapitel 3 das *Fair Scheme* erläutert, das das Ausführungsmodell beschreibt, das dem Übersetzungsprozess in dieser Arbeit zugrunde liegt. Es folgt eine genaue Definition des Zwischenformats *ICurry*, das eine leichte Übersetzung in verschiedene Zielsprachen ermöglicht, zusammen mit einigen Schritten, um von einem Curry-Programm zu einem ICurry-Programm zu gelangen. Anschließend werden in den Kapiteln 5 und 6 einige Entscheidungen bei dem Entwurf der Backends für die Zielsprachen *Python* und *C* erklärt. Kapitel 7 evaluiert diese beiden Backends unter den Gesichtspunkten der Ausführungsgeschwindigkeit sowie der Komplexität der Übersetzungen von Zwischenformat zu Zielsprache und stellt Vergleiche zu bereits existierenden Curry-Systemen an. Schließlich gibt Kapitel 8 einen Ausblick.

Es finden sich eine Kurzanleitung zur Benutzung der beiden Backends in Anhang A sowie eine ausführlichere Anleitung zur Entwicklung eines neuen Backends in Anhang B.

[1] LLVM Language Reference Manual: `https://llvm.org/docs/LangRef.html`, zuletzt abgerufen: 2018-10-07

Kapitel 2

Einführung Curry

Curry ist eine funktional-logische Sprache. Das bedeutet, dass Curry sowohl funktionale, wie auch logische Eigenschaften aufweist. Die funktionale Seite ist dabei funktionalen Sprache *Haskell* entlehnt. Abgesehen von den Konstrukten, die zur Unterstützung logischer Programmierung existieren, sowie der fehlenden Möglichkeit, Infixkonstruktoren anzugeben, ist die Syntax identisch zu der Haskells [Han16]. Auch die Seiteneffektfreiheit oder die bedarfsgesteuerte Auswertung mit der Unterstützung von Sharing sind Merkmale, in denen sich Haskell und Curry gleichen. Als logische Eigenschaft unterstützt Curry Nichtdeterminismus, wie er in Sprachen wie Prolog zu finden ist.

2.1. Syntax

Curry-Programme sind in Modulen organisiert. Ein Modul wird durch eine Zeile

```
module MyModule where
```

eingeleitet. Danach folgen Angaben, aus welchen anderen Modulen Funktionen und Datentypen verwendet werden, wie beispielsweise:

```
import List
import Maybe(fromJust)
import qualified IO
```

Anschließend folgen die Definitionen der Funktionen und Datentypen dieses Moduls. Eine Funktion besteht dabei aus einer optionalen Signatur und einer oder mehreren Regeln. Eine Regel hat eine linke und eine rechte Seite. Die linke Regelseite enthält dabei Muster. Sie bestimmen, ob eine Regel angewandt werden kann, wenn ein Funktionsaufruf ausgewertet wird. Außerdem bindet sie die Funktionsargumente oder Teile dieser an Variablen. Die rechte Regelseite ist ein Ausdruck. Dieser wird an die Stelle des Funktionsaufrufs gesetzt, falls die Regel anwendbar ist. Da Curry seiteneffektfrei ist, existieren keine Anweisungen.

```
foo :: Maybe Int -> Int
foo Nothing  = 42
foo (Just x) = x+4
```

In diesem Beispiel zeigt die Signatur der Funktion `foo` an, dass die Funktion ein Argument des Typs `Maybe Int` entgegennimmt und ein `Int` als Ergebnis liefert. Die linken Seiten der beiden Regeln geben an, dass die erste Regel anwendbar ist, falls das Argument einen

Nothing-Konstruktor trägt, und die zweite Regel zum Tragen kommt, falls das Argument einen **Just**-Konstruktor aufweist. Außerdem wird in diesem Fall das Argument des **Just**-Konstruktors an die Variable x gebunden, sodass diese in der rechten Regelseite verwendbar ist. Die rechten Regelseiten sind hier zwei sehr einfache Ausdrücke. Die rechte Seite der ersten Regel wertet zu einem Literal aus, während die der zweiten Regel zu einer Addition eines Literals zu dem entpackten Wert auswertet.

Ausdrücke haben eine rekursive Struktur. Neben den bereits im obigen Beispiel vorkommenden Literalen, Variablen und Funktionsapplikationen (im Fall von + im Beispiel in Infixschreibweise) existieren einige weitere Ausdrücke, die in dieser Arbeit jedoch nicht alle erklärt werden können. Eine vollständige Beschreibung aller Ausdrücke sowie eine genaue Beschreibung der gesamten Syntax sowie Semantik von Curry findet sich im *Curry-Report* [Han16].

Nur wenige dieser Ausdrücke sind notwendig, damit die Sprache berechnungsuniversell ist. Diese Kernkonstrukte werden in Abschnitt 2.6 etwas näher betrachtet. Viele andere Konstrukte stellen nur syntaktischen Zucker dar und lassen sich in diese Kernkonstrukte übersetzen. So existieren beispielsweise *List Comprehensions*, die eine natürliche Notation von Listen erlauben, ein spezielles `if-then-else`-Konstrukt, das eine Fallunterscheidung anbietet und etwa dem ternären Operator vieler imperativer Sprachen entspricht, oder die `do`-Notation, durch die monadische Verkettungen ähnlich zu Anweisungen imperativer Sprachen notiert werden können.

Neben Funktionen kennt Curry Datentypen. Diese bestehen aus einem Namen, einer Liste von Typparametern und einem oder mehreren Konstruktoren. Die Datentypdefinition

```
data Maybe a = Nothing
             | Just a
```

findet sich beispielsweise in der `Prelude` der Curry-Standardbibliothek. Dabei weist der Typ **Maybe** ein Typargument auf, auf das mit der Typvariable `a` verwiesen wird. Er hat zwei Konstruktoren. **Nothing** ist ein nullstelliger und **Just** ein einstelliger Konstruktor, wobei das Argument, das **Just** anhängt, von dem Typ sein muss, über den **Maybe** parametrisiert wird.

Seit kurzem unterstützt Curry auch Typklassen, wie sie in Haskell verwendet werden können [Tee16]. Diese geben eine Eigenschaft vor, die ein Typ haben kann. Durch die Angabe einer Instanz kann dann definiert werden, wie ein Datentyp der durch eine Typklasse modellierten Eigenschaft entspricht.

In einer Typklasse wird eine Menge von Funktionen festgelegt, die als Schnittstelle verstanden werden kann, über die mit Daten der unterstützten Typen umgegangen werden kann. Für diese wird meist nur eine Signatur, aber keine Funktionsregeln angegeben. Eine Instanz der Typklasse für einen Typ enthält dann für jede so vorgegebene Funktion die jeweils für diesen Typ zu verwendende Implementierung dieser Funktion. Es können auch Funktionsregeln in der Typklasse selbst angegeben werden. Diese werden dann als Standardimplementierung angesehen und verwendet, falls eine Instanz keine eigene Implementierung der jeweiligen Funktion beinhaltet.

Zum Beispiel ist die Typklasse **Eq** der `Prelude` so angegeben:

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

  x == y = not (x /= y)
  x /= y = not (x == y)
```

Diese gibt bereits für beide Funktionen eine Implementierung an, die sich auf die jeweils andere Funktion stützt. So genügt es, eine beliebige der beiden Funktionen zu implementieren, um eine vollständige Instanz zu erhalten. Dies wird in der Instanz für den Typ `Int` ausgenutzt:

```
instance Eq Int where
  i == i' = i `eqInt` i'
```

`eqInt` ist dabei eine Funktion, die zwei Werte vom Typ `Int` auf Gleichheit überprüft und an anderer Stelle definiert ist.

2.2. Seiteneffektfreiheit und referenzielle Transparenz

Programmierung in Curry ist vollständig frei von Seiteneffekten. Das bedeutet, dass keine Funktion auf globalen Zustand zugreift oder ihn verändert. Damit ist eine Funktion vollständig durch ihr Ergebnis in Abhängigkeit ihrer Parameter beschrieben. Insbesondere darf ein Funktionsaufruf zu jeder Zeit durch das Ergebnis dieses Aufrufs ersetzt werden, ohne die Semantik des Programms zu ändern. Diese Eigenschaft wird auch *referenzielle Transparenz* genannt. Sie erlaubt es, einen Teilausdruck, der an mehreren Stellen in einem Ausdruck vorkommt, zwischen diesen Vorkommen zu teilen (engl. *Sharing*). Dadurch genügt es, diesen Teilausdruck nur ein einziges Mal auszuwerten, anstatt dies für jedes Vorkommen tun zu müssen. Das erlaubt es, mehrfache Berechnungen eines gleichen Ergebnisses zu vermeiden und kann so eine Leistungsverbesserung des Programms erreichen.

Dies passiert zum Beispiel in dem Ausdruck `let x = fib 10000 in x + x`. Ohne Sharing würde die 10000. Fibonacci-Zahl jeweils ein mal für die linke und die rechte Seite des + berechnet werden, obwohl klar ist, dass das Ergebnis dieser beiden Berechnungen gleich sein wird. Mit Sharing wird diese Fibonacci-Zahl nur einmal berechnet. Da der Berechnungsaufwand einer einzigen Addition im Verhältnis zur Berechnung der Fibonacci-Zahl vernachlässigbar ist, wird dieser Ausdruck durch Sharing um einen Faktor zwei schneller ausgeführt.

2.3. Laziness

Genau wie das Auswerten mehrerer gleicher Teilausdrücke durch die referenzielle Transparenz vermieden werden darf, so kann auch ganz auf die Auswertung eines Teilausdrucks verzichtet werden, falls das Ergebnis dieses Teilausdrucks nicht benötigt wird. Curry nutzt dies aus und wertet generell immer nur die Teilausdrücke aus, deren Ergebnisse als nächstes benötigt werden. Dies wird auch bedarfsgesteuerte Auswertung oder *lazy* Auswertung genannt.

Dadurch werden nicht nur unnötige Auswertungen vermieden, sondern es sind einige Programmiertechniken möglich, die bei einer strikten Auswertung nicht möglich sind, ohne eine solche bedarfsgesteuerte Auswertung explizit zu simulieren. Beispielsweise kann problemlos mit unendlichen Datenstrukturen umgegangen werden:

```
myNumbers :: [Int]
myNumbers = 1 : map (+1) myNumbers

firstTenOddNumbers :: [Int]
firstTenOddNumbers = take 10 $ filter odd myNumbers
```

Obwohl `myNumbers` eine durch Selbstverweis unendliche Liste ist, terminiert ein Aufruf von `firstTenOddNumbers`. Durch die *lazy* Auswertung werden überhaupt nur diejenigen Elemente aus `myNumbers` ausgewertet, die benötigt werden. Alle übrigen Elemente der Liste werden als unendlicher Rest unausgewertet gelassen.

Auch ist eine eigene Definition einer Fallunterscheidung möglich:

```
myIf :: Bool -> a -> a -> a
myIf True  t _ = t
myIf False _ f = f
```

In einer strikten Sprache würde eine solche Definition nicht vermeiden können, dass der jeweils ungewünschte Teilausdruck ausgewertet würde. So könnte beispielsweise eine Fixpunktiteration nie terminieren, da sie sich selbst in einer Endlosschleife auch dann weiter aufruft, wenn der Fixpunkt bereits gefunden ist:

```
fix :: (a -> Bool) -> (a -> a) -> a -> a
fix p f x = myIf (p x) x (fix p f (f x))
```

Dabei bezeichnet das Argument `p` ein Prädikat, das prüft, ob der Fixpunkt bereits erreicht ist, `f` berechnet die nächste Iteration und `x` ist der Wert der jeweiligen Iteration.

2.4. Nichtdeterminismus

Aus der logischen Programmierung übernimmt Curry Nichtdeterminismus. Dieser Nichtdeterminismus kann in Curry auf zwei Arten eingeführt werden. Einerseits können Funktionen so gestaltet sein, dass für einen Aufruf mehrere der Programmregeln anwendbar sind. Dies geschieht dadurch, dass die Muster der linken Regelseiten sich überlappen. Bei Haskell würde in diesem Fall nur die erste passende Regel ausgewählt. In Curry aber werden *alle* passenden Regeln angewandt, sodass mehrere mögliche Ergebnisse entstehen, mit denen weitergearbeitet wird. Dies zeigt die Funktion `foo`:

```
foo :: Int -> Int
foo 1 = 42
foo 2 = 21
foo x = x
```

Im Aufruf `foo 1` passt sowohl die erste, als auch die dritte Regel. Damit wertet der Ausdruck sowohl zu `42` über die erste Regel, als auch zu `1` über die dritte Regel aus.

Andererseits können freie Variablen verwendet werden. Durch diese kann mit unbekannten Werten gerechnet werden, indem das Curry-System alle möglichen Werte rät und anschließende Berechnungen mit jeder dieser geratenen Möglichkeiten durchführt. Im Ausdruck `if x then 1 else 2 where x free` werden für die freie Variable `x` alle Werte ihres Typs `Bool`, also `False` und `True` geraten. Damit ergeben sich die Ergebnisse `2` für `x = False` und `1` für `x = True`.

Häufig enthält die Menge aller möglichen Werte für eine freie Variable viele solcher Werte, über die leicht ersichtlich ist, dass sie nicht zielführend sind. Daher werden freie Variablen zusätzlich unifiziert. Im Ausdruck `cond (x =:= Just y) (x,y) where x,y free` wird die Funktion `=:=` verwendet. Diese führt einen strukturellen Vergleich der beiden Argumente durch und liefert nur `True` im positiven Fall, führt aber zu einem Fehlschlag im negativen Fall. So kann sie durch Unifikation anhand des `Just` des rechten Arguments feststellen, dass auch das linke Argument den Konstruktor `Just` tragen muss, und so kann darauf verzichtet werden, `Nothing` für `x` zu probieren. Da `cond` bei `True` im ersten Argument das zweite Argument zurückliefert, wird dieser Ausdruck zu den Ergebnissen `(Just False, False)` und `(Just True, True)` ausgewertet.

2.5. Call-Time Choice

Bei der Auswertung eines Ausdrucks, der nichtdeterministische Teilausdrücke enthält, ist in manchen Fällen nicht klar, welche nichtdeterministisch gewählten Teilergebnisse verwendet werden sollen. Sei beispielsweise diese nichtdeterministische, nullstellige Funktion gegeben:

```
coin :: Int
coin = 1
coin = 2
```

Da keine ihrer zwei Regeln Muster enthält, ist sie trivialerweise nichtdeterministisch und kann zu `1` und `2` ausgewertet werden.

Soll der Ausdruck `2 * coin` ausgewertet werden, so ist offensichtlich, dass die Ergebnisse `2` und `4` entstehen können. Soll der Ausdruck `coin + coin` ausgewertet werden, so können `2`, `3`, `3` und `4` entstehen. Das ergibt sich aus der Tatsache, dass sowohl das linke, als auch das rechte `coin` natürlich zu beiden Ergebnissen ausgewertet werden können. Für den Ausdruck `let x = coin in x + x` können nun zwei mögliche Ausgänge erwartet werden. Einerseits kann erwartet werden, dass nur `2` und `4` mögliche Ergebnisse sind. Dies gebietet schon das mathematische Verständnis: Es gilt bekanntlich $x + x = 2 * x$ und oben ist bereits festgestellt worden, dass `2 * coin` ebendiese Ergebnisse liefert. Andererseits kann auch auf die zusätzliche Variable `x` verzichtet werden und das `coin` aus der Bindung direkt an alle Stellen geschrieben werden, an denen `x` zuvor verwendet wurde. Dann ergibt sich aber wieder `coin + coin`, das zu vier Ergebnissen ausgewertet wird.

Diese beiden Möglichkeiten werden *Call-Time Choice* und *Run-Time Choice* genannt. Curry unterstützt die *Call-Time Choice*. Die Namen ergeben sich daraus, wann Unterscheidungen über nichtdeterministische Ausdrücke stattfinden. Unter Annahme von *Call-Time Choice* Semantik wird ein nichtdeterministischer Ausdruck entschieden und damit in alle (nun deterministischen) Möglichkeiten aufgeteilt, wo er im Quellprogramm notiert ist, wo also sein Aufruf (engl. *call*) zu finden ist. Damit löst bereits die **`let`**-Bindung den Nichtdeterminismus auf, sodass im obigen Beispiel `x` immer nur deterministische Werte, also `1` *oder* `2`, nicht aber nichtdeterministisch beide Werte trägt. Durch *Run-Time Choice* Semantik wird Nichtdeterminismus an der Stelle aufgelöst, wo ein Ausdruck ausgewertet (oder ausgeführt, engl. *run*) wird. So erhielte `x` im obigen Beispiel nichtdeterministisch sowohl `1` als auch `2`. Erst wenn die Funktion `+` die Auswertung des Ausdrucks fordert, wird dieser Nichtdeterminismus in die beiden Werte aufgespalten. Dann besteht kein Zusammenhang zwischen dem Nichtdeterminismus des linken und dem des rechten Ausdrucks mehr, also können diese auch unterschiedlich ausgewertet werden, sodass zusätzliche Kombinationen zusätzliche Ergebnisse liefern können.

Da Curry *Call-Time Choice* unterstützt, ist Sharing in Curry nicht nur zur Effizienzsteigerung, sondern auch zur Wahrung der Korrektheit notwendig, da nicht nur die Ausdrücke, sondern auch die Auswahl eines nichtdeterministischen Wertes zwischen den Stellen geteilt werden muss.

2.6. FlatCurry

Viele Curry-Werkzeuge arbeiten nicht auf dem ursprünglichen Curry-Quelltext oder einer abstrakten Darstellung dieses Quellprogramms. Sie verwenden stattdessen eine bereits stark vereinfachte Darstellung der Sprache, *FlatCurry*. Damit können diese Werkzeuge darauf verzichten, mit jedem möglichen Konstrukt umgehen können zu müssen. Das *Curry-Frontend* übersetzt dazu das ursprüngliche Curry-Programm in ein FlatCurry-Programm.

FlatCurry definiert eine abstrakte Syntax. Diese enthält die Informationen des Modulkopfes, eine Liste von Datentypen und eine Liste von Funktionen. Dabei ist anders als in Curry allerdings eine Reihenfolge vorgegeben, sodass alle Datentypen vor allen Funktionen angegeben sind. Typklassen sind in der FlatCurry-Darstellung nicht mehr vorhanden. Eine Typklasse wird stattdessen zu einem Wörterbuch-Datentyp und Funktionen, die in einem solchen Wörterbuch die jeweilig benötigte Instanz finden [Tee16]. Eine Instanz kann so durch eine Instanz des Wörterbuchs ausgedrückt werden, die die Implementierungen jeder Funktion für den Datentyp, über den die Instanz gebildet wird, beinhaltet.

Ein FlatCurry-Datentyp entspricht genau einem Curry-Datentyp. Allerdings enthält FlatCurry keine Konstruktion für *Record*-Datentypen. Ein solcher wird in FlatCurry durch eine gewöhnliche Datentypdeklaration abgebildet und Zugriffe auf Daten mittels *Record*-Syntax werden auf Zugriffe mittels *Pattern Matching* abgebildet.

Die meisten Unterschiede zwischen Curry und FlatCurry liegen bei den Funktionen. Anders als die einer Curry-Funktion lässt sich die Struktur einer FlatCurry-Funktion durch

nur wenige Konstrukte beschreiben.

```
data FuncDecl = Func QName Arity Visibility TypeExpr Rule
```

Eine Funktion hat neben einigen zusätzlichen Eigenschaften hauptsächlich einen Namen und genau eine Regel.

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

Diese Regel hat, wenn sie keine extern implementierte Funktion ist, eine Liste von Variablen, die die Argumente der Funktion binden, und einen Ausdruck, zu dem diese Funktion ausgewertet wird.

```
data Expr = Var VarIndex
          | Lit Literal
          | Comb CombType QName [Expr]
          | Let [(VarIndex,Expr)] Expr
          | Free [VarIndex] Expr
          | Or Expr Expr
          | Case CaseType Expr [BranchExpr]
          | Typed Expr TypeExpr
```

`Var` verweist dabei auf eine Variable, die in der Regel oder durch `Let` oder `Free` eingeführt wurde. `Lit` stellt ein Literal dar, der Typ `Literal` erlaubt dabei eine Unterscheidung über die Basistypen **`Int`**, **`Char`** und **`Float`**. `Comb` stellt eine vollständige oder partielle Funktions- oder Konstruktorapplikation dar. Dabei zählt der Typ `CombType` diese Möglichkeiten auf. Die Funktion oder der Konstruktor wird durch ihren oder seinen Namen identifiziert. `Let` führt eine oder mehrere Bindungen von Ausdrücken an Variablen ein. `Free` führt eine neue freie Variable ein. `Or` führt Nichtdeterminismus durch überlappende Regeln ein. `Case` erlaubt eine Fallunterscheidung über einen Ausdruck. `Typed` erlaubt es schließlich, explizit getypte Ausdrücke anzugeben. Die in der Fallunterscheidung angegebenen Fälle werden dabei so ausgedrückt:

```
data BranchExpr = Branch Pattern Expr
data Pattern    = Pattern QName [VarIndex]
                | LPattern Literal
```

Sie unterscheiden so nur über genau einen Konstruktor und binden dabei Variablen.

Da FlatCurry strukturell wesentlich einfacher ist als Curry, führt das Curry-Frontend viele Transformationen durch, um Curry-Strukturen auf Flat-Curry-Konstrukte zu reduzieren. Hier sollen nur wenige dieser Transformationen kurz erklärt werden, um eine Idee von der Arbeit des Curry-Frontends zu erhalten.

Eine wesentliche Einschränkung von FlatCurry gegenüber Curry ist bereits an der Definition von `FuncDecl` und `Rule` zu erkennen. Für jede Funktion kann nur genau eine Regel definiert werden und diese trägt nur Variablen, keine Muster. Damit können Fallunterscheidungen nicht über die Regeln erfolgen. Stattdessen werden mehrere Regeln in einen `Case`-Ausdruck

umgeformt, sodass alle Curry-Regeln in nur einer FlatCurry-Regel vereint werden können. Auch sind die FlatCurry-Muster weniger mächtig als Curry-Muster. Da sie nur einen Konstruktor bezeichnen, können sie auch immer nur über einen Konstruktor unterscheiden. Ein Muster `Just (Just x)` kann so nicht ausgedrückt werden. Stattdessen würde eine Fallunterscheidung mit einem solchen Muster in mehrere verschachtelte Fallunterscheidungen, entsprechend etwa

```
case ... of
  Just a -> case a of
              Just x -> ...
              Nothing -> ...
  Nothing -> ...
```

übersetzt, sodass jeder `Case`-Ausdruck nur über einen einzigen Konstruktor eine Entscheidung trifft.

Das bedeutet auch, dass Nichtdeterminismus nicht durch überlappende Programmregeln entstehen kann. Stattdessen werden solche Überlappungen erkannt und derart übersetzt, dass beide nichtdeterministischen Möglichkeiten durch die Funktion `Prelude.?` verknüpft werden. Die aus Abschnitt 2.5 bekannte `coin`-Funktion würde so etwa in die Funktion

```
coin = 1 ? 2
```

übersetzt.

Außerdem bietet Curry Konstrukte, die nur syntaktischen Zucker darstellen, aber sich selbst vollständig durch andere Konstrukte darstellen lassen. So wird beispielsweise die `do`-Notation in eine Kette von monadischen *Bind-* (`>>=`) oder Verkettungs- (`>>`) Operationen übersetzt. So entsteht aus

```
do
  x <- exp1
  exp2
  y <- exp3
  exp4
```

der Ausdruck:

```
exp1 >>= (\x ->
exp2 >> (
exp3 >>= (\y ->
exp4)))
```

So werden letztendlich im Curry-Frontend alle Curry-Konstrukte auf die wenigen FlatCurry-Konstrukte reduziert. So kann mit wenig Aufwand der gesamte Umfang eines Curry-Programms dargestellt und weiterverarbeitet werden.

Kapitel 3

The Fair Scheme

Das *Fair Scheme* [AJ13] ist ein Ausführungsmodell für funktional-logische Sprachen. Es beschreibt eine Art Breitensuche auf einem Zustandsgraphen, wobei Nichtdeterminismus durch flache Kopien in diesem Graphen aufgelöst wird. Anders als bei Modellen, die konzeptuell einer Tiefensuche entsprechen (also beispielsweise mit *Backtracking* arbeiten), kann so vermieden werden, dass unendliche Berechnungen in einer nichtdeterministischen Auswahl die gesamte Berechnung blockieren und so berechenbare Ergebnisse in anderen Zweigen nie erreicht werden.

3.1. Einschub: Graphnotation

Für formale Argumentation über Graphen wird in dieser Arbeit eine lineare Graphnotation verwendet, die der in [EJ97] vorgestellten stark ähnelt. Die Darstellung ist von der rekursiven Darstellung von Bäumen inspiriert, in der ausgehend von der Wurzel jeder Knoten mit seinen Kindern notiert wird. Jedoch erhält jeder Knoten eine Identität, über die von mehreren Stellen auf ihn verwiesen werden kann, ohne einen Knoten mehrfach notieren zu müssen. Insbesondere können so auch Zyklen dargestellt werden. Es wird dabei immer von einem Knoten ausgegangen, sodass es nicht möglich ist, einen nicht zusammenhängenden Graphen vollständig anzugeben. Dies ist aber nicht weiter störend, da hier ohnehin immer nur eine Zusammenhangskomponente des Graphen von Interesse ist. Knoten, die ihren Zusammenhang mit dieser Komponente verlieren, sind, um etwas vorwegzugreifen, solche, die nicht mehr benötigt werden und von einem Garbage Collector entfernt werden.

Für einen Knoten wird immer eine Identität angegeben, die im Folgenden auch oft Knoten genannt wird. Optional kann dem Knoten, getrennt durch einen Doppelpunkt, eine Beschriftung und dahinter in Klammern eine Liste von Kindknoten folgen.

```
Node ::= n | n:Label(Node, ..., Node)
```

Beschriftung und Kinder werden für jeden Knoten höchstens einmal angegeben, außerdem findet sich diese Information bevorzugt so weit links wie möglich. In dieser Arbeit wird gänzlich darauf verzichtet, Beschriftung und Kinder eines Knotens anzugeben, falls diese für die Argumentation nicht relevant sind. So können Ausschnitte von Graphen gezeigt werden, indem auf Elternknoten der Wurzel verzichtet wird und Nachfahrenknoten nicht näher definiert werden.

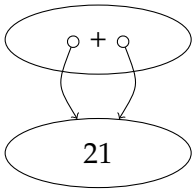

Abbildung 3.1. Sharing im Aufruf von `double 21` wobei im Programm `double x = x+x` definiert ist.

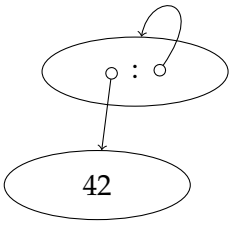

Abbildung 3.2. Zyklus im Graph von `fortytwos = 42:fortytwos`

Beispielsweise wird so der in Abbildung 3.1 dargestellte Graph notiert als $n1 : + (n2 : 21\ (),\ n2)$. Die Beschriftung und leere Kindliste des hier als $n2$ bezeichneten Knotens wird so nur bei dem linken Vorkommen notiert.

3.2. Programmzustand als Graph

Der Programmzustand eines funktional-logischen Programms kann auf den ersten Blick als Term verstanden werden, also als eine baumartige Struktur. Diese Darstellung übersieht aber das *Sharing*, das in funktionalen Sprachen die Leistung verbessert und in funktional-logischen Sprachen zusätzlich das Verhalten des Programmes beeinflussen kann (Call-Time Choice und Run-Time Choice, siehe 2.5). Ein gerichteter Graph ist die natürliche Erweiterung einer Baumstruktur, die Sharing leicht darstellen kann, indem ein Knoten mehreren Elternknoten als Kind zugeordnet ist (Abbildung 3.1). Auch die in *lazy* Sprachen oft zugelassenen, durch Selbstreferenzierung unendlichen Strukturen sind nicht baumförmig, lassen sich aber in einem Graphen leicht als Zyklus ausdrücken (Abbildung 3.2).

Ein solcher Graph enthält beschriftete Knoten. Mögliche Beschriftungen sind ein Literal, ein Konstruktorsymbol, ein Funktionssymbol, eine nichtdeterministische Auswahl (*Choice*) oder ein Fehlschlag. Die ausgehenden Kanten geben für konstruktor- und funktionsbeschriftete Knoten an, mit welchen Argumenten der Konstruktor bzw. die Funktion aufgerufen wird. Für Nichtdeterminismus durch *Choice* geben diese Kanten die nichtdeterministischen Alternativen an. Literale und Fehlschläge haben nie ausgehenden Kanten.

Das Programm selbst ist dabei kein Teil des Zustands. Es stellt stattdessen eine Menge von Transformationen dar, die auf dem Graphen durchgeführt werden. Zusammen mit den vom Fair Scheme vorgegebenen Transformationen zum Umgang mit Nichtdeterminismus und dem Auswertungsschema kann so ein Graph, der einen initialen Ausdruck abbildet, bis

zur Normalform berechnet werden.

Der Begriff einer Wurzel wird für diese Programmzustandsgraphen aus dem Sprachgebrauch über Bäume entlehnt. Eine Wurzel meint in diesem Kontext immer den Knoten, der den gesamten Ausdruck in und unter sich trägt, der ursprünglich ausgewertet werden soll. Dabei gilt weder, dass es nur *eine* Wurzel in dem Graphen gibt, da sich mehrere nichtdeterministische Zweige letztendlich zu mehreren Wurzeln aufspalten, noch gilt, dass eine Wurzel keine Elternknoten hat. Selbstreferenzen können auf den ganzen Ausdruck verweisen, sodass auch der Knoten dieses ganzen Ausdrucks eingehenden Kanten haben kann.

3.3. Choice-Nichtdeterminismus durch Pull-Tabbing

Nichtdeterminismus kommt in Curry auf zwei Arten vor. Die erste Art ist der *Choice*-Nichtdeterminismus. Dieser entsteht in einem Curry-Programm durch die Verwendung von überlappenden Programmregeln, wird aber vom Curry-Frontend so umgeformt, dass er sich durch den *Choice*-Operator (`(?) :: a -> a -> a`) ausdrückt, der dieser Art von Nichtdeterminismus seinen Namen gibt. Er spaltet die Berechnung in zwei Stränge auf, die jeweils einer der beiden angebotenen Alternativen bearbeiten. Die zweite Art sind freie Variablen, die in Abschnitt 3.5 näher erklärt werden.

Mit solchen Spaltungen des Berechnungsstranges lässt sich auf unterschiedliche Weisen umgehen. Eine einfache Weise ist das Backtracking, das auch beispielsweise von PAKCS [Han18] umgesetzt wird. Das funktioniert zwar oft, ist aber nicht vollständig, wenn eine der Alternativen nicht terminiert.

Stattdessen nutzt das Fair Scheme sogenannte Pull-Tab Schritte. Diese erhalten ihren Namen von dem Schieber eines Reißverschlusses (engl. *tab*), der, wenn man ihn herunterzieht (engl. *pull*), den Verschluss Zahn für Zahn in linke und rechte Seite öffnet [AAF+10]. Genauso wird eine Choice heruntergezogen und spaltet dabei die Berechnung in die nichtdeterministisch berechneten (Teil-)Ergebnisse.

Die Funktionalität zum Ausführen eines Pull-Tab-Schritts ist Teil des Laufzeitsystems. Dieses steuert, welche Teilterme ausgewertet werden sollen. Eine Funktion kann derart ausgewertet werden, dass sie eine neue Beschriftung und neue Kindbeziehungen in ihren Knoten schreibt. Eine Choice kann dagegen nicht ausgewertet werden, indem ihr Knoten geändert wird, da dies immer nichtdeterministische Zweige verlieren würde.

Stattdessen setzt ein Pull-Tab Schritt an der Kante zwischen einem funktions- oder konstruktorbeschrifteten Knoten und demjenigen choicebeschrifteten Kindknoten an, dessen Auswertung zur Kopfnormalform gefordert wird. Der Elternknoten wird dabei mit einer Choice beschriftet. Er wird von all seinen Kindknoten getrennt und erhält stattdessen zwei neue Kindknoten, einen für jeden nichtdeterministischen Zweig. Diese neuen Knoten werden jeweils mit der Funktion oder dem Konstruktor beschriftet, die oder den der Elternknoten zuvor trug. Auch die Kindbeziehungen werden fast vollständig aus diesem Knoten übernommen. Lediglich an der einen Position, die vorher auf die Choice, die hier gezogen werden soll, verwiesen hat, erhält jeweils einer der Knoten einen nichtdeterministischen Zweig der Choice,

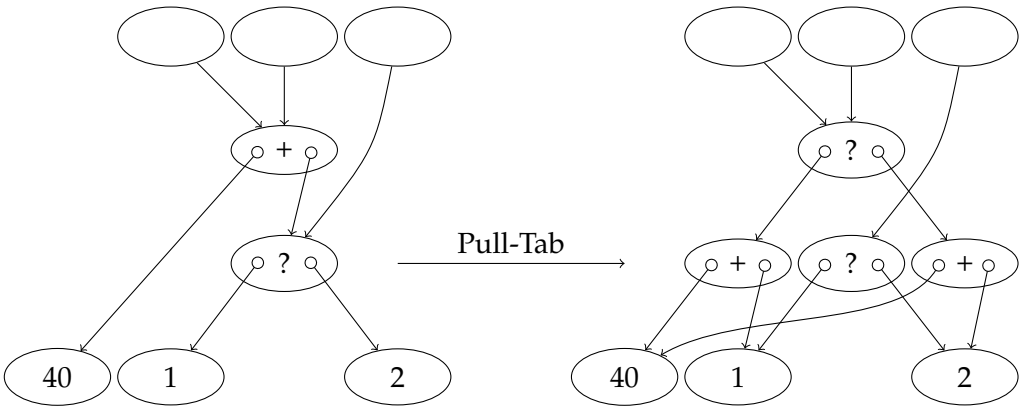

Abbildung 3.3. Beispiel eines Pull-Tab Schritts in einem Teilgraph, der durch den Ausdruck `let x = 40 + (1 ? 2) in ...` entsteht. Die leeren Knoten über dem `(+)`-Knoten deuten an, dass dieser Knoten zwischen mehreren Eltern geteilt wird.

anstatt des choicebeschrifteten Knotens.

Formal führt ein Pull-Tab-Schritt die folgende Graphtransformation durch:

$$n : s(n_0, \ldots, n_i : ?(n_{i,1}, n_{i,2}), \ldots, n_m) \rightarrow$$
$$n : ?(n' : s(n_0, \ldots, n_{i,1}, \ldots, n_m), n'' : s(n_0, \ldots, n_{i,2}, \ldots, n_m))$$

Dabei sind $n, n_0, \ldots, n_m, n_{i,1}, n_{i,2}$ Knoten, die vor dem Schritt im Graph existieren, n', n'' Knoten, die durch den Pull-Tab Schritt neu erzeugt werden, s ein Konstruktor oder eine Funktion und $i \in \mathbb{N}$ die Position, an der die Choice zu finden ist.

Dieser Pull-Tab Schritt verändert (oder entfernt) den Knoten n_i, der die Choice trägt, nicht, da dieser durch Sharing auch an anderer Stelle benötigt werden könnte.

Abbildung 3.3 zeigt, wie ein solcher Pull-Tab Schritt in einem echten Teilgraph aussehen kann. Es ist gut zu sehen, dass durch die Änderungen der Beschriftung des ursprünglichen `(+)`-Knotens keine Elternbeziehung dieses Knotens gestört wird. Ebenfalls ist zu sehen, dass der Schritt den ursprünglichen Choice-Knoten nicht ändert. Lediglich die Verwendung im `(+)`-Knoten wird aufgelöst, die Sicht des nicht näher spezifizierten Knotens auf den Choice-Knoten ist nach dem Pull-Tab Schritt genau wie die Sicht vor dem Schritt. Die Choice kann hier später ebenfalls durch einen Pull-Tab Schritt hochgezogen werden, falls die Ausführung dies verlangt. Nach einem solchen Schritt wäre der Knoten dann unbenutzt und kann bei der nächsten Gelegenheit durch einen Garbage-Collector entfernt werden.

So werden Choices nach und nach bis an eine Wurzel des Graphen gezogen. Dort werden keine Pull-Tab Schritte durchgeführt, da keine Elternbeziehung mehr vorliegt. Stattdessen können die nichtdeterministischen Zweige, die sich als Kinder unter diesem Knoten darstellen, einfach als nichtdeterministische Lösungen des anfänglichen Ausdrucks gesehen werden.

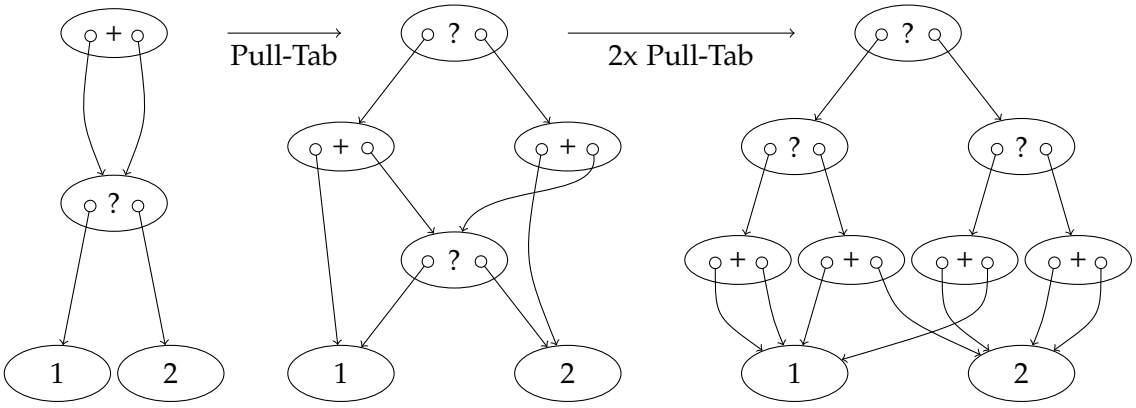

Abbildung 3.4. Beispiel für Pull-Tab Schritte in `let x = (1 ? 2) in x + x`. Die möglichen Ergebnisse entsprechen der Run-Time Choice Semantik.

3.4. Pull-Tab Schritte und Call-Time Choice

Pull-Tab Schritte in dieser Reinform sind noch nicht ausreichend, um den in Curry spezifizierten Nichtdeterminismus abzubilden. Sie berücksichtigen nämlich noch nicht, *wo* der Nichtdeterminismus eingeführt wird, der gerade behandelt werden soll. So wird jede Choice völlig unabhängig behandelt, auch wenn sie durch einen Pull-Tab Schritt aus einer anderen Choice entstanden ist und so in jedem Zweig in ihrer möglichen Auswahl eingeschränkt sein müsste. Pull-Tab Schritte bilden ohne zusätzliche Maßnahmen also die Semantik der Run-Time Choice ab. Dies wird durch das Beispiel in Abbildung 3.4 veranschaulicht. Der Ausdruck `let x = (1 ? 2) in x + x` wertet zu den Ergebnissen 2, 3, 3 und 4 aus, obwohl nach der Call-Time Choice Semantik in Curry nur 2 und 4 hätten berechnet werden dürfen. Dabei wird von der ersten zur zweiten Ansicht die Choice über die linke Kante zwischen Plus- und Choice-Knoten gezogen. Zwischen der zweiten und dritten Ansicht werden zwei Pull-Tab Schritte durchgeführt, jeweils über die rechte, vom jeweiligen Plus-Knoten ausgehende Kante. Außerdem zeigt die dritte Ansicht den ursprünglichen Choice-Knoten nicht mehr, da er von keiner Wurzel mehr erreicht werden kann und so keine Bedeutung mehr hat.

Um Call-Time Choice Semantik in dieses System zu bringen, kann jeder Choice eine Identität gegeben werden. Wird eine Choice durch einen Pull-Tab Schritt in Richtung der Wurzel gezogen, so bekommt die dadurch entstehende, neue Choice die gleiche Identität. Beim Auswerten können dann Pfade verworfen werden, die eine so identifizierte Choice bei mehreren Gelegenheiten unterschiedlich auswählen.

Im Beispiel hätten alle Choices dieselbe Identität, da sie alle durch Pull-Tab Schritte entstanden sind, die dieselbe Choice heraufziehen sollen. Damit würde zwar noch immer der Graph in der dritten Ansicht entstehen, in die ungewünschten Zweige würde dann aber nicht mehr abgestiegen werden.

3.5. Nichtdeterminismus durch freie Variablen

Eine zweite in Curry verfügbare Möglichkeit, Nichtdeterminismus einzuführen ist die Verwendung freier Variablen. Durch sie lässt sich mit Unbekannten arbeiten. Während der Berechnung werden passende Werte eingesetzt.

Das Fair Scheme erklärt kein Verfahren, wie mit freien Variablen umgegangen werden kann. Das ist aber kein Problem, denn der Nichtdeterminismus durch freie Variablen und der Nichtdeterminismus durch Choice-Konstrukte sind äquivalent [AH06]. Eine freie Variable wird so zu einer oder mehreren Choices umgeformt, sodass mit ihnen genauso gearbeitet werden kann wie mit jeden anderen nichtdeterministischen Ausdrücken.

Um in diese Umformung eine Struktur zu bringen und um die Verwendung von Polymorphie in freien Variablen zu erlauben, ist es sinnvoll, den Aufbau solcher Choices in Funktionen auszulagern. Diese Funktionen werden auch als Generatorfunktionen bezeichnet, da sie nichtdeterministisch alle möglichen Werte eines Typs generieren.

Es ist leicht zu erkennen, dass für jeden Typ, der im Quellprogramm deklariert ist, genau eine solche Generatorfunktion benötigt wird. Diese Funktion baut beliebig verschachtelte Choices auf, sodass jeder Konstruktor des Datentyps in einem nichtdeterministischen Zweig der Funktion zurückgegeben wird.

Die Argumente jedes Konstruktors werden dabei wiederum mit Generatoren besetzt. Ist der Typ eines solchen Arguments vollständig bekannt, so kann der passende Generator gewählt werden. Enthält der Typ eines Arguments dagegen eine Typvariable, über die der Datentyp parametrisiert ist, so kann auch die Generatorfunktion für das Argument nicht oder nicht vollständig bestimmt werden. In diesem Fall erwartet die Generatorfunktion für jeden Typparameter ein formales Argument. In diesem Argument wird ihr ein weiterer Generator übergeben, der zu dem Typ gehört, über den parametrisiert wird.

Beispielsweise erhält der Generator für **`Maybe`** einen Generator für **`Bool`** als einziges Argument, wenn eine freie Variable vom Typ **`Maybe Bool`** im Quellprogramm eingeführt wird. Für das Teilprogramm `coin :: Maybe Bool`
Die Generatoren für die Typen **`Bool`** und **`Maybe`** seien dabei als `gen_Bool` und `gen_Maybe` bezeichnet.

```
coin :: Maybe Bool
coin = gen_Maybe gen_Bool

gen_Bool :: Bool
gen_Bool = False ? True

gen_Maybe :: a -> Maybe a
gen_Maybe x = Nothing ? Just x
```

Der Generator für den Typ **`Bool`** ist dabei sehr einfach, da alle Konstruktoren dieses Typs Atome sind. Der **`Maybe`**-Generator zeigt, dass ein weiterer Generator von außen entgegengenommen und verwendet wird.

3.6. Generatoren, Literale und Unifikation

Für Datentypen mit praktisch unendlich vielen Konstruktoren ist es nicht sinnvoll möglich, eine Generatorfunktion nach diesem Schema zu erzeugen. Darunter fallen die Literale der Typen **`Int`**, **`Float`** und **`Char`**, die als algebraische Datentypen mit einer unendlich großen (bzw. endlich großen aber zu unhandlichen) Zahl von nullstelligen Konstruktoren angesehen werden können. Da es aber in Curry nur diese drei Literale gibt, kann für diese eine alternative Implementierung für Generatorfunktionen angegeben werden. So kann eine Ordnung auf den Werten des Typs festgelegt werden und durch Nutzung von Nachfolgerfunktionen können so alle Werte nichtdeterministisch erzeugt werden. Beispielsweise können die Zahlen von jeweils einschließlich 0 bis 1000 durch den folgenden Generator berechnet werden:

```
gen_0_1000 :: Int
gen_0_1000 = gen' 0
  where
    gen' x = if x == 1000 then x else x ? gen' (x+1)
```

Die dafür notwendige Ordnung kann beliebig gewählt werden. Die Binärdarstellung des jeweiligen Datentyps lässt beispielsweise leicht eine solche Ordnung ableiten.

Problematisch ist dabei aber, dass das Aufzählen des sehr großen Wertebereichs dieser Typen sehr lange dauert, obwohl oft nur ein einziger Zweig der Gesuchte ist. Der häufig genutzte `=:=`-Operator, der eigentlich Terme mit freien Variablen unifizieren soll, kann in diesem Fall nur alle unpassenden Zweige verwerfen, anstatt direkt den richtigen auszuwählen.

Es ist dafür hilfreich, Choices zu markieren, wenn sie Generatoren sind. Dann ist nämlich über sie bekannt, dass sie jeden Wert ihres Typs in einem nichtdeterministischen Zweig annehmen werden. `=:=` kann dann darauf verzichten, sich alle Möglichkeiten aufzählen zu lassen, und stattdessen den Generator mit einem konkreten Wert überschreiben. Dies ist, verglichen mit den anderen Änderungen auf dem Graphen, ein Sonderfall. Normalerweise verändern Funktionen nur den Knoten, der durch sie beschriftet ist. Die Semantik von `=:=` ist aber insofern besonders, als dass die ganze Berechnung ohnehin fehlschlägt, wenn linke und rechte Seite unterschiedlich sind. Daher verhindert das direkte Zuweisen eines Wertes an einen Generator nie das Finden einer möglichen Lösung.

Alle anderen Konstrukte können intuitiv unifiziert werden. In Konstruktorterme wird rekursiv abgestiegen, falls der Konstruktor beider Seiten übereinstimmt, Literale werden lediglich verglichen und Funktionsanwendungen müssen zunächst ausgewertet werden, bevor sie unifiziert werden können.

3.7. Generatoren und Call-Time Choice

Abschnitt 3.5 bietet zwar eine Möglichkeit, wie freie Variablen in Generatoren übersetzt werden können, diese entspricht aber noch nicht der von Curry spezifizierten Call-Time Choice Semantik. Während die Run-Time Choice Semantik nie einschränkt, welcher nichtdeterministische Zweig gewählt werden muss, und die Call-Time Choice Semantik verlangt,

dass im Quellprogramm notierter Nichtdeterminismus immer auf die gleiche Weise ausgewertet werden muss, schränkt der zuvor genannte Übersetzungsmechanismus die möglichen Auswahlen *noch* weiter ein. Während dieser Effekt im zuvor genannten Beispiel nicht auftritt, äußert er sich in dem folgenden ebenfalls sehr einfachen Fall.

```
data MyPair a = MyPair a a
coin :: MyPair Bool
coin = x where x free
```

Der eingeführte Typ `MyPair` funktioniert dabei wie ein 2-Tupel. Er wird hier anstelle eines Tupels verwendet, da Tupel durch syntaktischen Zucker unterstützt werden, auf dessen Einführung hier verzichtet werden soll. Es ist leicht zu erkennen, dass `coin` vier Werte annehmen kann: `MyPair False False`, `MyPair False True`, `MyPair True False` und `MyPair True True`. Gemäß des Schemas des vorherigen Abschnitts würde dieses Programm etwa in das Folgende übersetzt werden:

```
data MyPair a = MyPair a a

gen_Bool :: Bool
gen_Bool = False ? True

gen_MyPair :: a -> MyPair a
gen_MyPair x = MyPair x x

coin :: MyPair Bool
coin = gen_MyPair gen_Bool
```

So führt `gen_Bool` den Nichtdeterminismus für die `MyPair`-Argumente in der Definition von `coin` ein. Wird dieses Programm derart ausgewertet, dass eine Call-Time Choice Semantik für Choice-Terme verwendet wird, so darf das aus `gen_Bool` hervorgehende `False ? True` in einem Berechnungszweig nie unterschiedlich ausgewertet werden. Insbesondere tragen die beiden Argumente des `MyPair`-Konstruktors dieses Konstrukt. Damit sind nur `MyPair False False` und `MyPair True True` mögliche Lösungen.

Dies kann korrigiert werden, indem der Punkt, an dem der Nichtdeterminismus eingeführt wird, an eine andere Stelle verschoben wird. Durch Abstraktionen kann dieser Punkt gesteuert werden [Han13]. Generatorfunktionen erhalten dafür ein zusätzliches Argument, das zwar keine Funktion hat, aber die Auswertung der Generatorfunktion verzögert. Indem dieses Argument erst bei der tatsächlichen Verwendung des Generators an die Generatorfunktion gegeben wird, lässt sich der Punkt, an dem der Nichtdeterminismus eingeführt wird, direkt in die Generatorfunktion legen, die das Argument in Konstruktoren verwendet. Da keine Anforderungen an den Typ dieses Arguments gestellt sind, bietet es sich an, den Unit-Typ zu verwenden:

```
data MyPair a = MyPair a a
```

```
gen_Bool :: () -> Bool
gen_Bool () = False ? True

gen_MyPair :: (() -> a) -> () -> MyPair a
gen_MyPair x () = MyPair (x ()) (x ())

coin :: MyPair Bool
coin = gen_MyPair gen_Bool ()
```

Hier wird die Generatorfunktion erst in der `gen_MyPair`-Generatorfunktion vollständig appliziert und so Nichtdeterminismus eingeführt. Damit können beide Argumente des `MyPair`-Konstruktors unterschiedlich gewählt werden, sodass alle vier geforderten Lösungen berechnet werden.

Kapitel 4

Das Zwischenformat ICurry

Eine Übersetzung eines Programms von einer Sprache in eine andere erfolgt oft nicht direkt, sondern verwendet eine oder mehrere allgemeine Repräsentationen als Zwischenschritt. Dies bietet eine Ebene, auf der manche Optimierungen leichter durchgeführt werden können als auf der der Quell- oder Zielsprache. Zusätzlich ermöglicht es erst, modulare Compiler oder ganze Compilerframeworks, wie z.B. das LLVM-Projekt (früher *Low Level Virtual Machine*), in dessen Kern die LLVM-IR steht, zu entwickeln.

Obwohl in dieser Arbeit einzig Curry als Quellsprache von Interesse ist, steht doch die einfache Erweiterbarkeit um zusätzliche Zielsprachen im Vordergrund. Daher ist es auch hier sinnvoll, ein solches Zwischenformat zu benutzen.

Daher wird in dieser Arbeit das Format *ICurry* (für *Imperative Curry*), sowie das strukturell gleiche, aber etwas geschwätzigere, *Extended ICurry* definiert. Diese orientieren sich noch grob an den strukturellen Elementen aus Curry, haben aber bereits einen imperativen Charakter und sind so gestaltet, dass eine Übersetzung in eine imperative Sprache mit möglichst wenig Aufwand verbunden sein soll. Daher sind sie eher minimalistisch und nur auf die nötigsten Konstrukte beschränkt.

4.1. Grammatik und Beschreibung von ICurry

Das Format ICurry beschreibt nur eine abstrakte Syntax. Als Zwischensprache ist die Definition einer konkreten Syntax auch nicht unbedingt nötig, da sie nicht von Menschen gelesen oder geschrieben wird.

Die Anordnung von Curry-Elementen ist in der Struktur von ICurry noch zu erkennen. Wie auch in Curry ist der gesamte Inhalt der Programme in Module unterteilt. Curry-Datentypen werden auf ICurry-Datentypen abgebildet. Curry-Funktionen werden auf ICurry-Funktionen abgebildet. Diese ICurry-Funktionen sind nach außen hin ebenfalls referenziell transparent, verzichten aber auf bestimmte Schachtelungen, die in Curry möglich sind.

Die Übersetzung von Curry nach ICurry nutzt das *Curry-Frontend*, das Curry-Programme in das wesentlich einfachere, aber noch immer konzeptionell funktional-logische *FlatCurry* übersetzt. Daher sollten in der folgenden Grammatikbeschreibung eher Zusammenhänge zwischen ICurry- und entsprechenden FlatCurry-Konstrukten ersichtlich sein als solche zwischen ICurry- und AbstractCurry-Konstrukten.

```
type IQName = (String, String)     -- moduleName.localName
type IArity = Int                  -- arity
```

```
type IVarIndex = Int                -- a variable
type ITVarIndex = Int               -- a type variable
```

Zunächst verwendet ICurry einige Typaliase, deren Namen auf ihren Verwendungszweck schließen lassen.

```
data IProg = IProg String           -- module name
                   [String]         -- imports
                   [IDataType]      -- declared data types
                   [IFunction]      -- declared functions
```

Ein Modul, der Name `IProg` ist inspiriert von der entsprechenden FlatCurry-Struktur `Prog`, trägt also seinen Namen, eine Liste von importierten Modulen, eine Liste von definierten Datentypen und eine Liste von definierten Funktionen.

```
type IVisibility = Public           -- public
                 | Private          -- private
```

Die Sichtbarkeit gibt an, ob ein Datentyp, ein Konstruktor oder eine Funktion auch außerhalb des Moduls, in dem sie definiert ist, verwendet werden kann. Damit ist diese Information nur ein Mittel, das Zielprogramm etwas stärker zu strukturieren, da auch unsichtbare Funktionen sich nicht anders verhalten würden, wären sie sichtbar.

```
type IDataType = (IQName, IVisibility, [ITVarIndex], [IConstructor])
data IConstructor
  = IConstructor   IQName           -- the constructor's name
                   IVisibility      -- can be used from other modules?
                   [ITExpr]         -- one type expression for each argument
```

Ein Datentyp (`IDataType`) ist gegeben durch einen Typnamen, wie er in Signaturen vorkommen würde, eine Liste von Typvariablen, über die er parametrisiert sein kann und eine Liste von Konstruktoren. Ein Konstruktor (`IConstructor`) hat wiederum einen Namen sowie eine Liste von Argumenten. Für jedes dieser Argumente ist ein Typ gegeben. Dieser ist notwendig, um freie Variablen zu erlauben. Die Generatoren, in die freie Variablen übersetzt werden, benötigen zum Erzeugen eines Konstruktorterms natürlich auch Argumente. Um Generatoren für diese Argumente anzugeben, werden die jeweiligen Typen benötigt.

```
data ITExpr
  = ITVar          ITVarIndex     -- the type variable
  | ITFunc         ITExpr         -- domain type
                   ITExpr         -- range type
  | ITCons         IQName         -- the type's name that is applied
                   [ITExpr]       -- the arguments
```

Diese Typausdrücke (`ITExpr`) entsprechen genau denen in FlatCurry. Es können Typkonstruktoren (`ITCons`) appliziert werden und funktionale Typen aufgebaut werden (`ITFunc`). Außerdem kann auf die Typvariablen (`ITVar`), über die der Datentyp parametrisiert ist. Die

Typvariablen können zwar auch neu sein, also nicht in der Argumentliste des Typs eingeführt werden, dies ergibt aber nur in Kombination mit funktionalen Typen korrekte Programme. Daher ist dieser Fall eher uninteressant, da das Raten von Funktionen schwierig und oft nicht hilfreich ist und daher ohnehin viele Übersetzer keine Generatoren für Funktionstypen anbieten werden.

```
data IFunction
  = IFunction      IQName         -- the function's name
                   IVisibility    -- can be used from other modules?
                   IFuncBody      -- what the function does
```

Eine Funktion (`IFunction`) hat einen Namen und einen Körper. Über den Körper entscheidet sich, ob die Funktion in dem Modul definiert, oder nur die Curry-Bindung für eine extern definierte Funktion ist. ICurry verzichtet auf die Angabe von Typsignaturen. Diese würden ohnehin nicht verwendet, da alle Funktionen immer auf allgemeinen Knoten arbeiten. Eine speziellere Typung der erzeugten Funktionen ist so nicht möglich. Auch ist garantiert, dass das Programm korrekt getypt ist, da es sich andernfalls nicht zu FlatCurry und in der Folge zu ICurry übersetzen ließe. Daneben werden Typinformationen nur zur korrekten Auswahl eines Generators für freie Variablen benötigt. Dort sind sie, wie zuvor beschrieben, auch vorhanden, beziehen sich dann aber jeweils auf einen Generator, nie auf eine ganze Funktion.

```
data IFuncBody
  = IExternal      IArity         -- the function's arity
                   String         -- the function's external name
  | IFuncBody      [IVarIndex]    -- the function's arguments
                   IBlock         -- the function's actual behavior
```

Im Fall einer externen Funktion (`IExternal`) werden Informationen über diese Funktion benötigt. Das ist natürlich ein Name, unter der die Funktion gefunden werden kann. Dieser leitet sich aus dem qualifizierten Namen der Curry-Funktion ab. Weiterhin wird die Stelligkeit angegeben. Anders als bei in Curry definierten, übersetzten Funktionen, kann diese Information nicht aus der Argumentliste gewonnen werden. Hier äußert sich das Fehlen von Typsignaturen, die in FlatCurry die Stelligkeit beinhalten würden.

Für im Quellprogramm definierte Funktionen (`IFuncBody`) sind eine Variablenliste und ein Block nötig. Die Variablenliste enthält Bezeichner für die Kindknoten des Knotens, auf der die Funktion aufgerufen wird. Direkt nach Eintreten in eine Funktion werden Referenzen auf diese Knoten an jene Variablen gebunden.

```
type IAssign = (IVarIndex, IExpr)
data IBlock
  = ISimpleBlock   [IAssign]      -- assignments to local variables
                   IExpr          -- the return expression
  | ICaseConsBlock [IAssign]      -- assignments to local variables
                   IVarIndex      -- the variable to differentiate by
                   [IConsBranch]  -- the possible branches
```

```
    | ICaseLitBlock  [IAssign]      -- assignments to local variables
                     IVarIndex      -- the variable to differentiate by
                     [ILitBranch]   -- the possible branches
```

Ein Block (`IBlock`) entspricht etwa einem Block, wie er im Sprachgebrauch über imperative Sprachen vorkommt. Anders als in imperativen Sprachen allgemein ist jedoch zusätzlich die innere Struktur nicht frei, sondern es sind drei Arten von möglichen Blöcken mit jeweils unterschiedlicher Absicht zulässig. Allen diesen ist dabei gemein, dass sie lokale Variablen einführen können, bevor ihre eigentliche Arbeit beginnt. Dazu enthalten sie eine Liste von Zuweisungen. Eine Zuweisung (`IAssign`) enthält eine Variable, die sie einführt, und einen Ausdruck, der an diese Variable gebunden wird.

Der Unterschied zwischen den verschiedenen Blöcken ist, ob sie eine Bedingung abbilden. Ein `ISimpleBlock` beinhaltet keine Bedingung. Stattdessen trägt er einen Ausdruck, aus dem eine Graphstruktur aufgebaut wird, die in den Knoten geschrieben wird, der gerade ausgewertet wird.

`ICaseConsBlock` und `ICaseLitBlock` stellen jeweils eine Bedingung dar. Sie unterscheiden über einen Konstruktor oder ein Literal. Falls der entsprechende Knoten noch nicht ausgewertet ist, fordern sie die Auswertung dieses Knotens an, bevor sie darüber eine Entscheidung treffen können. Sie wählen den passenden Zweig aus der Liste von `IConsBranch` bzw. `ILitBranch` aus und führen den darin enthaltenen Block aus.

```
data IConsBranch
  = IConsBranch    IQName         -- the constructor to match this branch
                   [IVarIndex]    -- variable bindings
                   IBlock         -- what happens if this branch is taken
data ILitBranch
  = ILitBranch     ILiteral       -- the literal to match this branch
                   IBlock         -- what happens if this branch is taken
```

Der `ILitBranch` trägt dazu nur das Literal und den auszuführenden Block. Der `IConsBranch` trägt zusätzlich eine Variablenliste. Diese funktioniert wie die Variablenliste des Funktionskörpers. Aus ihr gehen Referenzen auf die Kinder des Konstruktorknotens, über den entschieden wurde, hervor.

```
data IExpr
  = IVar           IVarIndex      -- the variable
  | ILit           ILiteral       -- the literal's value
  | IFCall         IQName         -- the function's name
                   [IExpr]        -- the arguments
  | ICCall         IQName         -- the constructor's name
                   [IExpr]        -- the arguments
  | IOr            [IExpr]        -- the possibilities
```

Ausdrücke (`IExpr`) beschränken sich auf solche Konstrukte, die eine direkte Entsprechung bei den Knotenbeschriftungen haben oder auf bestehende Teilgraphen verweisen. Ein Ausdruck

kann so eine Variable (`IVar`), ein Literal (`ILit`), eine (möglicherweise partielle) Funktionsapplikation (`IFCall`), eine (möglicherweise partielle) Konstruktorapplikation (`ICCall`) oder eine Choice (`IOr`) sein. Durch eine Variable wird auf einen anderen Teilgraph verwiesen. So wird sofort Sharing eingeführt, wenn diese Variablen auch im Quellprogramm an mehreren Stellen verwendet werden. Die Applikationen führen einen Namen sowie eine Liste von Argumenten. Falls diese nicht der Stelligkeit der Funktion oder des Konstruktors entsprechen, so handelt es sich dabei um eine partielle Applikation. Anders als Curry oder FlatCurry, die nur Choices mit genau zwei Auswahlmöglichkeiten unterstützen, ist in ICurry eine beliebige Anzahl von Auswahlmöglichkeiten zulässig. Während dies in übersetzten Funktionen keinen großen Vorteil bringt, erlaubt es in externen Funktionen und Generatoren oft einfacheren Programmtext, da auf die Aufteilung und Verschachtelung in mehrere Choice-Teilausdrücke verzichtet werden kann.

```
data ILiteral
  = IInt          Int           -- the integer value
  | IChar         Char          -- the char value
  | IFloat        Float         -- the float value
```

Der `ILiteral`-Datentyp dient genau wie in FlatCurry dazu, alle drei Basistypen als Literal verwenden zu können.

4.2. Transformationen von FlatCurry zu ICurry

Obwohl eine starke Ähnlichkeit zwischen FlatCurry und ICurry zu erkennen ist, gibt es doch Unterschiede, die hauptsächlich dadurch begründet sind, dass ICurry bereits auf einige Strukturen verzichtet, deren Umsetzung im Ausführungsmodell des Fair Scheme nicht direkt möglich ist. Daher werden hier zwei Aspekte bei der Übersetzung von FlatCurry nach ICurry näher erläutert.

4.2.1. Let-Ausdrücke

`let` `...` **`in`** `...` kann in Curry überall dort verwendet werden, wo ein Ausdruck gefordert ist. FlatCurry bildet dies noch ab. ICurry dagegen verzichtet auf das beliebige Einführen neuer Variablen, da imperative Sprachen üblicherweise Variablen auf der Ebene von Anweisungen einführen und dies in Ausdrücken nicht zulassen. Ein Curry-Ausdruck `40 + (let x = 1 + 1 in x)` hat beispielsweise keine direkte Entsprechung in imperativen Sprachen. Zwar bieten einige Sprachen Lambda-Ausdrücke, in die solche Konstrukte übersetzt werden könnten, diese fehlen aber in anderen Sprachen. So würde die Zwischensprache manche Zielsprachen vor anderen bevorzugen, was in Konflikt mit dem Ziel, eine möglichst zielsprachenunabhängige Übersetzung zu erreichen, steht. Außerdem existiert durch die fehlende Garantie der referenziellen Transparenz in der Regel kein Sharing zwischen mehreren Aufrufen einer Funktion. Weil dann Choices, die zusammengehören sollen, ihren

Bezug untereinander verlieren, verliert eine solche Implementierung auch die durch Sharing realisierte Call-Time Choice Semantik. Sie wäre also nicht korrekt.

Stattdessen müssen Variablen, die durch ein solches **let** in einem Ausdruck entstehen, aus dem Ausdruck herausgehoben werden. In einer imperativen Sprache würde so in einer Anweisung die Variable x definiert werden. Der ursprüngliche Ausdruck würde in einer nachfolgenden Anweisung vorkommen. Darin würde die Variable nur verwendet. Eine Python-Funktion, die einen Wert, der aus diesem Ausdruck berechnet wird, zurückgibt, wäre beispielsweise die Folgende:

```
def f():
  x = 1 + 1
  return 40 + x
```

Ein weiterer Unterschied zwischen Curry und imperativen Sprachen mit Bezug auf **let**-Konstrukte ist, dass diese rekursiv sein können. Ausdrücke wie **let** xs = 42:xs **in** xs sind also zulässig. In strikten Sprachen sind solche Konstrukte nicht direkt möglich.

Wenn alle neu eingeführten Variablen bereits auf Anweisungsebene gehoben sind, ist die Lösung für dieses Problem eher einfach. Es werden erst alle Variablen eingeführt und erhalten einen leeren Knoten, der nur als Hülle für einen noch gänzlich unbekannten Wert funktionert. Erst wenn so alle Variablen bereits existieren, werden sie mit Inhalt gefüllt, indem die leeren Knoten eine Beschriftung und Kinder erhalten, die aus den jeweiligen Ausdrücken aufgebaut werden. Da die Variablen vorher bereits existieren, können sie verwendet werden, während eine Graphstruktur aus dem Ausdruck aufgebaut wird.

4.2.2. Case-Ausdrücke

Der Kontrollfluss wird in Curry durch mehrere Regeln einer Funktion oder durch **case**-Ausdrücke gesteuert. In FlatCurry werden diese beiden Konstrukte durch mehrere einfache **case**-Ausdrücke abgebildet, die jeweils nur über einen einzigen Konstruktor eine Entscheidung treffen. Diese können noch immer an jeder Stelle auftreten, an denen ein Ausdruck benötigt wird. Das Ausführungsmodell kann aber eine Fallunterscheidung nicht an beliebiger Stelle ausführen. Da die Auswertung bedarfsgesteuert passiert, ist es nötig, einen solchen Bedarf anzeigen zu können, sodass ein anderer Knoten ausgewertet wird, bevor die ursprüngliche Funktion weiter ausgeführt werden kann. Bedarf, einen anderen Knoten ausgewertet zu sehen, liegt immer genau dann vor, wenn anhand des enthaltenen Werts eine Entscheidung getroffen werden soll. Anders ausgedrückt verursacht jeder **case**-Ausdruck, dass ein anderer Wert in Kopfnormalform vorliegen muss. Daraus ergeben sich Einschränkungen für Fallunterscheidungen, die notwendig sind, um die bedarfsgesteuerte Auswertung immer zu ermöglichen.

So kann nicht erwartet werden, dass die Ausführung einer Funktion aus dem Inneren eines Ausdrucks unterbrochen werden kann, um den Bedarf der Kopfnormalform eines anderen Knotens zu signalisieren. Ähnlich wie bei den **let**-Konstrukten existieren mit *Exceptions* wieder Konstrukte in nur manchen Sprachen, die dies erlauben würden. Genau wie bei der

Lösung zur Umsetzung von **let** lässt sich auch dieses Problem lösen, indem **case**-Ausdrücke auf die Anweisungsebene gehoben werden. Damit ist es ihnen möglich, aus der Funktion herauszuspringen, falls zuerst die Kopfnormalform eines anderen Knotens benötigt wird. Daraus ergibt sich die Strukturierung in Blöcke, die in ICurry zum Einsatz kommt. Ein `ISimpleBlock` enthält keine Fallunterscheidung, sondern nur einen Ausdruck. Ein `ICaseConsBlock` oder `ICaseLitBlock` enthält eine Fallunterscheidung auf oberster Ebene. Dort ist es kein Problem, Anweisungen zur Umsetzung der Fallunterscheidung zu nutzen. Unterhalb eines solchen unterscheidenden Blocks werden weitere Blöcke angegeben. Dadurch wird noch nicht vorzeitig auf die Ausdrucksebene abgestiegen, sodass weitere Fallunterscheidungen ebenfalls noch Ausdrücke benutzen können.

Die zweite wesentliche Einschränkung für **case**-Konstrukte entsteht aus dem Wiedereinstieg in eine Funktion nach der Berechnung einer benötigten Kopfnormalform. Nur in manchen Fällen kann gewährleistet werden, dass die Funktion diese neu berechnete Kopfnormalform auch sieht und benutzt. In anderen Fällen würde diese Auswertung verloren gehen, sodass immer wieder die Berechnung der Kopfnormalform des gleichen Teilausdrucks angefragt wird. Dies tritt beispielsweise durch eine Konstruktion wie in der Funktion

```
f x = case not x of ...
```

auf. Wird diese Funktion ausgewertet, so baut sie den Graphen für den Ausdruck **not** `x` auf und fordert dessen Kopfnormalform an. Wenn diese berechnet ist, muss die gesamte Funktion von vorne beginnen, da es in kaum einer Sprache möglich ist, in der Mitte einer Funktion einzusteigen. Dann würde allerdings ein neuer Teilgraph für den Ausdruck **not** `x` aufgebaut, dessen Auswertung angefordert wird. Der bereits ausgewertete Teilgraph würde nicht weiter verwendet werden können.

Es ist leicht zu sehen, dass dieses Problem immer dann auftritt, wenn die Fallunterscheidung über einen Ausdruck erfolgt, der mehr ist als ein einfacher Verweis auf eine Variable. Daher ist eine Einschränkung von beliebigen Ausdrücken zu einem Verweis auf eine Variable sinnvoll. Bei näherer Betrachtung fällt zusätzlich aber auf, dass durch Nutzung von **let** diese Einschränkung umgangen werden kann. **let** `y =` **not** `x` **in case** `y` **of** `...` ist praktisch gleich zum vorherigen Beispiel, ist aber auch dann noch zulässig, wenn nur über Variablen unterschieden werden darf.

Daher ist es nötig, nicht über beliebige Variablen unterscheiden zu dürfen, sondern nur über solche, die Funktionsargumente sind. Da diese nicht innerhalb der Funktion zugewiesen werden, ist garantiert, dass jede Änderung in ihnen beim nächsten Aufruf der Funktion genutzt wird.

4.2.3. Funktionsweise

Ausgehend von der FlatCurry-Darstellung ist das Umformen einer Funktion in eine ICurry-Funktion mit den genannten strukturellen Änderungen nicht sehr kompliziert. Besonders ist bereits in einer FlatCurry-Funktion sichergestellt, dass kein Shadowing auftritt. Shadowing würde verhindern, dass das Vorziehen von lokalen Variablen aus dem Ausdruck in eigene

Anweisungen die Semantik der Funktion erhält. Anstatt dass eine verdeckende Variable nur in dem Teilausdruck verwendet würde, in dem sie definiert ist, wäre sie auch außerhalb dieses Sichtbarkeitsbereichs verfügbar und würde anstelle der verdeckten Variablen verwendet werden. Die Freiheit von Shadowing erlaubt eine beliebige Reihenfolge aller Anweisungen.

So kann beim Verarbeiten der rechten Regelseite einer Funktion in den Ausdruck abgestiegen werden. Während alle anderen Konstrukte dabei in ihre Entsprechungen übersetzt werden, sind es **`let`**- und **`case`**-Ausdrücke, die eine besondere Behandlung erhalten.

Für einen **`let`**-Ausdruck werden alle eingeführten Variablen aufgesammelt. Es wird mit dem **`in`**-Ausdruck weitergearbeitet. Weitere Variablen, die weitere **`let`**-Ausdrücke tiefer innerhalb eines Ausdrucks einführen, werden ebenso aufgesammelt, ohne aber die durch die verschachtelten **`let`**-Ausdrücke gegebene Hierarchie des ursprünglichen Ausdrucks zu erhalten. Alle durch **`let`** eingeführten Variablen in einem Ausdruck werden so in einer flachen Liste festgehalten. Mehr noch kann diese Liste als Darstellung einer Menge angesehen werden, denn wie zuvor erläutert ist die Reihenfolge der Einführung aller dieser Variablen unter Abwesenheit von Shadowing nicht von Bedeutung. Die Ausdrücke, die an diese Variablen gebunden werden sollen, werden ebenfalls auf diese Weise übersetzt. Dabei können natürlich weitere Bindungen durch **`let`** gefunden werden, die ebenfalls wieder aufgesammelt und so übersetzt werden. So werden alle Bindungen auf eine Ebene gezogen.

Ein **`case`**-Ausdruck wird wiederum unterschiedlich behandelt, je nachdem, ob er an der Wurzel einer rechten Regelseite steht oder an einer anderen Stelle. Die Wurzel der rechten Regelseite unterscheidet, welche Art von Block die Funktion auf oberster Ebene trägt. Findet sich dort eine Fallunterscheidung durch ein **`case`**, so wird ein `ICaseConsBlock` oder `ICaseLitBlock`, je nachdem, ob die Fallunterscheidung über einen Konstruktorterm oder über ein Literal erfolgt, den Block der Funktion stellen. Ist die Wurzel ein Ausdruck, der eine ICurry-Entsprechung hat, so wird ein `ISimpleBlock` erzeugt.

Findet sich ein **`case`**-Ausdruck tiefer im Ausdruck, so wird er in eine neue Funktion gehoben. Wie in Abschnitt 4.2.2 erläutert, wird ein Punkt benötigt, an dem nach der Berechnung einer Kopfnormalform mit der Funktion fortgesetzt wird. Indem der **`case`**-Ausdruck so an den Anfang einer neuen Funktion geschoben wird, stellt diese neue Funktion genau diesen Einstiegspunkt bereit. An die Stelle des **`case`**-Ausdrucks in der ursprünglichen Funktion tritt dann ein Funktionsaufruf von der neuen Funktion.

4.3. Typparameter für Generatoren

Da durch das Curry-Frontend bereits sichergestellt ist, dass nur typkorrekte Programme übersetzt werden, führt das ICurry-Format keine Typinformationen mit. Eine Ausnahme davon stellen Generatorfunktionen dar. Es muss natürlich bekannt sein, welche Konstruktoren eine freie Variable annehmen kann, damit genau diese erzeugt werden können. Diese Information kann aus dem Typ gewonnen werden.

FlatCurry beinhaltet die Signatur einer Funktion, aus der sich leicht der Typ des ganzen Ausdrucks, der die rechte Seite der Programmregel bildet, ablesen lässt. Im Allgemeinen

ist aber die freie Variable nur ein Teilausdruck unter dem gesamten Ausdruck. Da aus dem Typ des ganzen Ausdrucks nicht leicht Typen der Teilausdrücke berechnet werden können, nutzt die Übersetzung nicht reines FlatCurry, sondern die Erweiterung *Annotated FlatCurry*. Diese erlaubt es, jeden Teilausdruck mit zusätzlichen Informationen auszustatten. Eine der häufigsten Anwendungen dieser Möglichkeit ist, den Typ des Ausdrucks an dieser Stelle zu notieren. Das Curry-Frontend bietet es an, die Typinformationen, die während der Typprüfung inferiert werden, in diesem Format zu speichern, sodass sie bei der anschließenden Übersetzung nach ICurry genutzt werden können.

Typargumente für ICurry werden daraus in drei Schritten gewonnen. Als erstes wird aus dem Typ jeder freien Variable berechnet, welche Informationen noch benötigt werden, um ihren Generator vollständig zu bestimmen. Anschließend wird anhand des statischen Aufrufgraphen ermittelt, von welchen Stellen ein Typ durchgereicht werden muss. Schließlich wird diese Information genutzt, um zusätzliche Argumente in Funktionen und Funktionsaufrufe einzufügen. In diesen werden Generatoren übergeben.

Der erste Schritt erfolgt unabhängig für jede Funktion eines Moduls. Es wird jede freie Variable, die in dieser Funktion eingeführt wird, betrachtet. Ihr Typ wird rekursiv untersucht. Mögliche Typausdrücke an dieser Stelle sind eine Typabstraktion (*forall* a . τ), ein funktionaler Typ (τ_1 -> τ_2), ein Typkonstruktoraufruf (C τ_1 τ_2 ...) oder eine Typvariable (α). Aufrufe von Typkonstruktoren enthalten bereits die notwendige Information, welcher Generator verwendet werden muss, nämlich der des Typs, der appliziert wird. In ihre Argumente wird abgestiegen, um auch dort alle notwendigen Typinformationen zu sammeln. Freie Variablen eines funktionalen Typs sind zwar oft nicht sinnvoll, werden hier aber noch nicht verworfen, da sie hier noch leicht behandelt werden können, indem einfach in beide Seiten des Typs abgestiegen wird. Eine Typabstraktion wird sich nie im Typ einer freien Variable finden, da Curry zur Zeit dieser Arbeit keinen Polymorphismus höheren Ranges unterstützt. Damit ist die Nutzung einer Typvariable die hier gesuchte Information. Jedes Vorkommen einer Typvariable im Typ einer freien Variable zeigt so an, dass Typinformationen von außen in die Funktion gelangen müssen. So wird für die Funktion eine Menge von Typvariablen in einer separaten Datenstruktur gespeichert, die genau aus allen Vorkommen von Typvariablen in den Typen freier Variablen entsteht, aber keine Duplikate enthält.

Der zweite Schritt beschäftigt sich mit den Beziehungen mehrerer Funktionen untereinander. Er kann daher nicht jede Funktion einzeln betrachten. Stattdessen werden alle Funktionen eines Moduls gleichzeitig untersucht. Die Informationen über benötigte Typargumente aller verwendeter Funktionen anderer Module müssen dabei schon vorliegen, da sie beeinflussen können, ob eine Funktion ein Typargument benötigt oder nicht. Glücklicherweise sind zyklische Abhängigkeiten in Curry nicht zulässig [Han16], sodass sämtliche Typargumente über alle Funktionen in einem Modul in einem einzigen solchen Schritt berechenbar sind.

Dazu wird eine Fixpunktiteration durchgeführt. Darin werden wiederholt alle Funktionen einmal bearbeitet. Alle Funktionsaufrufe, die sich innerhalb einer Funktion finden, werden auf benötigte Typargumente untersucht. Diese ergeben sich dabei genau aus den `Comb`- (oder in Annotated FlatCurry `AComb`-) Ausdrücken mit Kombinationstyp `FuncCall` oder `FuncPartCall`.

Funktionen höherer Ordnung werden nicht bearbeitet, da sie bereits alle Typargumente beinhalten müssen. Das ist leicht nachzuvollziehen, da Curry keine Polymorphie höheren Ranges unterstützt, und so für funktionswertige Argumente immer ein konkreter Typ bestimmt werden kann. Damit wird bei dem Aufruf einer als Argument übergebenen Funktion erwartet, dass die Funktion bereits partiell auf alle notwendigen Typargumente appliziert ist, falls sie solche benötigt. Dies geschieht wiederum in der ursprünglichen Kombination durch `FuncPartCall`, die im Fall von Funktionen höherer Ordnung auch dann verwendet wird, wenn noch keine Argumente appliziert werden.

Für jeden solchen Funktionsaufruf wird die aufgerufene Funktion ermittelt und die Menge von benötigten Typargumenten geladen. Dann wird der Typausdruck aus der Signatur der Funktion mit dem Typausdruck der aufgerufenen Funktion unifiziert. So kann jeder Typvariable der Funktionssignatur ein konkreterer Typ zugeordnet werden. Möglicherweise ist dieser konkretere Typ aber noch immer nicht vollständig bekannt. Er kann beispielsweise ein Typkonstruktoraufruf auf eine oder mehrere Typvariablen sein. In beiden Fällen ist dabei dennoch erreicht worden, dass sich alle Typvariablen nun nicht mehr auf die Signatur der aufgerufenen Funktion beziehen, sondern auf die der aufrufenden Funktion, die gerade untersucht wird. Alle auftretenden Typvariablen in diesem Typausdruck werden der Menge an benötigten Typausdrücken hinzugefügt.

Als Beispiel soll eine Iteration in diesem Programm betrachtet werden:

```
g :: Maybe a
g = Just x where x free

f :: Maybe b
f = g
```

In einer vorherigen Iteration soll bereits berechnet worden sein, dass `g` vom Typ `Maybe a` ist und ein Typargument für die Typvariable `a` benötigt. In der nun betrachteten Iteration wird für `g` keine zusätzliche Information gewonnen. Für `f` wird nun der Funktionsaufruf von `g` gefunden. Der Aufruf der Funktion `g` in der Funktion `f` hat den Typ `Maybe b`. Dieser wird mit dem Typausdruck der `g`-Signatur `Maybe a` unifiziert. Das Ergebnis der Unifikation ist offensichtlich, dass der Typ, der in `g` durch die Typvariable `a` repräsentiert wird, in `f` durch `b` dargestellt ist. Weil nun die Funktion `g` ein Typargument für `a` benötigt, benötigt `f` Typargumente für alle Typvariablen im Typausdruck `b`. Also benötigt `f` ein Typargument für die Typvariable `b`.

Der Fixpunkt ist offensichtlich erreicht, sobald nach einer Iteration die gleiche Menge an benötigten Typargumenten entsteht, die vor der Iteration verwendet wurde.

Dieser Algorithmus stellt dies dar:

Input: An Annotated FlatCurry module *prog*
Result: Set of all required type arguments *rt*

```
1  rt ← loadRequiredTypeArgs(imports(prog));
2  unchanged ← false;
3  while not unchanged do
4      unchanged ← true;
5      for (fname, fbody) ← program do
6          for (fname′, type) ← functionCalls(fbody) do
7              fsig ← lookupSignature(fname);
8              σ ← unify(type, fsig);
9              texps ← {e|(a, e) ∈ σ, a ∈ rt[fname′]};
10             unchanged ← unchanged and typeVars(texps)\rt[fname] = ∅;
11             rt[fname] ← rt[fname] ∪ typeVars(texps);
```

In einem dritten Schritt werden die Funktionen und Funktionsaufrufe derart angepasst, dass alle benötigten Typargumente übergeben und angenommen werden. Dazu wird für jedes benötigte Typargument ein formales Argument in eine Funktion eingeführt. Diese neuen Typargumente werden, da sie immer sofort appliziert werden, an die linke Seite der Argumentliste gesetzt. Die Reihenfolge der Typargumente ist zwar beliebig, muss aber zwischen Funktionsdefinition und Funktionsaufruf gleich sein. Um dies sicherzustellen, bietet es sich an, die Typargumente anhand der Ordnung auf den Namen der Typvariablen zu sortieren. Im Rumpf einer Funktion wird dann jede Funktionsapplikation um die benötigten Typargumente ergänzt. Dazu wird wie im vorherigen Schritt durch Unifikation bestimmt, womit jede Typvariable belegt wird. Falls sie durch eine andere Typvariable gegeben ist, wird das entsprechende neu eingeführte Typargument an die aufgerufene Funktion übergeben. Falls dagegen ein konkreterer Typ bestimmt werden kann, so wird ein Aufruf der zu diesem Typen gehörenden Generatorfunktion eingefügt. Die Argumente dieser Funktion werden genauso bestimmt. Sie erhalten ebenso wieder Generatoraufrufe oder eingeführte Typargumente.

Werden Typargumente direkt an aufgerufene Funktionen übergeben, ergibt sich das in Abschnitt 3.7 beschriebene Problem, dass freie Variablen durch zu aggressives Sharing manche Werte nicht annehmen, obwohl sie das eigentlich sollten. Anstatt aber tatsächliche Abstraktionen einzuführen, die in die durch Curry vorgegebenen Konstrukte passen, wird eine einzige, zusätzliche externe Funktion durch ICurry eingeführt. `unshare :: a -> a` tut, was ihr Name vermuten lässt. Sie verhindert, dass ein Knoten zwischen mehreren Eltern geteilt wird. Sie kopiert dazu ihren Choice-beschrifteten Argumentknoten und gibt der Kopie eine andere Identität, sodass die Entscheidung dieser kopierten Choice nicht mehr an die Entscheidung über eine andere Choice gekoppelt ist. An allen Stellen, an denen eine Typvariable weitergegeben wird, wird diese in einen Aufruf von `unshare` verpackt. Damit werden die Zusammenhänge, die fälschlicherweise einige Werte verbieten, aufgebrochen, sodass sich die Call-Time Choice Semantik zeigen kann. Dies stellt einen Vorteil zu der

vorgestellten Implementierung durch Abstraktionen dar, weil ein Backend sich so nicht mit dem korrekten Umgang mit dem imaginären Unit-Argument auseinander setzen muss, sondern Generatorfunktionen ohne nähere Kenntnis dieses Mechanismus umsetzen kann.

4.4. Semantik für ICurry-Laufzeitsystem

Ein Zustand des Laufzeitsystems ist ein 4-Tupel (G, k, Q, R).

Dabei ist G der gerichtete, knotenbeschriftete Graph, der den Programmzustand abbildet. $\mathcal{N}$ bezeichne dabei die Knotenmenge. Ein Knoten kann beschriftet sein durch einen Konstruktor, ein Funktionssymbol, ein Literal oder den Choice-Operator (?). Die Kantenmenge als ganzes ist nicht von weiterem Interesse. Einzelne Kanten werden durch Nennen der Kindknoten angegeben.

$k \in \mathcal{K}$ bezeichnet die Kontrolle. Dabei ist $\mathcal{K} = \mathcal{N} \cup (\mathcal{N} \times \mathbb{N}) \cup \{\gg\}$.

In der Kontrolle kann also ein Knoten stehen. Dies bedeutet, dass dieser Knoten zur Kopfnormalform ausgewertet werden soll. Zu dem Knoten kann eine Zahl ergänzt sein. Diese gibt die Position an, über die ein Pull-Tab-Schritt stattfinden soll. Entsprechend wird sie auch nur dann betrachtet, wenn der Knoten mit einem Choice-Symbol beschriftet ist. Das spezielle Symbol $\gg$ zeigt an, dass das nächste Element der Warteschlange Q bearbeitet werden soll.

Q ist eine Liste von Tupeln (S, C) und dient als Warteschlange für weitere Auswertungen. Dabei ist S eine Liste über der Trägermenge $\mathcal{N} \cup (\mathcal{N} \times \mathbb{N})$ und ist als Aufrufstapel bei der Auswertung zu verstehen. C ist eine partielle Funktion, gegeben als eine Menge von Abbildungen $x \Rightarrow y$, wobei $x, y \in \mathbb{N}$. Sie enthält die Information über bei der Auswertung getroffene Auswahlen an Choice-beschrifteten Knoten.

Um Missverständnisse durch doppeldeutige Notation zwischen Stack, Warteschlange und Graphbeschriftung zu vermeiden, werden hier :: als Listenkonstruktor der Warteschlange und ::: als Listenkonstruktor eines Stacks verwendet.

R ist schließlich eine Teilmenge von $\mathcal{N}$, und ist die Menge aller bisher berechneten Ergebnisse.

Im Folgenden werden einige Konventionen verwendet. So bezeichnet f als Knotenbeschriftung immer ein Funktionssymbol. c als Knotenbeschriftung meint immer ein Konstruktorsymbol. Literale werden nicht explizit betrachtet sondern als nullstellige Konstruktoren angesehen. s wird als beliebiges Symbol als Beschriftung verstanden. n mit optionalem Index bezeichnet einen Knoten von G bzw. G', je nachdem, ob sich die Bezeichnung auf den Zustand vor bzw. nach einem Schritt bezieht. Sind Beschriftung oder Kinder eines Knotens von Interesse, so werden diese an den Knoten geschrieben. Findet sich also die Notation $n : s(n_0, \ldots, n_m)$ in der Kontrolle, im Stapel oder an einer anderen Stelle, so ist n in G mit s beschriftet und hat genau die Kinder n_0 bis n_m. Diese Notation kann (muss aber nicht) rekursiv fortgesetzt sein, um größere Ausschnitte des Graphen darzulegen.

Änderungen am Graphen werden dargestellt durch $G' = G[n : s'(n_0, \ldots, n_m)]$, wobei n bereits ein Knoten in G ist (wir fügen also nie zusammenhangslose Knoten in G ein) und n_0 bis n_m bereits in G sein können, dies aber nicht müssen. Handelt es sich aber um neue

Knoten, so sind diese mit Beschriftung und Kindern angegeben. Dies ändert die Beschriftung des Knotens n zu s', und die Kinder zu $n_0, \ldots, n_m$.

Choice-beschriftete Knoten enthalten als erstes Pseudo-Kind immer eine ID, um Choices einander zuordnen zu können. Diese sind keine Knoten im herkömmlichen Sinn, sondern lediglich eine natürliche Zahl oder $\perp$, falls die Choice noch keinen Identifikator erhalten hat.

Dazu gibt es noch die Prozedur *newid*, die – etwas unvollständig definiert – beginnend bei 0 mit jedem Aufruf die nächsthöhere natürliche Zahl liefert. Eine genauere Definition ließe sich leicht durch einen Zähler im Zustandstupel angeben. Aus Lesbarkeitsgründen wird darauf allerdings verzichtet.

4.4.1. Ausgangszustand, Endzustand

Die Abarbeitung des Programms beginnt üblicherweise damit, dass G den auszuwertenden Ausdruck (z.B. ein einzelner Knoten ohne Kinder mit der Beschriftung *main*) enthält, die Kontrolle mit n die Auswertung des Wurzelknotens des Ausdrucks fordert und sowohl Warteschlange als auch Ergebnismenge leer sind.

Die Auswertung terminiert, sobald die Kontrolle das Symbol $\gg$ enthält, sich aber keine weiteren Elemente mehr in der Warteschlange befinden. In der Ergebnismenge finden sich nun Verweise auf alle Knoten im Graphen, an denen sich die Normalformen des Ausdrucks, mit dem begonnen wurde, finden.

4.4.2. Regeln

Wir beginnen mit allgemeinen Regeln, die zum leichteren Umgang in der gewählten Notation hilfreich sind, sich aber noch nicht mit den Eigenheiten des Ausführungsmodells selbst beschäftigen.

1. *Fehlschlag* Wir verwerfen eine Berechnung, wenn die Auswertung eines bereits als Fehlschlag bekannten Knotens gefordert wird:

$$(G, n : <fail>, (S, C) :: Q, R) \rightarrow (G, \gg, Q, R)$$

2. *Zweigwechsel* Wir bearbeiten den nächsten Aufrufstapel der Warteschlange:

$$(G, \gg, (n : S, C) :: Q, R) \rightarrow (G, n, (S, C) :: Q, R)$$

3. *Positionsfreiheit* Wir verwerfen die Positionsinformation, wenn an dem auszuwertenden Knoten kein Choice-Operator steht, um keine Regeln doppelt angeben zu müssen:

$$(G, (n : s, i), Q, R) \rightarrow (G, n, Q, R)$$

   falls $s \neq ?$.

Nun folgen Regeln zum Umgang mit funktionsbeschrifteten Knoten. Die Funktionen, mit denen Knoten beschrieben sein können, erwarten als Argumente immer Kinder des Knotens,

den sie beschriften. Dabei können an einem Knoten mehr Argumente hängen als die Funktion erwartet, falls sie funktionswertig ist. Die Funktion kann entweder ein $demand(i)$ mit $i \in \mathbb{N}$ zurückgeben. Das zeigt an, dass das i-te Argument zur Kopfnormalform ausgewertet werden muss, bevor die Funktion einen Wert liefern kann. Oder die Funktion kann eine Beschriftung und eine Liste von Kindknoten zurückliefern. Das zeigt an, dass eine Auswertung zu einem Knoten mit einer solchen Beschriftung und Argumentliste ausgewertet werden kann. Eine genaue Definition, wie sich Funktionen für einzelne *ICurry*-Konstrukte verhalten sollen, folgt später.

In diesen und folgenden Regeln zeigt sich an der Verwendung der Warteschlange, dass hier eine Breitensuche beschrieben ist. Es wird immer der aktuell bearbeitete bzw. neue Stapel an das Ende der Warteschlange gelegt, während die zuvor genannten Regeln immer einen Stapel vom Anfang der Warteschlange nehmen. So kann sich die Ausführung nicht an einem Stapel aufhängen, während andere Stapel zielführender wären.

Es ist leicht zu sehen, dass die Breitensuche durch eine Tiefensuche ersetzt werden kann, indem Stapel nicht an das Ende, sondern wieder an den Anfang der Warteschlange gestellt werden.

4. *Abstieg* Falls die Funktion ein Argument fordert, müssen wir dieses zunächst auswerten:

$$(G, n : f(n_0, \ldots, n_m), (S, C) :: Q, R) \to (G, \gg, Q +\!\!+ [((n_i, i) ::: n ::: S, C)], R)$$

   falls $f(n_0, \ldots, n_m) = demand(i)$, die Funktion also die Kopfnormalform des i-ten Arguments fordert. An S baut sich so der Aufrufstapel auf.

5. *Auswertung* Falls die Funktion kein Argument fordert, so werten wir sie aus:

$$(G, n : f(n_0, \ldots, n_m), (S, C) :: Q, R) \to (G', \gg, Q +\!\!+ [S, C)], R)$$

   wobei $G' = G[n : s(n'_0, \ldots, n'_p, n_q, \ldots, n_m)]$, falls f q-stellig und $f(n_0, \ldots, n_{q-1}) = (s, [n'_0, \ldots, n'_p])$. Der Knoten erhält also eine neue Beschriftung mit neuen Kindern. Einige der ursprünglichen Kinder werden aber zusätzlich an den Knoten gehängt. Dies ist nötig, da eine Funktion nicht alle Argumente auswerten muss, sondern wiederum eine Funktion zurückgeben kann, die die nachfolgenden Argumente auswertet.

Nun betrachten wir Choices an einer Ausdruckswurzel. Hier spalten sie die Auswertung des Ausdrucks mit Choice in mehrere Auswertungen ohne diese Choice auf.

6. *Spaltung: keine ID* Wir lösen eine Choice an der Wurzel auf (Fall 1: Choice trägt noch keine ID). Die Choice bekommt dabei eine ID.

$$\begin{aligned}&(G, n :?(\bot, n_0, \ldots, n_m), ([], C) :: Q, R) \to \\ &(G', \gg, Q +\!\!+ [([n_0], C[x \Rightarrow 0]), \ldots, ([n_m], C[x \Rightarrow m])], R)\end{aligned}$$

   wobei $G' = G[n :?(x, n_0, \ldots, n_m)]$, $x = \text{newid}()$ und es gilt $(x \Rightarrow y) \notin C$ für alle $y \in \mathbb{N}$.

7. *Spaltung: uneingeschränkt* Wir lösen eine Choice an der Wurzel auf (Fall 2: Choice trägt

zwar eine ID, diese ist aber noch nicht eingeschränkt).

$$(G, n :?(x, n_0, \ldots, n_m), ([], C) :: Q, R) \rightarrow$$
$$(G, \gg, Q +\!\!+ [([n_0], C[x \Rightarrow 0]), \ldots, ([n_m], C[x \Rightarrow m])], R)$$

wobei $x \neq \bot$ und es gilt $(x \Rightarrow y) \notin C$ für alle $y \in \mathbb{N}$.

8. *Spaltung: eingeschränkt* Wir lösen eine Choice an der Wurzel auf (Fall 3: Choice trägt eine ID, die schon eingeschränkt ist).

$$(G, n :?(x, n_0, \ldots, n_m), ([], C) :: Q, R) \rightarrow (G, \gg, Q +\!\!+ [([n_y], C)], R)$$

wobei $x \neq \bot$ und es gilt $(x \Rightarrow y) \in C$ für ein $y \in \mathbb{N}$. Dieser Fall tritt immer dann ein, wenn eine Choice mit mehreren Eltern durch einen Pull-Tab Schritt bereits auf einer Seite des Sharings an die Wurzel gezogen und dort entschieden wurde, und dann die zweite, zugehörige Choice betrachtet wird, die nun gleich der anderen Choice entschieden werden muss.

Es folgen die Regeln für Choices, die nicht an einer Wurzel eines Ausdrucks stehen. Diese werden durch einen Pull-Tab Schritt näher an die Wurzel gezogen.

9. *Pull-Tab: keine ID* Wir führen einen Pull-Tab Schritt durch (Fall 1: Choice trägt noch keine ID). Die Choice bekommt dabei eine ID.

$$(G, (n :?(\bot, n_0, \ldots, n_m), i), ((n' : s(n'_0, \ldots, n'_q)) ::: S, C) :: Q, R) \rightarrow$$
$$(G', \gg, Q +\!\!+ [S, C])$$

wobei $G' = G[n' :?(x, n''_0 : s(n'_0, \ldots, n'_{i-1}, n_0, n'_{i+1}, \ldots, n'_q), \ldots, n''_m : s(n'_0, \ldots, n'_{i-1}, n_m, n'_{i+1}, \ldots, n'_q))][n :?(x, n_0, \ldots, n_m)]$, $x = \text{newid}()$ und es gilt $(x \Rightarrow y) \notin C$ für alle $y \in \mathbb{N}$.

10. *Pull-Tab: uneingeschränkt* Wir führen einen Pull-Tab Schritt durch (Fall 2: Choice trägt zwar eine ID, diese ist aber noch nicht eingeschränkt).

$$(G, (n :?(x, n_0, \ldots, n_m), i), ((n' : s(n'_0, \ldots, n'_q)) ::: S, C) :: Q, R) \rightarrow$$
$$(G', \gg, Q +\!\!+ [S, C])$$

wobei $G' = G[n' :?(x, n''_0 : s(n'_0, \ldots, n'_{i-1}, n_0, n'_{i+1}, \ldots, n'_q), \ldots, n''_m : s(n'_0, \ldots, n'_{i-1}, n_m, n'_{i+1}, \ldots, n'_q))]$, $x \neq \bot$ und es gilt $(x \Rightarrow y) \notin C$ für alle $y \in \mathbb{N}$.

11. *Pull-Tab: eingeschränkt* Wir wählen einen Zweig aus (Fall 3: Choice trägt eine ID, die schon eingeschränkt ist).

$$(G, (n :?(x, n_0, \ldots, n_m), i), ((n' : s(n'_0, \ldots, n'_q)) ::: S, C) :: Q, R) \rightarrow$$
$$(G', \gg, Q +\!\!+ [S, C])$$

wobei $G' = G[n' : s(n'_0, \ldots, n'_{i-1}, n_y, n'_{i+1}, \ldots, n'_q)]$, $x \neq \bot$ und es gilt $(x \Rightarrow y) \in C$ für ein $y \in \mathbb{N}$.

Schließlich kann eine Berechnung als abgeschlossen angesehen werden, sobald die Kopfnormalform erreicht ist. Dies ist für die Semantik des Laufzeitsystems ausreichend, da die eventuell gewünschte Auswertung bis zur Normalform durch eine externe Funktion erreicht werden kann.

12. *Lösung* Falls der gesamte Ausdruck in Kopfnormalform ist, so sind wir mit diesem fertig.

$$(G, n : c(\ldots), ([], C) :: Q, R) \rightarrow (G, \gg, Q, R \cup \{n\})$$

4.5. Semantik für übersetzte ICurry-Funktionen, in Umgangssprache

Eine ICurry-Funktion erhält beim Auswerten bzw. beim Versuch, die Funktion auszuwerten, als Argumente immer eine Teilmenge der Kindknoten, die unter dem auszuwertenden Knoten stehen. Sie hat dann zwei Möglichkeiten. Sie kann eine Beschriftung mit neuen Kindern zurückgeben. Dabei können diese Kinder bereits existierende Knoten oder neu erzeugte Knoten sein. Dies ist das Ergebnis der Funktionsauswertung. Alternativ kann sie durch $demand(i)$ mit $i \in \mathbb{N}$ eine Position zurückgeben. Dies bedeutet, dass zum Auswerten dieser Funktion erst das Argument an Position i zur Kopfnormalform ausgewertet werden muss.

Die Übersetzung von *ICurry*-Konstrukten soll dabei Folgendes umsetzen:

Ausdrücke (`IExpr`) bauen neue Teilgraphen auf. Gegebenenfalls werden dabei mittels Variablen existierende Knoten (wieder-)verwendet. Welche Knoten dabei jeweils gemeint sind, ist leicht den Namen der Konstruktoren entnehmbar.

Alle Blöcke können als erstes einige Variablen deklarieren und diesen dann einen Wert zuweisen. Das wird beispielsweise benötigt, um Sharing zu ermöglichen. Insbesondere müssen aber beim Abarbeiten eines Blocks zuerst alle Variablen einen leeren Knoten erhalten, damit sie bereits in allen Zuweisungen verwendet werden können. Dies ist nötig, da in diesen Ausdrücken auch Variablen verwendet werden können, die erst nach dem Ausdruck einen Wert erhalten. Das prominenteste Beispiel für diese Situation ist das Aufbauen einer unendlichen Datenstruktur, wie es beispielsweise der zuvor in Abbildung 3.2 dargestellte Ausdruck `let xs = 42:xs in xs` tut.

Ein einfacher Block (`ISimpleBlock`) führt immer zu einem Ergebnis einer Funktion. Er gibt den aus dem angegebenen Ausdruck aufgebauten Teilgraphen direkt zurück und fordert nie die Auswertung eines Arguments.

Ein unterscheidender Block (`ICaseLitBlock` und `ICaseConsBlock`) trifft eine Auswahl in Abhängigkeit von einer Variable. Diese Variable ist immer ein Argument der Funktion, damit die zugehörige Position zurückgegeben werden kann. Dies passiert immer dann, wenn ein solcher Block erreicht wird, die Variable aber einen Ausdruck erhält, der nicht in Kopfnormalform ist. Ist aber ein Ausdruck in Kopfnormalform enthalten, so wird eine Auswahl über den Konstruktor getroffen und mit dem Block im passenden Zweig fortgesetzt. Bei der Unterscheidung über einen Konstruktor werden zusätzlich alle Kindknoten an

Variablen gebunden, indem die Variablen je eine Referenz auf einen Kindknoten erhalten. Dies erfolgt noch bevor die lokalen Variablen des Blocks durch einen Ausdruck einen Wert erhalten, da die Bindungen in diesen Ausdrücken bereits verwendet werden können. Wird über ein Literal unterschieden, so werden offensichtlich keine solchen Variablen gebunden, da ein Literal bekanntlich keine Argumente hat.

Wenige extern definierte Funktionen mit spezieller Bedeutung für das Laufzeitsystem (wie z.B. `apply` für den Umgang mit partiell applizierten Funktionen oder `unshare`, was eine Pluralsemantik für die korrekte Verwendung von Generatoren umsetzt), können von diesem Schema abweichen. Andere (z.B. arithmetische Funktionen) bilden dieses Schema in handgeschriebenem Programmtext in Zielsprache nach.

4.6. Semantik für übersetzte ICurry-Funktionen, annähernd formell

Um die Semantik für ICurry-Funktionen anzugeben, sollen zunächst einige Hilfsfunktionen eingeführt werden.

selC (für *select children*) nimmt eine Liste von Variablennamen sowie einen Knoten und liefert eine Symboltabelle, die den Variablennamen sequenziell die Kindknoten des übergebenen Knotens zuordnet:

$$selC([v_0, \ldots, v_m], n) = \{(v_0, n_0), \ldots, (v_m, n_m)\}$$

Dabei sind $v_0, \ldots, v_m$ Variablennamen und n ein Knoten mit den Kindern $n_0, \ldots, n_p$, wobei $p \geqslant m$ gilt, da nicht immer alle Argumente von Interesse sind.

evalE (für *evaluate expression*) nimmt einen ICurry-Ausdruck und eine Symboltabelle und erzeugt daraus einen neuen Teilgraphen.

$$evalE(e, S) = \begin{cases} S[n], & \text{falls } e = \texttt{IVar } n \\ Node(l), & \text{falls } e = \texttt{ILit } l \\ Node(f', [evalE(e_0, S), \ldots, evalE(e_m, S)]), & \text{falls } e = \texttt{IFCall } f\ [e_0, \ldots, e_m] \\ Node(c, [evalE(e_0, S), \ldots, evalE(e_m, S)]), & \text{falls } e = \texttt{ICCall } c\ [e_0, \ldots, e_m] \end{cases}$$

Dabei sind $e, e_0, \ldots, e_m$ Ausdrücke, S eine Symboltabelle, f ein Bezeichner für eine ICurry-Funktion, c ein Konstruktorsymbol und f' die Funktion, die die Semantik der ICurry-Funktion f abbildet.

local bringt die lokalen Variablen eines Blocks in die Symboltabelle. Dazu bekommt es eine Menge von Abbildungen neuer Variablennamen auf je einen Ausdruck, sowie die bisherige Symboltabelle. Da Variablen verwendet werden können, bevor sie gebunden werden (nicht zuletzt um unendliche Datenstrukturen zuzulassen), erhalten sie zunächst einen leeren Knoten, bevor die Teilgraphen aus den zuzuweisenden Ausdrücken aufgebaut werden.

$$
\begin{aligned}
&local(\{(v_0, e_0), \ldots, (v_m, e_m)\}, S) = \\
&\quad local'(\{(v_0, e_0), \ldots, (v_m, e_m)\}, S \cup \{(v_0, n_0), \ldots, (v_m, n_m)\}) \\
&local'(\{(v_0, e_0), \ldots, (v_m, e_m)\}, S) = \\
&\quad \{(v_0, assign(n_0, evalE(e_0, S))), \ldots, (v_m, assign(n_m, evalE(e_m, S)))\}
\end{aligned}
$$

Dabei sind $v_0, \ldots, v_m$ Variablennamen, $e_0, \ldots, e_m$ ICurry-Ausdrücke und $n_0, \ldots, n_m$ sind neue, leere Knoten.

Die Funktion *assign* trägt dabei die Beschriftung und Kinder des zweiten übergebenen Knotens in den ersten Knoten ein und gibt diesen zurück. Dem aufmerksamen Beobachter wird dabei auffallen, dass sich keine formale Definition für die *assign*-Funktion angeben lässt. In der Tat würde man sie in der Programmierung als seiteneffektbehaftet bezeichnen. Da für eine wasserdichte formale Definition allerdings Kanten und Beschriftungen von Teilgraphen durch sämtliche Definitionen gereicht werden müssten, wird hier zugunsten der Lesbarkeit auf die vollständige Formalität verzichtet und stattdessen diese Abkürzung verwendet.

Die Funktion B_l bzw. B_c wählt den Block eines Branches über Literale bzw. Konstruktorterme, C_l bzw. C_c das Literal bzw. den Konstruktor. Anders als in der Semantikbeschreibung des Laufzeitsystems wird hier zwischen Literalen und Konstruktoren unterschieden, um nah an der *ICurry*-Struktur bleiben zu können.

$$
\begin{aligned}
B_l(\text{ILitBranch literal block}) &= \text{block} \\
B_c(\text{IConsBranch cons binds block}) &= \text{block} \\
C_l(\text{ILitBranch literal block}) &= \text{literal} \\
C_c(\text{IConsBranch cons binds block}) &= \text{cons}
\end{aligned}
$$

Die Funktion $localdefs$ wählt die Liste von Zuweisungen an lokale Variablen aus einem Block.

$$
localdefs(b) = \begin{cases} a, & \text{falls } b = \text{ISimpleBlock v a e} \\ a, & \text{falls } b = \text{ICaseLitBlock v a d bs} \\ a, & \text{falls } b = \text{ICasConsBlock v a d bs} \end{cases}
$$

Dabei ist b ein ICurry-Block.

Die Funktion $fromNode$ liest die Beschriftung s und die Kinder cs aus einem Knoten.

$$
fromNode(Node(s, cs)) = (s, cs)
$$

Die Funktion *bind* bringt die Knoten, die in einem Branch über einen Konstruktorterm gebunden werden sollen, in die Form einer Symboltabelle.

$$
bind(\text{IConsBranch cons binds block}, n) = selC(binds, n)
$$

Dabei ist n der Knoten, über den eine Fallunterscheidung durchgeführt wird.

Die Semantik einer ICurry-Funktion mit einer Liste von gebundenen Argumenten $vs_0, \ldots, vs_m$ und einem Block b kann nun gegeben werden durch eine Funktion f.

$$
f(x_0, \ldots, x_m) = f'_b(S \cup local(localdefs(b), S))
$$

wobei $S = \{(vs_0, x_0), \ldots, (vs_m, x_m)\}$.

f' ist dabei eine Schar von Hilfsfunktionen, parametrisiert über Blöcke.

$$f'_{\text{ISimpleBlock v a e}}(S) = fromNode(evalE(e, S))$$

$$f'_{\texttt{ICaseLitBlock v a d [}b_0,\ldots,b_m\texttt{]}}(S) = \begin{cases} demand(p), & \text{falls } vs[p] = d \text{ und } S[d] \text{ nicht in HNF} \\ f'_{B_l(b_0)}(S), & \text{falls } S[d] = C_l(b_0) \\ \quad\vdots & \\ f'_{B_l(b_m)}(S), & \text{falls } S[d] = C_l(b_m) \\ < fail > & \text{sonst} \end{cases}$$

$$f'_{\texttt{ICaseConsBlock v a d [}b_0,\ldots,b_m\texttt{]}}(S) = \begin{cases} demand(p), & \text{falls } vs[p] = d \text{ und } S[d] \text{ nicht in HNF} \\ f'_{B_c(b_0)}(S \cup bind(b_0, S)), & \text{falls } S[d] = C_c(b_0) \\ \quad\vdots & \\ f'_{B_c(b_m)}(S \cup bind(b_m, S)), & \text{falls } S[d] = C_c(b_m) \end{cases}$$

4.7. Semantik für ICurry-Typen

Ein *ICurry*-Typ wird in eine Funktion der Zielsprache übersetzt. Diese liefert einen Generator zurück. Ein Generator ist im einfachsten Fall nur eine Choice über alle Konstruktoren des Typs. Zur Unifikation durch die Funktion `=:=` kann es aber hilfreich sein, diese Choice als Generator zu markieren. Die Argumente dieser Funktion entstehen direkt aus den Typargumenten des Typs. Durch sie werden weitere Generatoren von außen in den Typ gegeben, um ihn zu konkretisieren. Wird ein solcher Generator von außen in einem Konstruktorterm verwendet, so ist er in einen Aufruf der Funktion `Prelude.unshare` zu verpacken, damit *Call-Time-Choice* erreicht wird und die Auswahlen nicht über voneinander unabhängige freie Variablen geteilt werden.

4.8. Extended ICurry

Das bisher vorgestellte ICurry-Format zeigt, verglichen mit FlatCurry, das als Ausgangsformat verwendet wird, bereits eine imperative Struktur. *Extended ICurry* ist ein weiteres Format, das zwar keine weiteren strukturellen Änderungen an einem ICurry-Programm durchführt, aber zwei Transformationen beinhaltet, die nur lokale Änderungen darstellen. Diese sollen lediglich den Umgang mit dem ICurry-Format vereinfachen und so Code, der sich in allen oder zumindest vielen Backends finden würde, vermeiden. Hier sollen diese zwei Unterschiede zwischen ICurry und Extended ICurry nun kurz erläutert werden.

ICurry übernimmt alle Bezeichner für Funktionen, Datentypen und Konstruktoren direkt aus der FlatCurry-Darstellung, sodass sie direkten Bezug zu den Bezeichnern des Quellprogramms haben. Damit sind sie jeweils durch einen qualifizierten Namen, der sich aus dem Namen des beinhaltenden Moduls und einem nur in diesem Modul eindeutigen lokalen Namen zusammensetzt, benannt. Während das für Funktionen in vielen Sprachen durchaus sinnvoll ist, da Funktionen auch dort wahrscheinlich Zeichenketten als Namen haben, ist es spätestens für Konstruktoren sinnvoll, numerische Identifikatoren anstelle von Zeichenketten zu verwenden. Diese lassen sich deutlich schneller vergleichen, was von Vorteil ist, weil Vergleiche bedingt durch die Funktionsweise von Curry die häufigsten Operationen auf Konstruktoren sind. Funktionen und Typen erhalten ebenfalls einen numerischen Identifikator, da durchaus Fälle denkbar sind, in denen diese hilfreich sind. Beispielsweise kann gewünscht sein, Funktionsaufrufe selbst zu verteilen, sodass eine Entscheidung über Funktionen getroffen werden muss [Kir17, S. 41].

Die erste Transformation zum Extended ICurry Format ergänzt also die bereits vorhandenen qualifizierten Namen um numerische Identifikatoren. Die numerischen Identifikatoren für Funktionen und für Datentypen sind innerhalb eines Moduls eindeutig. Eine programmweite Eindeutigkeit wäre nicht sinnvoll, da dies die Möglichkeit, Module unabhängig voneinander übersetzen zu können, stark beeinträchtigen würde. Konstruktoren erhalten numerische Identifikatoren, die innerhalb ihres Datentyps eindeutig sind. Hier ist eine weiterreichende Eindeutigkeit nicht nötig, da in einem typkorrekten Programm nie Konstruktoren eines Typs mit denen eines anderen Typs verglichen werden.

Entsprechend einfach kann die Zuordnung von numerischen Identifikatoren zu den Funktionen, Typen oder Konstruktoren eines Moduls erfolgen. So werden alle Funktionen in einem Modul, alle Typen in einem Modul und alle Konstruktoren in einem Datentyp aufsteigend durchnummeriert.

Die zweite Transformation beschäftigt sich mit der Eigenschaft einiger Parser, das Einlesen von Strukturen mit zu hoher Schachtelungstiefe zu verweigern. Dies ist beispielsweise der Fall, wenn Module, die Zeichenketten enthalten, nach Python übersetzt werden und mit der Referenzimplementierung *CPython* ausgeführt werden. Während auch andere tief verschachtelte Ausdrücke diese Situation ebenfalls verursachen können, erhöhen Zeichenketten durch ihre Darstellung als verkettete Liste mit jedem weiteren Zeichen ihre Schachtelungstiefe mindestens um eins. Je nach verwendeten Hilfsstrukturen kann die Schachtelungstiefe auch deutlich schneller wachsen.

Es wäre nun denkbar, weitere Konstrukte zur Darstellung von Listen oder gar Zeichenketten einzuführen, die verschachtelte Strukturen vermeiden, dies ist aber aus zwei Gründen nicht der gewählte Ansatz. Einerseits ist das Problem damit nur in diesen Spezialfällen gelöst und ein zu tief verschachtelter Ausdruck in Quellsprache wird noch immer so in die Zielsprache übersetzt, dass er nicht eingelesen werden kann. Andererseits würde ein zusätzliches syntaktisches Konstrukt die Komplexität der Backends erhöhen, was unerwünscht ist.

Daher begrenzt die Transformation von ICurry zu Extended ICurry die maximale Schachtelungstiefe, ohne ein neues Konstrukt einzuführen. Dazu wird beim Aufbauen der Extended

ICurry Struktur die Schachtelungstiefe mitgezählt und, sobald eine bestimmte Tiefe erreicht ist, der verbleibende Teilausdruck an eine Variable gebunden und diese Variable anstelle des Teilausdrucks verwendet. So können tief verschachtelte Ausdrücke unabhängig ihres Ursprungs vermieden werden.

4.9. Ninja als Buildsystem

Während der Übersetzung von Curry in eine Zielsprache durchläuft eine Quelldatei so mehrere Transformationsschritte und Zwischensprachen. Außerdem müssen für ein vollständiges Programm immer mehrere Module übersetzt werden. Daher ist es nötig, ein System zur automatischen Durchführung des Buildvorgangs zu benutzen.

Sowohl *PAKCS* als auch *KiCS2* steuern diesen Vorgang selbst. Das hat zwar den Vorteil, dass neue Abhängigkeiten sofort aufgenommen werden können, ist aber zu unflexibel, um eine leichte Integration mit anderen Sprachen zu erlauben. Stattdessen erlaubt die Übersetzung zu (Extended) ICurry ein leichtes Erzeugen von *Ninja*-Buildfiles. Ninja [1] versteht sich als Alternative zu *make*, verzichtet aber auf mächtige Konstrukte zugunsten einfacher maschineller Erzeugbarkeit und hoher Arbeitsgeschwindigkeit.

Für viele Sprachen stehen Bibliotheken bereit, mit denen sich leicht Ninja-Buildfiles erzeugen lassen. Daher soll durch den Einsatz von Ninja die Integration mit solchen Teilen eines Projektes, die nicht in Curry implementiert sind, möglichst einfach gehalten werden.

Andererseits genügt es bei der Nutzung von Ninja, den Abhängigkeitsgraphen zwischen allen Übersetzungseinheiten nur lokal zu betrachten. Es genügt, nur die direkten Abhängigkeiten eines Moduls im Ninja-Buildfile aufzuführen. Ninja kann daraus den vollständigen Graphen mit allen transitiven Abhängigkeiten und sich daraus ergebenden Reihenfolgen ablesen und anhand der Zeitstempel erkennen, welche Dateien (erneut) übersetzt werden müssen. So werden ohne weiteren Aufwand inkrementelle Übersetzungsvorgänge sowie paralleles Übersetzen auf mehreren Prozessorkernen ermöglicht.

Anstatt alle Stufen des Übersetzungsvorgangs so zu vereinen, dass sie zur richtigen Zeit die richtigen Module übersetzen, genügt es, jede Transformation einzeln aufrufbar zu machen. Da CPM nur eine ausführbare Datei als Übersetzungsziel angeben lässt, bietet es sich an, die einzelnen Übersetzungsphasen als Unterbefehle in eine ausführbare Datei zu verpacken. So kann `icurry f2i` von FlatCurry nach ICurry und `icurry i2e` von ICurry nach Extended ICurry übersetzen.

Für ein Projekt, das nur aus einer Curry-Quelldatei `Testmodule` besteht, die nur von der *Prelude* abhängt, können diese Ninja-Deklarationen erzeugt werden:

```
rule tfcy
  command = kics2-frontend --typed-flat -i $
  "$$(realpath "$$(dirname "$$(which kics2-frontend)")")/../lib" $in
rule icy
```

[1] `https://ninja-build.org/`, zuletzt abgerufen: 2018-10-11

```
  command = icurry f2i -I "./curry" $in $out
rule eicy
  command = icurry i2e -I "./curry" $in $out

build /opt/kics2-2.0.0/lib/.curry/Prelude.tfcy: $
  tfcy /opt/kics2-2.0.0/lib/Prelude.curry
build /home/marc/testproject/curry/src/.curry/Testmodule.tfcy: $
  tfcy /home/marc/testproject/curry/src/Testmodule.curry $
  | /opt/kics2-2.0.0/lib/.curry/Prelude.tfcy

build ./curry/Prelude.icy | ./curry/Prelude.ictdeps: $
  icy /opt/kics2-2.0.0/lib/.curry/Prelude.tfcy
build ./curry/Testmodule.icy | ./curry/Testmodule.ictdeps: $
  icy /home/marc/testproject/curry/src/.curry/Testmodule.tfcy $
  | ./curry/Prelude.ictdeps

build ./curry/Prelude.eicy: eicy ./curry/Prelude.icy
build ./curry/Testmodule.eicy: eicy ./curry/Testmodule.icy $
  | ./curry/Prelude.eicy
```

Die ersten Deklarationen sind Regeln. Sie geben an, wie eine Datei aus anderen erzeugt werden kann. Die zuvor bereits genannten Unterbefehle von `icurry` werden für die Übersetzung von Flat- zu ICurry und von I- zu Extended ICurry genutzt. Außerdem wird direkt auf das Curry-Frontend aus der *KiCS2*-Installation zurückgegriffen. Ein Dollarsymbol am Ende einer Zeile dient als Fluchtzeichen, um eine Zeile umbrechen zu können.

Anschließend folgt die Beschreibung des Abhängigkeitsgraphen durch Angabe aller Kanten, entlang derer eine Datei aus anderen erzeugt werden kann. Für eine Kante werden explizit und implizit erzeugte Dateien, gefolgt von der zu nutzenden Regel sowie expliziten und impliziten Abhängigkeiten angegeben. Der Unterschied zwischen impliziten und nicht-impliziten Dateien ist, dass implizite Ein- und Ausgaben nicht an den Befehl der Regel übergeben werden. Sie werden nur zur Berechnung der Abhängigkeiten verwendet und der Befehl kann ihre Namen selbstständig herleiten. So kann die Übersetzung nach ICurry die Typabhängigkeits-Dateien (`*.ictdeps`) durch einfaches Austauschen der Dateierweiterung finden. Genauso findet das Curry-Frontend Abhängigkeiten selbstständig, indem es sie im `CURRYPATH` sucht, anstatt sie als Befehlszeilenargument entgegenzunehmen.

Die ausführbare Datei `icurry` enthält selbst keine Möglichkeit, ein Ninja-Buildfile zu erstellen. Eine Übersetzung nur nach Extended ICurry wäre auch wenig hilfreich. Stattdessen bietet das ICurry-Paket Routinen zum Ermitteln aller Abhängigkeiten eines Moduls und einiger Hilfsfunktionen, mit denen für ein Modul und alle seine Abhängigkeiten Ninja-Buildanweisungen erzeugt werden können. So kann ein Backend mit sehr wenig Aufwand ein Meta-Buildsystem anbieten, das ein Ninja-Buildfile zur vollständigen Übersetzung von Curry bis zur Zielsprache erzeugen kann.

Kapitel 5

Das Python-Backend

Im Zuge dieser Arbeit wird mit *curry2python* ein Backend entwickelt, das ein ICurry-Programm nach Python übersetzt. Python ist eine moderne Sprache mit mächtigen Konstrukten, was eine Übersetzung in diese Sprache erleichtert, da so oft natürliche Entsprechungen für alle ICurry-Strukturen gefunden werden können. Da Python bereits eine automatische Speicherverwaltung bereitstellt, ist keine eigene Implementierung einer solchen nötig. Als dynamische Sprache erlaubt Python, Typen für manche Strukturen nicht festzulegen, sodass diese während der Entwicklung noch sehr leicht modifiziert werden können, ohne zu jeder Zeit ein starres Typsystem zufriedenstellen zu müssen. Damit eignet sich Python gut als Zielsprache für ein erstes Backend, da sich parallel zur Entwicklung des Backends auch das Zwischenformat und die Transformation zum Zwischenformat noch entwickeln.

Ein Backend besteht aus zwei Teilen. Ein Teil erledigt die Übersetzung von ICurry in die Zielsprache und transformiert so das Programm. Der zweite Teil stellt das Laufzeitsystem bereit. Dieses implementiert das in Kapitel 3 vorgestellte *Fair Scheme* und stellt Implementierungen für alle als `external` definierten Funktionen der Standardbibliothek bereit.

5.1. Graphdarstellung

Das Python-Backend stellt den Programmzustandsgraphen mithilfe einer Klasse `Node` dar. Diese stellt lediglich eine generische Hülle für beliebige Inhalte dar. Ihr Zweck ist es, die Identität eines Knotens bereitzustellen und es somit zu erlauben, einen Knoten zu mutieren, ohne seine Identität zu verändern. Darüber hinaus bietet sie einige Funktionen, um den Inhalt eines anderen Knotens zu übernehmen oder um zu prüfen, von welchem Typ ihr Inhalt ist.

Sämtliche Informationen über einen Knoten sind in Objekten spezieller Klassen enthalten, die jeweils einem `IExpr`-Konstruktor entsprechen. Eine Ausnahme bildet dabei der `IVar`-Konstruktor, der bereits während des Aufbauens der Graphstruktur aufgelöst wird, indem der verwiesene Teilgraph eingesetzt wird. So enthält die Klasse `FCall` einen Verweis auf eine Python-Funktion, sowie eine Liste von Argumenten, in der sich wiederum Knoten befinden. `CCall` beinhaltet den numerischen Identifikator eines Konstruktors und wiederum eine Argumentliste. Ein `Choice`-Inhalt enthält neben der Argumentliste, in der die nichtdeterministisch wählbaren Berechnungszweige enthalten sind, eine Ganzzahl, die die Identität dieser `Choice` darstellt, die in Abschnitt 3.4 erläutert ist, sowie ein Flag, das anzeigt, ob diese Choice einen gesamten Generator darstellt. Diese Information wird in der Implementierung der strukturellen Gleichheit (`=:=`) genutzt, um einen Generator während der Unifikation als

eine freie Variable zu erkennen und ihm so einen Wert zuzuweisen. Literale werden ohne zusätzliche Klasse direkt in den Inhalt eines Knotens geschrieben. Ein Fehlschlag wird durch das Fehlen eines Inhalts im Knoten repräsentiert.

Neben den Inhalten, die sich direkt aus jeweils einer ICurry-Struktur ergeben, nutzt das Python-Backend eine weitere Inhaltsklasse, die `ExtRes`. Diese ist ähnlich zur `Node`-Klasse, da sie nur ein Behältnis für einen unbestimmten Inhalt, eine *ext*erne *Res*source darstellt. Sie wird benutzt, um Informationen, die nicht direkt durch Curry-Funktionen bearbeitet werden sollen und entsprechend nicht in eine Graphstruktur gezwängt werden müssen oder können, im Graph festhalten zu können. Damit dient diese Klasse als Markierung für das Laufzeitsystem, dass an einer Stelle solche Daten vorliegen, die es nicht weiter betrachten soll. Dies wird beispielsweise benötigt, um aus dem Zustandsgraph auf *Handles*, die Betriebssystemressourcen wie Dateien oder Netzwerkverbindungen repräsentieren, verweisen zu können. Diese stellen auch aus Sicht von Python eine transparente Ressource dar und lassen sich damit nicht in einen Teilgraph umformen.

5.2. Übersetzung von ICurry nach Python

Die eigentliche Übersetzung eines ICurry-Moduls nach Python erfolgt sehr direkt. Da das ICurry-Format bewusst so entworfen ist, dass strukturelle Unterschiede zwischen ICurry und der Zielsprache vermieden werden, verzichtet das Python-Backend auf eine abstrakte Darstellung der Python-Syntax. Die Übersetzung besteht so aus nur wenig mehr als einem Pretty-Printer, der die imperative Struktur des ICurry-Formats mit der Syntax von Python ausdrückt.

Jedes ICurry-Modul wird in ein Python-Modul übersetzt. Dabei erhält das Python-Modul das Präfix `curryprog.`, um zu verhindern, dass Namenskonflikte entstehen. Eine ICurry-Funktion wird direkt in eine Python-Funktion übersetzt. Da Curry einige Sonderzeichen in Funktionen, besonders aber in Operatoren, zulässt, die in Python nicht Teil eines Bezeichners sein können, werden diese durch Digraphen ersetzt, wobei der Unterstrich als Fluchtzeichen funktioniert. So wird beispielsweise der Unterstrich selbst als `_u` kodiert oder das Pluszeichen als `_p`. Alle Funktionen erhalten außerdem das Präfix `C_` um ein versehentliches Überschreiben einer Funktion aus der Python-Standardbibliothek zu verhindern. Die in ICurry enthaltene Sichtbarkeit einer Funktion wird nicht verwendet, da Python kein Konzept bietet, die Verwendung aus verschiedenen Modulen zu beschränken.

Eine Funktion wird so immer durch

```
def <escaped_local_function_name>(node):
  x1 = node.content.args[0]
  x2 = node.content.args[1]
  ...
```

eingeleitet. Die Funktion nimmt also als einziges Argument nur den Knoten entgegen, auf den sie angewandt wird. Sie entpackt als erstes alle Argumente, die sie benötigt, aus dem Knoten. Anschließend folgt sofort der Block, der die Funktion beschreibt.

5.2.1. Übersetzung unterscheidender Blöcke

Die Übersetzung eines unterscheidenden Blocks erfolgt dabei sehr direkt. Beispielsweise sei ein einfacher unterscheidender Block in vereinfachter Darstellung gegeben:

```
ICaseConsBlock ["x2"]                     -- new variable x2
               [("x2", IFCall "fail" [])] -- x2 = fail
               "x1"                       -- case x1 of
               [("Nothing", [], block1),  --   Nothing -> block1
                ("Just", ["x3"], block2)] --   Just x3 -> block2
```

Dieser wird in ungefähr einen solchen Python-Abschnitt übersetzt:

```
if not x1.isHNF():
  return 0
x2 = Node()
x2.setContent(FCall(curryprog.Prelude.C_fail, []))
if x1.content.cons == "Prelude.Nothing":
  # translation of block1
elif x1.content.cons == "Prelude.Just":
  x3 = x1.content.args[0]
  # translation of block2
```

Der tatsächlich erzeugte Zielcode enthält allerdings numerische Identifikatoren anstelle der Namen der **Maybe**-Konstruktoren, die hier nur zum besseren Verständnis dargestellt sind.

Unterscheidende Blöcke beginnen immer mit der Prüfung, ob die Variable, über die unterschieden wird, bereits in Kopfnormalform ist. Zwar ist es den ICurry-Blöcken nicht strukturell anzusehen, aber da nur über solche Variablen unterschieden werden darf, die ein Argument der Funktion sind, darf diese Prüfung bereits vor dem Erstellen und Zuweisen der lokalen Variablen geschehen. Der Wert, der im Fall einer nicht-Kopfnormalform zurückgegeben wird, ist immer die Position der Variable, über die unterschieden wird, in der Argumentliste der Funktion. Dies ist zwar nicht aus dem Block ersichtlich, aber hier ist die Variable `x1` das erste Argument der Funktion. Falls die Funktion ein Ergebnis liefert, also nicht verlangen muss, dass zuerst eines ihrer Argumente ausgewertet werden muss, so zeigt sie dies durch einen Ergebniswert von `None` an.

Anschließend werden lokale Variablen eingeführt und mit je einem Wert belegt. Dies erfolgt in jedem Block auf genau diese Weise.

Danach beginnt die Fallunterscheidung über den Konstruktor des betrachteten Knotens. Dies geschieht durch eine Kette von `if-elif`-Vergleichen, da Python kein `switch-case`-Konstrukt, wie es aus anderen Sprachen bekannt ist, bietet. Im Fall von **`Just`** wird das Argument des Konstruktors an eine Variable gebunden, indem einfach die Variable auf den Knoten des Arguments verweist. Eine Unterscheidung über Literale (`ICaseLitBlock`) erzeugt nie Programmtext zum Entpacken von Argumenten, da Literale nie Argumente tragen. Stattdessen enthalten Fallunterscheidungen über Literale zusätzlich einen `else`-Zweig, der benötigt wird, da die Fallunterscheidung nicht jeden Wert der Basistypen abdecken kann:

```
    else:
      node.setContent(None)
      return None
```

Indem die Beschriftung des Knotens, der zuvor mit dem Funktionsaufruf beschriftet war, durch Setzen auf `None` entfernt wird, wird dieser Knoten als Fehlschlag angesehen. Da die Funktion etwas erreicht hat, sei es auch nur die Feststellung, dass kein Ergebnis berechnet werden kann, gibt sie entsprechend `None` zurück, um dies anzuzeigen und um keine Auswertung eines Arguments zu fordern.

5.2.2. Übersetzung unbedingter Blöcke

Die Übersetzung unbedingter Blöcke funktioniert im Allgemeinen ebenfalls offensichtlich. Ein `ISimpleBlock` führt genau wie die bedingten Blöcke zunächst lokale Variablen ein und belegt sie. Danach weist sie allerdings sofort dem Knoten, der mit dieser Funktion beschriftet ist, einen neuen Inhalt zu, bevor angezeigt wird, dass ein Ergebnis produziert wurde.

Ein `ISimpleBlock [] [] (IFCall "Prelude.+$"[IVar 0, IVar 0])` (wieder in reduzierter Darstellung), der beispielsweise direkt aus dem Rumpf einer `double`-Funktion, die nur auf **`Int`** operiert, übersetzt worden sein kann, würde damit in diesen Python-Quelltextabschnitt übersetzt:

```
    node.setContent(FCall(curryprog.Prelude.C__p_r, [x0, x0]))
    return None
```

Dabei ist gut zu erkennen, dass die `+$`-Funktion, die die Ganzzahladdition darstellt und vollständig aus Sonderzeichen besteht, einen gänzlich neuen Namen erhalten hat, der sich aus der Anpassung jedes unzulässigen Sonderzeichens ergibt.

`setContent` nimmt lediglich einen Inhalt entgegen, keinen ganzen Knoten. Dies ist in den meisten Fällen sinnvoll, da dadurch vermieden wird, einen Knoten nur zu erzeugen, um seinen Inhalt in einen anderen Knoten kopieren zu können. Es kann leicht vermieden werden, die Inhalte in einen Konstruktoraufruf von `Node(...)` zu verpacken. Allerdings stellt sich im Fall von `IVar`-Ausdrücken das Problem, dass diese bereits in einem Knoten enthalten sind. Daher wird dieser Spezialfall erkannt und unter Nutzung der `copyContentsFromNode` der Inhalt der Variable, nicht der Knoten selbst, in den anderen Knoten geschrieben. Ein einfaches Beispiel für diese Konstruktion ist die **`id`**-Funktion, die direkt das einzige Argument der Funktion in den Knoten schreibt und in diese Python-Funktion übersetzt wird:

```
  def C_id(node):
    x1 = node.content.args[0]
    node.copyContentsFromNode(x1, 1)
    return None
```

5.2.3. Rettung unverarbeiteter Argumente

In `C_id` ist erstmalig ein zweites Argument einer Funktion, die den Inhalt eines Knotens setzt, sichtbar. Sowohl `setContent` als auch `copyContentsFromNode` erlauben es, über diesen Parameter anzugeben, wie viele Argumente die Funktion tatsächlich konsumiert hat. Die Notwendigkeit dieses Arguments entsteht durch die Möglichkeit, funktionswertige Funktionen anzugeben. Beispielsweise ist **`id`** `double` funktionswertig. Der Curry-Ausdruck **`id`** `double 21` baut aber eine Struktur `Node(FCall(`**`id`**`,[Node(FCall(double),[]),Node(21)]))` auf. Wird nun der äußerste Knoten ausgewertet, so wird die **`id`**-Funktion aufgerufen. Diese verarbeitet nur das erste Argument, also die partielle Applikation der `double`-Funktion. Würde sie nun den Inhalt ihres Knotens mit dieser partiellen Applikation überschreiben, so wäre das Argument `21` verloren und einer der nächsten Auswertungsschritte würde einen Laufzeitfehler verursachen, da das Argument nicht gefunden werden kann. Dadurch, dass **`id`** der `copyContentsFromNode`-Methode mitteilt, dass sie nur ein Argument tatsächlich verarbeitet, nämlich die partiell applizierte `double`-Funktion, können weitere, unbenutzte Argumente aus dem Funktionsaufruf in das Funktionsergebnis übernommen werden.

5.2.4. Übersetzung von Datentypen

Aus einem Datentyp, der in einem ICurry-Programm angegeben ist, wird lediglich eine Generatorfunktion aufgebaut. Diese enthält keine Besonderheiten, die über die allgemeine Beschreibung der Abschnitte 3.6 und 3.7 hinausgehen. Beispielsweise wird für den Typ **`Prelude.Maybe`** der folgende Generator erzeugt:

```
def C_Maybe(node):
  x0 = node.content.args[0]
  node.setContent(Choice([ Node(CCall(0, [  ]))
                         , Node(CCall(1, [
                                Node(
                                  FCall(
                                    C_unshare, [x0])) ])) ])
                  .setIsGenerator(True))
  return None
```

Genau wie für andere Funktionen wird als erstes das Argument entpackt, das einen Generator für den Typ enthält, durch den **`Maybe`** parametrisiert wird. Dieses wird natürlich in einen Aufruf von `unshare` verpackt. Alle Konstruktoren werden in eine Choice gestellt, damit sie nichtdeterministisch ausgewählt werden können. Diese Choice wird als Generator markiert, damit für die `=:=`-Funktion bekannt ist, dass eine Unifikation erlaubt ist. Anders als in vorherigen Beispielen wird hier darauf verzichtet, die numerischen Identifikatoren durch besser lesbare Zeichenketten zu ersetzen. Es ist allerdings durch die Argumente leicht zu sehen, dass der Konstruktor **`Nothing`** durch den numerischen Identifikator `0` und **`Just`** entsprechend durch `1` dargestellt wird.

Im Vergleich mit zuvor vorgestellten Funktionsausschnitten ist weiterhin auffällig, dass das unqualifizierte `C_unshare` anstatt des qualifizierten `curryprog.Prelude.C_unshare` aufgerufen wird. Dies entsteht aus dem Umstand, dass sich der **`Maybe`**-Typ in der `Prelude` befindet und die Generatorfunktion so im gleichen Modul wie die aufgerufene Funktion definiert ist. In diesem Fall wird auf Funktionen nur durch ihren lokalen Namen verwiesen. Dies ist der Tatsache geschuldet, dass Python keine qualifizierten Namen für lokale Definitionen kennt. Daher werden zwar für alle Aufrufe von Funktionen anderer Module voll qualifizierte Namen verwendet, weil somit leicht Eindeutigkeit sichergestellt werden kann. Für Aufrufe von Funktionen im selben Modul werden allerdings nur die lokalen Namen verwendet. Als kleiner positiver Nebeneffekt ergibt sich, dass der Zielcode lesbarer wird.

5.3. Organisation des Laufzeitsystems

Das Laufzeitsystem besteht aus drei Python-Modulen. Eines dieser Module enthält die in Abschnitt 5.1 vorgestellten Klassen zur Graphdarstellung. Diese müssen in allen Teilen, die Kontakt mit übersetzten Curry-Modulen haben, zur Verfügung stehen. Somit ist es sinnvoll, diese in einem eigenen Modul zu halten. Ein weiteres Modul enthält eine Reihe von Hilfsfunktionen und -dekoratoren, die die Implementierung von externen Funktionen erleichtern sollen. Einige dieser Konstrukte werden in Abschnitt 5.5 näher betrachtet. Das dritte Modul enthält die eigentliche Implementierung des Fair Scheme.

Diese Implementierung hält sich sehr nah an die formale Beschreibung des Fair Scheme [AJ13]. Sie besteht aus zwei Klassen. Eine Klasse, `Progstate` enthält den Programmzustand. Der Programmzustand selber ist, wie eingangs bereits dargestellt, ein großer Graph. Um diesen Graph im Sinne eines Curry-Programms transformieren zu können, ist allerdings noch eine zusätzliche Information nötig. So muss bekannt sein, wo der auszuwertende Ausdruck beginnt. Dieser Knoten, der genau den Beginn des Ausdrucks markiert, muss also ebenfalls Teil des Programmzustands sein. Nun ziehen aber Pull-Tab Schritte (siehe Abschnitt 3.3) Choices zu dieser Wurzel hin. Damit werden sie schließlich in mehrere Wurzeln gespalten, die so alle je einen Ausdruck beschreiben, der eine Lösung des ursprünglichen Ausdrucks ist. Der Programmzustand muss also eine Liste von Wurzeln statt nur einer einzelnen enthalten. Außerdem entstehen beim Durchführen eines Pull-Tab Schritts mehrere Choices gleicher Identität. Werden diese an einer Wurzel in mehrere unabhängige Lösungen aufgespalten, so wird für jede dieser Lösungen die Choice entschieden. Diese Entscheidung muss zu jeder Wurzel gespeichert werden, damit später ausgewertete Choices gleicher Identität auf die gleiche Weise entschieden werden können.

Diese drei Informationen reichen aus, um ein korrektes Laufzeitsystem zu implementieren. Tatsächlich nutzte eine erste Version des Laufzeitsystems nur diese. Allerdings muss, wenn sich der Programmzustand auf einen Graphen, eine Liste von Wurzeln und zu jeder Wurzel die getroffenen Entscheidungen beschränkt, für jeden Schritt erst derjenige Knoten ermittelt werden, an dem eine Transformation möglich ist. Da dazu von der Wurzel bis zu diesem Knoten gelaufen werden muss, wird dadurch viel Zeit verschwendet. Stattdessen

erhält jede Wurzel zusätzlich einen Aufrufstapel, der diese Information erfasst und es so erlaubt, auf das Herablaufen des Graphen verzichten zu können. Damit finden sich in dieser realen Implementierung genau die Strukturen, die in der formalen Semantikdefinition des Laufzeitsystems aus Abschnitt 4.4 verwendet werden.

Daneben sind in der Klasse `CurryRuntime` alle Funktionen gesammelt, die jeweils einen Teil der Definition des Fair Scheme abbilden. Diese entsprechen in weiten Teilen den Beschreibungen dieser Funktionen des dieser Arbeit in Anhang B zu findenden *Implementation Guide*. Sie bieten allerdings einige Optimierungen, die sich bemühen, mehrere Schritte hintereinander auszuführen, ohne in einen anderen, aus Nichtdeterminismus entstandenen Berechnungszweig zu wechseln. Außerdem beinhaltet die Python-Implementierung des Laufzeitsystems mit der Methode `normalize` noch eine Auswertung zur Normalform. Im *Implementation Guide* fehlt eine Beschreibung dieser Funktion. Stattdessen weist er darauf hin, dass die `$!!` Funktion, die ohnehin extern implementiert werden muss, anstelle einer expliziten Auswertung zur Normalform verwendet werden kann. Dies hat den Vorteil, dass der Aufrufstapel natürlich genutzt wird, ohne ihn in einer zusätzlichen Implementierung einer Auswertung zur Normalform explizit betrachten zu müssen.

5.4. Anbindung externer Funktionen

Die Standardbibliothek der Sprache Curry bietet einige Funktionen, die sich nicht vollständig durch Curry-Konstrukte implementieren lassen. Darunter fallen natürlich Funktionen zum Umgang mit Betriebssystemressourcen wie dem Dateisystem oder der Netzwerkkommunikation. Aber auch einfache Funktionen, die mit den Basisdatentypen **`Int`**, **`Float`** und **`Char`** arbeiten, tun dies nicht durch eine große Fallunterscheidung über den ganzen Wertebereich des jeweiligen Typs. Stattdessen stützen sie sich auf eine handgeschriebene Implementierung in Zielsprache.

Der Übersetzer des Python-Backends erzeugt für jede externe Funktion, die er in einem ICurry-Modul findet, eine Python-Funktion, die nur die externe Implementierung aufruft. Beispielsweise wird für die **`getChar`**-Funktion der `Prelude` diese Funktion erzeugt:

```
def C_getChar(node):
  return currynative.Prelude.Prelude_dgetChar(node)
```

Es wäre auch möglich, der externen Implementierung direkt ihren in Curry-Programmen verwendeten Namen zu geben, und so einen zusätzlichen Funktionsaufruf zu vermeiden, indem für die Funktion dieser Programmtext erzeugt würde:

```
C_getChar = currynative.Prelude.Prelude_dgetChar
```

Dies erschwert allerdings die Entwicklung, da bereits das Laden eines Moduls erfordert, dass schon alle externen Implementierungen dieses Moduls definiert sind. Die Abstraktion erlaubt es, nur solche externen Funktionen definieren zu müssen, die vom auszuführenden Curry-Programm auch verwendet werden, da der Name der externen Implementierung nur dann aufgelöst wird, wenn die Funktion aufgerufen wird.

Der Name, unter dem die externe Implementierung gefunden werden soll, setzt sich dabei aus dem Modulpräfix `currynative.`, dem Modulnamen und dem externen Namen der Funktion zusammen. Das Modulpräfix ist dabei bewusst verschieden von dem Modulpräfix der übersetzten Module (`curryprog.`) gewählt. So können übersetzte Module und solche, die handgeschriebene externe Implementierungen beinhalten, an unterschiedlichen Pfaden gespeichert sein. Dies ist notwendig, um ein Projekt, das neben Curry-Programmteilen auch externe Implementierungen enthält, sinnvoll mit einem Versionsverwaltungssystem zu verwenden.

Über den Modulnamen wird die Modulhierarchie in die externen Implementierungen gespiegelt. Damit ist vorgegeben, dass sich für jedes Modul, das extern definierte Funktionen enthält, genau ein Modul unter `currynative.` findet, das die zugehörigen Implementierungen enthält.

Der externe Name der Funktion entsteht aus dem externen Namen einer FlatCurry-Funktion. In der Praxis findet sich dort der qualifizierte Name der Funktion. Dieser wird wie alle anderen Namen behandelt, Sonderzeichen werden durch entsprechende Zeichenfolgen ersetzt. Im obigen Beispiel ist der externe Name der **`getChar`**-Funktion `Prelude.getChar`, sodass nur der Punkt zu `_d` (von engl. *dot*) ersetzt wird.

Eine externe Funktion hat genau die gleichen Zugriffsmöglichkeiten wie eine übersetzte Funktion. Allerdings steht ihr zusätzlich der gesamte Sprachumfang von Python zur Verfügung. Beispielsweise nutzt die externe Implementierung der `+$`-Funktion der `Prelude`, die die Ganzzahladdition darstellt, die Python-Addition `+`:

```
def Prelude_d_p_r(node):
  args = node.content.args
  if not args[0].isHNF():
    return 0
  if not args[1].isHNF():
    return 1
  node.setContent(args[0].content+args[1].content)
```

Obwohl diese Implementierung einer übersetzten Funktion ähnlich sieht, unterscheidet sie sich strukturell von einer solchen Funktion. Sie entpackt nicht ihre Argumente in Variablen. Auch prüft sie gleich zwei Argumente darauf, ob diese in Kopfnormalform sind, bevor die Funktion überhaupt etwas tut. Schließlich verzichtet sie auf das **`return`** `None`, da ohnehin der implizite Rückgabewert der Funktion `None` ist und so die Geschwätzigkeit der handgeschriebenen Funktion reduziert wird.

5.5. Hilfsfunktionen für Interoperabilität

Viele der externen Funktionen sind sich strukturell sehr ähnlich. Die meisten zweistelligen Funktionen auf den Basistypen entsprechen beispielsweise dem Muster der zuvor vorgestellten `+$`-Funktion. Sie prüfen, ob alle Argumente des Knotens ausgewertet sind, wenden

eine Python-Funktion auf diese Argumente an und weisen das Ergebnis dem Knoten zu. Ein noch extremeres Beispiel sind primitive **IO**-Aktionen. Diese müssen zusätzlich zu ihren gewünschten Argumenten auch mit dem *Welt*-Objekt umgehen, das die korrekte Ausführungsreihenfolge sicherstellt.

Durch Dekoratoren bietet Python die Möglichkeit, zu verändern, welche Argumente in eine Funktion gelangen und was mit dem Rückgabewert geschieht. Ein Modul sammelt einige solcher Dekoratoren. Diese sind nur dazu gedacht, die Implementierung externer Funktionen zu vereinfachen. Sie bauen dazu gemeinsame Strukturen um diese externen Funktionen herum.

Die beschriebenen Funktionen, die die Auswertung aller Argumente fordern und dem Knoten einen Rückgabewert zuweisen, können durch diesen Dekorator vereinfacht werden:

```
def SimpleFuncAllStrictContent(func):
  numArgs = len(signature(func).parameters)-1
  @wraps(func)
  def wrapper(node):
    args = node.content.args[0:numArgs]
    for i in range(0,numArgs):
      if not args[i].isHNF():
        return i
    node.setContent(func(node,*(a.content for a in args)),numArgs)
  return wrapper
```

Dieser nimmt die Funktion entgegen, die die eigentlich auszuführende Funktionalität beinhaltet, und liefert stattdessen die umschließende Funktion `wrapper`. Diese erhält durch `@wraps(func)` die Signatur der umschlossenen Funktion. Das ist notwendig, damit mehrere Dekoratoren verwendet werden können, da diese oft Informationen über die Signatur der ursprünglichen Funktion benötigen. Der `wrapper` kümmert sich nun um die Anforderung der Kopfnormalformen und ruft die eigentliche Funktion `func` erst dann auf, wenn alle Kopfnormalformen vorliegen. Anhand der Verwendung der Funktion ist bereits erkennbar, dass auch der direkte Umgang mit Knoten, zumindest auf der obersten Ebene, entfällt. Stattdessen können die Argumente der Funktion auch in der Python-Implementierung als Argumente entgegengenommen werden. Der Rückgabewert muss nicht per `setContent` in den Knoten geschrieben werden sondern kann als Rückgabewert der Python-Implementierung durch den Dekorator in den Knoten gelangen.

Die externe Implementierung der `+$`-Funktion verkürzt sich so auf nur drei wesentlich aussagekräftigere Zeilen:

```
@SimpleFuncAllStrictContent
def Prelude_d_p_r(node, a, b):
  return a + b
```

Neben Dekoratoren, die so verschiedene Kombinationen von Striktheit bei **IO**- und nicht-**IO**-Funktionen abbilden, enthält das Modul einige Hilfsfunktionen zum Umgang mit Listen

und Strings als Spezialfall von Listen. Diese werden im Programmzustand natürlich auch als Graphstruktur dargestellt und müssen daher zur Benutzung in Python in native Listen oder Strings umgewandelt werden. Dazu bietet das Modul Funktionen, die diese Umwandlung strikt und nicht-strikt vornehmen können.

Kapitel 6

Das C-Backend

Das Ziel dieser Arbeit ist nicht, am Ende einen Übersetzer von Curry in eine andere Sprache zu erhalten. Stattdessen steht die Allgemeinheit des Übersetzungsprozesses im Vordergrund. Daher ist es sinnvoll, ein zweites Backend zu entwickeln. So kann einerseits erkannt werden, falls Zwischenformat oder Übersetzungsprozess sich auf Besonderheiten der Zielsprache des ersten Backends stützen. Das gibt zusätzliche Sicherheit, dass das Zwischenformat tatsächlich dazu geeignet ist, um ein Backend zur Übersetzung in jede beliebige imperative Sprache zu entwickeln. Andererseits kann so gleichzeitig die Dokumentation, die während der Entwicklung des ersten Backends entstanden ist, durch eine direkte Anwendung der enthaltenen Informationen ausprobiert und so auf Korrektheit überprüft werden.

Verglichen mit Python ist C eine sehr alte Sprache. C bietet keine der modernen Sprachfeatures, die Python anbietet, wie beispielsweise Closures, Mechanismen zum Trennen von Namensräumen oder eine automatische Speicherverwaltung. Dafür ist C deutlich näher an Maschinensprache und moderne C-Compiler erlauben eine starke Optimierung des übersetzten Programms. Anders als durch die Übersetzung nach Python ist es durch eine Übersetzung nach C so möglich, Curry bis zu nativen Programmen einer Maschine zu übersetzen und so eine wesentlich höhere Ausführungsgeschwindigkeit zu erreichen.

Strukturell ist das C-Backend *curry2c* sehr ähnlich zu dem in Kapitel 5 erläuterten *curry2python*. Es beinhaltet ebenso einen Übersetzer, der allerdings ein ICurry-Modul nicht nur in eine C-Quelldatei übersetzt, sondern auch eine Headerdatei erzeugt, die zur Einbindung in andere Module benötigt wird.

6.1. Graphdarstellung

curry2c stellt den Programmzustandsgraphen durch Instanzen einer Struktur `Node` dar. Anders als das *curry2python* kann ein solcher Knoten aber nicht nur als Hülle für einen beliebigen Inhalt verstanden werden, sondern die Struktur des Inhalts muss bereits festgelegt sein.

```
struct Node {
  char nodeType;
  char numChildren;
  Children children;
  union {
    int intLiteral;
    char charLiteral;
```

```
      float floatLiteral;
      int cons;
      int (*func)(Node*);
      choice_tag_t tag;
    } spec;
};
```

Daher enthält der Knoten zunächst die Information, von welchem Typ der Inhalt ist (`nodeType`). Zu fast jedem der möglichen Typen wird eine Zusatzinformation gespeichert. Die drei den jeweiligen Basistypen entsprechenden Literal-Knotentypen fassen natürlich den entsprechenden Wert. Ein Konstruktoraufruf enthält die Information, welcher Konstruktor verwendet wird. Ein Funktionsaufruf enthält einen Funktionszeiger auf die C-Funktion, in die die aufgerufene Curry-Funktion übersetzt wurde. Eine Choice enthält eine Marke, über die Choices als zusammenhängend identifiziert werden können. Daneben gibt es noch einen Knotentyp für Generatoren, der als Erweiterung einfacher Choices ebenfalls diese Marke benutzt. Schließlich gibt es noch den Fehlschlag, der keine zusätzliche Information trägt.

Da die meisten Typen mit Kindknoten umgehen, werden diese ebenfalls bereits auf oberster Ebene in einem Knoten eingeführt. Diese werden durch eine Kindanzahl (`numChildren`) und ein Union `Children` dargestellt. Das `Children`-Union enthält dabei die Kindknoten:

```
typedef union {
  Node *direct[NUM_NODE_CHILDREN];
  Node **indirect;
} Children;
```

Es erlaubt, eine Anzahl von Kindern direkt im Knoten unterzubringen. Da aber die Größe des Knotens begrenzt ist, muss auch diese Anzahl begrenzt sein. Zwar erlaubt es C, ein Array beliebiger Größe am Ende einer Struktur zu halten [C99, §6.7.2.1.17], dies ist allerdings nicht hilfreich, da Knoten geändert werden. So müsste der Knoten auch wachsen können. Das ist aber nicht möglich, weil im Allgemeinen nicht sichergestellt ist, dass der Knoten danach an der gleichen Adresse im Speicher liegt. Damit würde er seine Identität ändern, sodass Verweise auf diesen Knoten die alte Version oder freigegebenen Speicher sehen. Stattdessen können die Kindverweise auch über ein zusätzliches, dynamisch angelegtes Feld gespeichert werden, falls der Knoten mehr Kinder haben soll, als er selbst fassen kann. Dieses Feld darf zur Laufzeit verschoben werden, da nur dieser eine Knoten darauf verweist und dieser natürlich um die neue Adresse des Feldes aktualisiert werden kann.

Jeder Umgang mit Knoten erfolgt nur durch eine Reihe von Prozeduren. So werden die übersetzten und handgeschriebenen Funktionen einfacher und es kann zentral sichergestellt werden, dass Kindknoten je nach Kindanzahl immer an der richtigen Position zu finden sind.

6.2. Speicherverwaltung

Python bietet eine automatische Speicherverwaltung an. Daher kann das Python-Backend einfach neue Knoten erzeugen und muss sich keine weiteren Gedanken um die Entfernung nicht mehr benötigter Knoten machen. C selbst bietet keinen Mechanismus zum automatischen Freigeben nicht mehr benötigter Speicherbereiche. Durch das Übersetzungsschema werden bei der Ausführung fast jeder Funktion ein oder mehrere Knoten erzeugt. Um Speicherlecks zu vermeiden, ist es nötig, diese Knoten auch wieder freizugeben. Die Schwierigkeit dabei liegt darin, zu erkennen, welche Knoten noch benötigt werden und welche Knoten nicht mehr von einer Wurzel des Graphen erreichbar sind. Das C-Backend bietet zwei Methoden zur Speicherverwaltung.

Die erste Methode ignoriert das Problem der notwendigen Speicherfreigabe vollständig. Sie alloziert Knoten mittels `malloc` und gibt sie nie wieder frei. Damit ist diese Methode praktisch nicht relevant, da sie innerhalb sehr kurzer Zeit den gesamten zur Verfügung stehenden Speicher aufbraucht. Dennoch eignet sie sich gut während der Entwicklung des Laufzeitsystems und externer Funktionen, da so zusätzliche Komplexität vermieden wird und Fehler sich schneller finden lassen. Da die Testprogramme, die während der Entwicklung verwendet werden, klein sind und nur kurz laufen, stört das Speicherleck in diesem Fall nicht.

Als zweite, auch in der Praxis geeignete Methode kann auf den *Boehm-Demers-Weiser Conservative Garbage Collector* [BW88] zurückgegriffen werden. Dieser ist eine allgemeine Bibliothek zur automatischen Speicherverwaltung in C. Er erreicht dies, indem er die Registerinhalte und den Stack daraufhin untersucht, welche Speicherbereiche noch erreicht werden können. Er untersucht die so gefundenen Speicherbereiche, ob sie Adressen zu weiteren Speicherbereichen enthalten, und kann so alle diejenigen Bereiche finden, mit denen das Programm noch arbeiten kann. Speicherbereiche, die nicht auf diesem Weg erreichbar sind, sind auch vom Programm nicht mehr verwendbar, da sie nicht gefunden werden können. Diese kann er freigeben. Da dem Garbage Collector die Struktur der Daten nicht bekannt ist, muss er *konservativ* vorgehen, also jedem Wert, der – als Zeiger interpretiert – auf einen anderen, vom Garbage Collector verwalteten Speicherbereich verweist, folgen. So soll gewährleistet werden, dass nie erreichbarer und somit potenziell noch benötigter Speicher freigegeben wird. Andererseits können dadurch auch einzelne Speicherbereiche, die nicht mehr benötigt werden, fälschlicherweise erhalten werden, falls ihre Adresse zufällig einem Wert, der nicht als Zeiger verwendet wird, entspricht. Für das Laufzeitsystem bedeutet das, dass es von Vorteil ist, Zeiger auf Kindknoten explizit zurückzusetzen, anstatt nur die Anzahl der Kinder zu ändern, da der allgemeine Garbage Collector nicht wissen kann, dass diese Verweise nicht mehr verwendet werden.

Während der Ausführung eines Curry-Programms werden sehr viele Knoten erzeugt, die alle die gleiche Größe und Struktur haben. Daneben werden vom Laufzeitsystem nur wenige andersartige Speicherbereiche alloziert. Es ist daher denkbar, dass eine speziellere Speicherverwaltung, die die Struktur eines Knotens kennt und tiefer mit dieser umgehen kann, einen zusätzlichen Laufzeitgewinn gegenüber dem allgemeineren Boehm GC bieten

kann. Da die Laufzeitoptimierung eines Backends aber nicht Ziel dieser Arbeit ist, wird auf die Implementierung einer eigenen Speicherverwaltung verzichtet.

6.3. Umgang mit Einschränkungen durch die Sprache C

Als sehr alte Sprache lässt C einige Komfortfunktionalitäten, die moderne Sprachen bieten, vermissen. Diese sind jedoch nicht wirklich nötig, um eine Zielsprache zu unterstützen. Stattdessen kann um sie herum gearbeitet werden, um trotzdem ein korrektes Backend für die Sprache C anzugeben.

Der erste Unterschied zu einer modernen Programmiersprache ist die Organisation der Kompilationseinheiten. Während moderne Sprachen diese oft einer Modulstruktur folgen lassen, kennt C nicht einmal eine Unterteilung in Module oder ähnliche Organisationseinheiten. Dies stellt keine Einschränkung dar, aber erfordert, eine Struktur der Kompilationseinheiten festzulegen. So wird jedes Curry-Modul in eine C-Quelldatei übersetzt. Die Modulhierarchie wird genau wie in Curry durch Verzeichnisse abgebildet, sodass auch hier sofort Eindeutigkeit gewährleistet werden kann. Um Überlappungen auch mit Nicht-Curry-Modulen zu verhindern, erhält auch im C-Backend jedes Modul das Präfix `curryprog`. Außerdem genügt es nicht, eine C-Quelldatei für ein Modul zu erzeugen. Sollen Funktionen anderer Quelldateien benutzt werden, müssen die Deklarationen oder Vorausdeklarationen dieser Funktionen bereits bekannt sein. In C werden dafür Headerdateien genutzt. Diese werden für jedes Modul zusätzlich zu den Quelldateien erzeugt. Außerdem enthalten diese Definitionen für alle Konstruktoren, sodass im Quelltext ein sprechender Name verwendet werden kann, während der Ausführung aber günstige Ganzzahlkonstanten zum Einsatz kommen können.

Das Fehlen einer vorgegebenen Modulstruktur zeigt sich auch bei der Übersetzung von Funktionen. Um Eindeutigkeit gewährleisten zu können, ist der Name einer Funktion aus dem gesamten hierarchischen Modulnamen zusätzlich zu dem lokalen Namen zusammengesetzt. Ein flacher und in C gültiger Bezeichner lässt sich daraus leicht ableiten, indem die Komponenten durch einen Punkt miteinander verbunden werden und im Ergebnis Sonderzeichen durch Digraphen ersetzt werden. Dabei zeigt sich jedoch oft das Problem, dass die ersten 31 Zeichen eines Bezeichners für eine Funktion eindeutig sein müssen [C99, §5.2.4.1]. Da ein oft signifikanter Teil dieses Bereichs bereits durch Präfix und Modulnamen verwendet wird, stehen zu wenige Zeichen zur Verfügung, um den lokalen Namen einer Funktion zu fassen. Verstärkt wird dies durch die Tatsache, dass durch Lambda-Lifting entstandene oder für Typklassen erzeugte Funktionen oft bereits einen langen lokalen Namen tragen, da der ursprünglich notierte Name während dieser Prozesse um verschiedene Komponenten erweitert wird. Entsprechend genügte es auch nicht, den lokalen Namen vor die Modulhierarchie zu stellen. Als Lösung für dieses Problem wird der Bezeichner wie beschrieben berechnet und, falls er zu lang ist, sein MD5-Hash gebildet. Diese Berechnung erfolgt durch das Programm *md5sum*, das sich auf vielen gängigen Systemen findet. Die C-Funktion erhält dann einen Namen, der aus `C_` gefolgt von diesem Hash besteht. So soll ein möglichst großer Teil der zur Verfügung stehenden Zeichen zur eindeutigen Identifikation

einer Funktion verwendet werden können. Zwar kann immer noch nicht ausgeschlossen werden, dass die so erzeugten Namen zweier unterschiedlicher Funktionen gleich sind, aber für diesen Einsatzzweck ist ein solches Risiko akzeptabel, da C-Compiler oder Linker dies im weitergehenden Übersetzungsvorgang erkennen können und eine Warnung oder einen Fehler anzeigen können.

6.4. Organisation des Laufzeitsystems

Das Laufzeitsystem ist sehr ähnlich strukturiert wie das für die Sprache Python. `curry_types.c/curry_types.h` enthalten die Datenstruktur für Knoten sowie Funktionen und Makros, um mit Knoten zu arbeiten. `curry_helpers.c/curry_helpers.h` bieten eine Reihe von Makros und Funktionen, die das Implementieren externer Funktionen erleichtern sollen. Diese werden in Abschnitt 6.5 näher erklärt. Da C keine primitiven Datenstrukturen bietet, sind einfache Implementierungen von *Stack*, *Queue* und *DecisionMap* in `curry_support_structures.c/curry_support_structures.h` angegeben. All diese Strukturen sind je als verkettete Liste ausgelegt. Während dies eine sinnvolle Struktur für *Stack* und *Queue* darstellt, wäre für die *Decision Map* eine baumartige Struktur mit schnellerem Zugriff eine möglicherweise bessere Wahl. Da in dieser Arbeit aber die Korrektheit sowie die Einfachheit eines Backends wichtiger ist als eine optimale Laufzeit, wird hier bewusst auf die Implementierung einer komplexeren Datenstruktur verzichtet. `curry_runtime.c/curry_runtime.h` enthalten eine Implementierung des Laufzeitsystems sowie Funktionen, um mit dem Laufzeitsystem zu interagieren, wie beispielsweise eine Funktion `curry_add_eval`, mit der die Auswertung eines Ausdrucks in die Warteschlange einer Laufzeitsystem-Instanz eingereiht werden kann. Mittels der `curry_alloc.h` werden Speicherallokationen gemäß des gewählten Allokationsmechanismus an `malloc` oder den *BoehmGC* weitergeleitet. In `currynative/Prelude.c/currynative/Prelude.h` und weiteren Dateien nach diesem Schema beinhaltet das Laufzeitsystem die Implementierungen externer Funktionen der Standardbibliothek.

Obwohl es möglich ist, das Laufzeitsystem zu einer Bibliothek zu übersetzen, ergeben sich daraus mehrere Nachteile. So erfolgt die Auswahl, welcher Allokationsmechanismus verwendet werden soll, bereits zur Kompilierzeit. Damit ist eine Bibliothek, die das Laufzeitsystem enthält, bereits darauf festgelegt. Wird das Programm übersetzt, so muss es den gleichen Mechanismus wählen. Ein Wechseln zwischen den Mechanismen, beispielsweise um die Korrektheit mit beiden Mechanismen zu prüfen, erfordert so mehrere Buildverzeichnisse oder ein Neukompilieren der Bibliothek mit anderen Einstellungen.

Weiter kann das Laufzeitsystem nicht vollständig isoliert von einem Programm existieren. Manche extern definierten Funktionen greifen auf Funktionen und Konstruktoren der *Prelude* zurück. Diese sind aber nicht notwendigerweise extern, sondern werden aus dem Curry-Quelltext der *Prelude* übersetzt. Entsprechend liegen sie erst dann vor, wenn auch ein Programm übersetzt werden soll. Um das Laufzeitsystem dennoch zu einer Bibliothek übersetzen zu können, enthält es in `curry_shim_prelude.h` alle Definitionen, die zum Übersetzen

der externen Funktionen notwendig sind. Da dabei aber die numerischen Identifikatoren der Konstruktoren bereits fest eingesetzt sein müssen, können Abweichungen zwischen Laufzeitsystem und übersetzten Programmteilen entstehen, falls sich Änderungen im Übersetzungsprozess oder in der *Prelude* der Curry-Standardbibliothek ergeben. Diese führen zu fehlerhaften Programmen.

Wird das Laufzeitsystem zusammen mit den Curry-Programmteilen übersetzt, ergeben sich diese Probleme nicht, daher ist dies das empfohlene Vorgehen.

6.5. Hilfsmakros für Interoperabilität

Das C-Backend soll native Implementierungen für die externen Funktionen der Curry-Standardbibliothek bieten, genau wie es auch das Python-Backend tut. Entsprechend stellen sich die gleichen Herausforderungen wie bei der Implementierung in Python. Die zur Verfügung stehenden sprachlichen Möglichkeiten sind allerdings sehr verschieden zu denen, die Python anbietet. Während das Python-Laufzeitsystem mit Dekoratoren und somit funktionalen Mechanismen arbeitet, um die Entwicklung nativer Implementierungen zu erleichtern, hat C keine vergleichbare Funktionalität.

Stattdessen bietet C einen Präprozessor. Mit diesem können Makros definiert werden, die durch textuelle Ersetzungen syntaktischen Zucker bereitstellen können. Zwar sind die in Python genutzten Dekoratoren durch den vollständigen Sprachumfang mächtiger, aber da sie zur Laufzeit ausgewertet werden müssen, haben sie einen negativen Einfluss auf die Ausführungsgeschwindigkeit des Programms. Da Makros im Programmtext direkt durch die angegebenen Anweisungen ersetzt werden, verlangsamen sie das Programm bei korrekter Verwendung nie gegenüber einer Implementierung, die auf die Makros verzichtet.

Ähnlich wie die Dekoratoren finden auch die Makros hauptsächlich Anwendung beim Ein- und Ausstieg in die Funktion. Bereits für die Einleitung der Funktionsdefinition ist ein Makro hilfreich, der dem Knoten einen einheitlichen Namen gibt. Das erlaubt es, Zugriffe auf Kinder des Knotens leicht mittels eines weiteren Makros zu verkürzen.

```
#define CURRY_FUNC(name) int name(Node *node)
#define ARG(arg) (node_getArg(node,(arg)))
```

Anstatt Argumente vor der Funktion als strikt zu erklären, kann dies durch ein Makro innerhalb des Funktionsrumpfes geschehen. Damit wird die gleiche Struktur erreicht, die auch übersetzte Funktionen annehmen:

```
#define STRICT_ARG(arg) if(!node_isHNF(ARG(arg))) return ((arg)+1)
```

Schließlich muss bei erfolgreicher Berechnung der Funktion ein Wert zurückgegeben werden, indem er in den Knoten geschrieben wird. Durch `CURRY_RETURN` kann dies zusammen mit dem tatsächlichen Beenden der Funktion geschehen:

```
#define CURRY_RETURN(v) \
  do { \
```

```
    node_setContent(node, v); \
    return COULD_EXECUTE; \
  } while(0)
```

Damit kann die bereits gesehene +$-Funktion wieder sehr natürlich implementiert werden. CONTENT_LIT_INT ist dabei ein Makro um leicht ein Ganzzahlliteral in node_setContent und somit auch CURRY_RETURN angeben zu können. Es fügt den notwendigen Typcast hinzu und zeigt gleichzeitig an, dass der Knoten keine Kinder haben soll.

```
CURRY_FUNC(currynative_df_dPrelude_d_p_r) {
  STRICT_ARG(0);
  STRICT_ARG(1);

  CURRY_RETURN(CONTENT_LIT_INT(
    node_getIntLiteral(ARG(0))
    +node_getIntLiteral(ARG(1))));
}
```

Besonders auch für die Unterstützung von IO-Funktionen sind unterstützende Konstrukte hilfreich.

```
#define CURRY_IO_WAIT_WORLD() \
  STRICT_ARG(node_numArgs(node)-1)

#define CURRY_IO_RETURN(v) \
  do { \
    node_setContent(node, \
      CONTENT_CCALL(CURRY_WORLD_WORLD,1,((Node*[]) { \
        node_construct(v) \
      })) \
    ); \
    return COULD_EXECUTE; \
  } while(0)
```

CURRY_IO_WAIT_WORLD leitet eine IO-Funktion ein, indem es auf das Weltobjekt wartet, das per Konvention immer als letztes Argument übergeben wird. CURRY_IO_RETURN verpackt das Ergebnis in einen Konstruktoraufruf. Das ist aus zwei Gründen notwendig. >>=$ erwartet immer ein Ergebnis, das an die rechtsseitige Funktion übergeben werden kann. Dazu muss die linksseitige Aktion vollständig abgeschlossen sein. Das wird daran erkannt, dass der Knoten keinen Funktionsaufruf mehr enthält. Für das Ergebnis einer IO-Aktion `IO (a -> b)`würde ohne das Verpacken in einen Konstruktoraufruf nicht erkannt werden, wann die IO-Aktion beendet ist und die (noch nicht vollständig applizierte) Ergebnisfunktion würde ausgewertet. IO-Aktionen können auch fehlschlagen. Damit der Fehler weitergereicht werden kann, wird anhand des Konstruktors angezeigt, ob ein Ergebnis erreicht wurde oder ein Fehler weitergegeben muss.

Die Funktion **putChar** kann so sehr natürlich übersetzt werden

```
CURRY_FUNC(currynative_df_dPrelude_dprim_uputChar) {
  CURRY_IO_WAIT_WORLD();
  putc(node_getCharLiteral(ARG(0)), stdout);
  CURRY_IO_RETURN(CURRY_UNIT);
}
```

Kapitel 7

Evaluation

Die in dieser Arbeit entwickelten Backends werden unter zwei Gesichtspunkten untersucht. Einerseits ist es immer interessant, wie schnell die Programme sind, die von einem Compiler übersetzt wurden. Daher werden einige Geschwindigkeitsvergleiche zwischen *curry2python* und *curry2c* sowie den etablierten Curry-Systemen *PAKCS* und *KiCS2* angestellt.

Andererseits wird in dieser Arbeit mehrfach betont, dass der Fokus nicht auf besonders optimierten Backends liegt, sondern darauf, ein Backend mit möglichst wenig Aufwand entwickeln zu können. Daher wird die Komplexität der Backends anhand der Anzahl der Codezeilen untersucht und mit der Komplexität von *Cam* und *Hurry* verglichen, die ebenfalls in imperative Sprachen übersetzen.

7.1. Geschwindigkeit

Die verwendeten Benchmarks sind so aufgebaut, dass sie entweder funktional sind und die logischen Fähigkeiten der Sprache Curry nicht verwenden, oder dass sie bewusst Nichtdeterminismus in hohem Ausmaß verwenden, um die Leistung der logischen Seite zu überprüfen. Alle Benchmarks stammen dabei ursprünglich aus dem *KiCS2*-Projekt, sind aber leicht abgewandelt, um leichter automatisch verschiedene Problemgrößen verwenden zu können.

Die Benchmarks werden auf einem Intel i7-6700k mit vier Kernen und einem Grundtakt von 4 GHz ausgeführt, dem 32 GiB DDR4-3200 zur Seite stehen. Als Betriebssystem arbeitet Gentoo Linux. Die *KiCS2*-Benchmarks werden von KiCS2 in der Version 2.0.0-b13 auf GHC der Version 8.0.2 übersetzt. Dabei wird zum einen die Standard-Suchstrategie der Breitensuche verwendet (*KiCS2* in den Vergleichstabellen), zum anderen wird auch die ebenfalls angebotene Tiefensuche ausprobiert (*KiCS2dfs* in den Tabellen). *PAKCS*-Benchmarks nutzen PAKCS 2.0.1 auf SWI-Prolog 7.4.2. Die nach Python übersetzten Benchmarks werden nicht mit dem Standardinterpreter *CPython* ausgeführt, sondern stattdessen mit dem wesentlich schnelleren *PyPy3* in der Version 6.0.0. Als C-Compiler zur weiteren Übersetzung der durch das *curry2c* erzeugten Programme kommt GCC 7.3.0 zum Einsatz. Dabei wird die Optimierungsstufe `-O3` sowie *link time optimization* (`-flto`) benutzt. Alle Benchmarks werden automatisch durch das Werkzeug *bench* [1] ausgeführt, das selbstständig die Ergebnisse mittelt.

[1] https://github.com/Gabriel439/bench/, zuletzt abgerufen 2018-09-19

7.1.1. Funktionale Benchmarks erster Ordnung

Die Benchmarks *Reverse*, *ReverseUser*, *Tak* und *TakPeano* nutzen weder logische Konstrukte noch Funktionen höherer Ordnung. *Reverse* dreht dabei eine Liste um. *ReverseUser* tut das gleiche, verwendet dabei aber nicht die eingebaute Listendarstellung, sondern definiert einen eigenen, zu dieser Darstellung äquivalenten Datentyp. *Tak* berechnet das Ergebnis einer hochgradig rekursiven Funktion. *TakPeano* berechnet das Ergebnis der gleichen Funktion, nutzt dabei aber Peano-Zahlen statt des nativen `Int`-Datentyps.

Die Größen haben in diesen Benchmarks keinen natürlichen Bezug zu den jeweiligen realen Problemgrößen. Allerdings sind sie derart gestaltet, dass die Laufzeiten von *Reverse* mit *ReverseUser* sowie *Tak* mit *TakPeano* verglichen werden können.

Von manchen Benchmarks werden mehrere Problemgrößen aufgelistet. Das ist hauptsächlich dem Umstand geschuldet, dass es sehr große Laufzeitunterschiede zwischen den schnellen Systemem KiCS2 und *curry2c* und den langsamen Systemen PAKCS und *curry2python* gibt.

Tabelle 7.1. Ergebnisse funktionaler Benchmarks erster Ordnung

	Größe	KiCS2	KiCS2dfs	PAKCS	C	PyPy
Reverse	2	0,21s	0,18s	8,52s	0,88s	2,15s
Reverse	3	16,97s	14,18s	147,7s	15,67s	49,93s
ReverseUser	2	0,21s	0,18s	8,57s	0,86s	2,18s
ReverseUser	3	18,07s	15,57s	145,2s	15,70s	44,49s
Tak	0	0,09s	0,08s	44,80s	1,56s	6,61s
Tak	1	0,33s	0,29s	163,3s	5,91s	25,60s
TakPeano	0	0,16s	0,13s	57,58s	2,81s	14,95s
TakPeano	1	0,60s	0,50s	213,7s	9,70s	54,04s

Allgemein bietet KiCS2 die kürzesten Laufzeiten, die Verwendung der Tiefensuche hat dabei einen kleinen Laufzeitvorteil gegenüber der Breitensuche. *curry2c* ist in den meisten Benchmarks merklich langsamer. Eine Ausnahme davon sind die größeren *Reverse*- und *ReverseUser*-Benchmarks. Hier erreicht die Übersetzung nach C eine leicht schnellere Ausführung als KiCS2. Das Python-Backend erzeugt erwartungsgemäß langsamere Programme als das C-Backend. PAKCS bildet das Schlusslicht.

Während der Verzicht auf die eingebaute Listendarstellung in KiCS2 und PAKCS einen leichten negativen Einfluss auf die Ausführungsgeschwindigkeit hat, zeigt sich praktisch kein Unterschied zwischen *Reverse* und *ReverseUser* für die Übersetzung nach dieser Arbeit. Das ist einleuchtend, da es hier keine gesonderte Übersetzung für den Listentyp gibt und dieser genau wie jeder andere Datentyp übersetzt und verwendet wird.

7.1.2. Funktionale Benchmarks höherer Ordnung

Die Benchmarks *ReverseHO*, *Primes*, *PrimesPeano*, *Queens* und *QueensUser* nutzen Funktionen höherer Ordnung, aber keinen Nichtdeterminismus. *ReverseHO* dreht eine Liste unter Verwendung einer selbst definierten `fold`-Funktion um. *Primes* berechnet Primzahlen nach einer Siebmethode unter Verwendung einer selbst definierten Funktion **`filter`**. *PrimesPeano* berechnet Primzahlen nach der gleichen Methode, verwendet aber Peano-Zahlen statt des nativen **`Int`**-Datentyps. *Queens* berechnet, wie viele Lösungen für das Damenproblem auf einem Schachbrett existieren, und verwendet dazu List Comprehensions. *QueensUser* bildet die Berechnung des *Queens*-Benchmarks auf einem eigenen Listen-Datentyp nach, wobei natürlich auf den syntaktischen Zucker der List Comprehensions verzichtet werden muss und stattdessen der Programmtext notiert ist, der während der Übersetzung aus den List Comprehensions erzeugt würde.

Die Größe des *ReverseHO*-Benchmarks ist wieder ohne natürlichen Bezug zur Größe des tatsächlichen Problems. In den *Primes*-Benchmarks gibt die Größe an, bis zu welcher Primzahl gesucht wird, wären diese durchnummeriert. In den *Queens*-Benchmarks entspricht die Größe der Kantenlänge des Schachfelds.

Tabelle 7.2. Ergebnisse funktionaler Benchmarks höherer Ordnung

	Größe	KiCS2	KiCS2dfs	PAKCS	C	PyPy
ReverseHO	3	0,003s	0,003s	0,29s	0,02s	0,47s
Primes	4000	2,23s	1,77s	598,4s	21,74s	217,2s
Primes	8000	11,27s	9,02s	—	85,18s	—
PrimesPeano	200	0,16s	0,28s	44,41s	0,61s	3,50s
PrimesPeano	400	1,48s	2,22s	382,4s	4,60s	22,23s
Queens	9	0,05s	0,04s	20,16s	1,00s	11,20s
Queens	10	0,25s	0,20s	119,5s	4,77s	46,75s
QueensUser	9	0,06s	0,04s	29,58s	1,12s	11,08s
QueensUser	10	0,27s	0,22s	188,9s	6,39s	55,13s

Aus Zeitgründen wird auf eine Angabe der PAKCS- und Python-Laufzeiten für den größeren *Primes*-Benchmark verzichtet.

Bei Verwendung von Funktionen höherer Ordnung zeigt sich ein ähnliches Bild wie bei der Verwendung von ausschließlich Funktionen erster Ordnung. Auffällig ist, dass sich der zuvor beobachtete Unterschied zwischen eingebautem und selbst definiertem Listen-Datentyp hier anders darstellt. Mit KiCS2 sind beide Darstellungen praktisch gleich schnell, aber beide Backends dieser Arbeit zeigen eine langsamere Ausführungsgeschwindigkeit bei Verwendung eines eigenen Listentypen. Eine mögliche Erklärung dafür ist, dass durch das Desugaring der List Comprehensions im nicht-*User*-Benchmark anderer, effizienterer Code entsteht als der handgeschriebene Code des *User*-Benchmarks.

7.1.3. Logisch-Funktionale Benchmarks

Die Benchmarks *PermSort*, *PermSortPeano* und *Last* nutzen Nichtdeterminismus in ihren Berechnungen. *PermSort* sortiert eine Liste, indem nichtdeterministisch mittels des *Choice*-Operators alle möglichen Permutationen einer Liste erzeugt werden und dann alle unsortierten Permutationen verworfen werden. *PermSortPeano* tut das gleiche, verwendet aber Peano-Zahlen. *Last* berechnet das letzte Element einer Liste `xs`, indem die Gleichung `xs =:= xs' ++ [x]` unter Verwendung freier Variablen `xs'` und `x` nach `x` aufgelöst wird.

Für alle Benchmarks dieses Abschnitts gibt die Größe die Länge der jeweiligen Liste an.

Tabelle 7.3. Ergebnisse logisch-funktionaler Benchmarks

	Größe	KiCS2	KiCS2dfs	PAKCS	C	PyPy
Last	4000	0,04s	0,03s	0,53s	0,16s	45,77s
Last	8000	0,09s	0,05s	1,03s	0,54s	195,4s
PermSort	13	3,51s	0,66s	10,70s	0,04s	16,07s
PermSort	14	11,17s	1,98s	32,15s	0,10s	49,59s
PermSortPeano	12	12,10s	2,79s	5,57s	0,20s	193,4s
PermSortPeano	13	40,13s	9,55s	18,10s	0,51s	—
PermSortPeano	14	122,2s	32,76s	57,73s	1,15s	—

Aus Zeitgründen wird auf eine Angabe der Python-Laufzeiten für die größeren *PermSortPeano*-Benchmarks verzichtet.

Nutzt ein Programm Nichtdeterminismus, so findet sich ein anderes Muster in den Ergebnissen. *Last* ist dabei noch sehr ähnlich zu den vorangegangenen Benchmarks. Allerdings zeigt sich bereits, dass PAKCS im Vergleich zu den anderen Systemen besser abschneidet als bei rein deterministischen Problemen. In den stark nichtdeterministischen *PermSort*-Benchmarks ist die Leistung von KiCS2 mit Verwendung von Breitensuche eher schlecht. Nicht nur die Übersetzung nach C ergibt eine schnellere Ausführung, auch PAKCS ist ähnlich schnell wie KiCS2 mit Breitensuche. Da PAKCS mit Prolog als Übersetzungsziel eine Sprache verwendet, die selbst bereits nichtdeterministisch ist, lässt sich der Vorteil von PAKCS hier durch eine bessere Unterstützung von Nichtdeterminismus und damit zusammenhängenden Optimierungen erklären. Genauso ist KiCS2 mit Tiefensuche deutlich schneller. Die Breitensuche ist also eine aufwändigere Methode zum Umgang mit Nichtdeterminismus. Der Unterschied zwischen C und Python ist stärker ausgeprägt als in den vorherigen Benchmarks. Das deutet darauf hin, dass die Datenstruktur, die die *Decision Map* bereitstellt, in C effizienter umgesetzt ist als in Python.

7.2. Vollständigkeit ohne Terminierung

Nichtdeterministische Programme arbeiten an mehreren Möglichkeiten, die Lösung eines Ausdrucks sein kann. Es ist eine weitere Differenzierung nötig, zu welchen Ergebnissen

ein Programm gelangen kann, sobald eine nichtdeterministische Möglichkeit mit einer unendlichen Berechnung beschäftigt ist. Vollständigkeit sei hier derart definiert, dass solche Ergebnisse, die berechnet werden können, auch berechnet werden, sodass nach endlicher Zeit höchstens noch solche Berechnungen in Arbeit sind, die sich mit einer Endlosschleife beschäftigen.

Ob dies erreicht werden kann, ist allerdings davon abhängig, wie der Nichtdeterminismus implementiert ist. Einfache Suchen, die erst (Teil-)Ausdrücke auswerten, bevor sie sich anderen nichtdeterministischen Möglichkeiten zuwenden, sind nach dieser Definition nicht vollständig.

```
loop :: a
loop = loop

nt1 = 42 ? loop
nt2 = loop ? 42
nt3 = loop ? 42 ? loop
```

Die Funktion `loop` ist eine Endlosschleife, sodass ein Versuch, sie zu einem Ergebnis auszuwerten, nie terminiert. Die Funktionen `nt1` und `nt2` können auch von einfachen Suchen vollständig berechnet werden. `nt1` ergibt beispielsweise das Ergebnis `42`, falls die linke Seite des `?`-Operators zuerst ausgewertet wird. `nt2` wird analog vollständig berechnet, falls die rechte Seite zuerst ausgewertet wird.

`nt3` schließt den berechenbaren Ausdruck zwischen zwei Endlosschleifen ein, sodass weder der Versuch, zuerst die linke Seite auszuwerten, noch der Versuch, zuerst die rechte Seite auszuwerten, das Ergebnis `42` liefert. *PAKCS* nutzt eine Tiefensuche, während *KiCS2* in Standardkonfiguration eine Breitensuche durchführt. Diese versuchen immer, einen Ausdruck auszuwerten und können die Funktion `nt3` nie zu dem Ergebnis `42` auswerten. Das Fair Scheme erlaubt eine Verschränkung der Auswertung der einzelnen nichtdeterministischen Möglichkeiten, sodass die vom Python- und vom C-Backend übersetzten Programme jeweils das Ergebnis `42` liefern, bevor sie in einer Endlosschleife festhängen.

7.3. Implementationsaufwand eines Backends

Der nötige Aufwand, um ein neues Backend zu implementieren, ist natürlich eine subjektive Größe und lässt sich damit nicht allgemein und aussagekräftig bemessen. Daher ist in dieser Arbeit mit der Anzahl an Codezeilen nur eine sehr einfache Metrik gewählt, um zumindest einen Anhaltspunkt für die Komplexität eines Backends zu erhalten. Grob kann festgehalten werden, dass bei korrekter Anwendung gängiger Programmiersprachen mehr Codezeilen eine größere Komplexität bedeuten. Davon ausgenommen sind natürlich solche Zeilen, die leer sind oder lediglich Kommentare enthalten. Da diese das Verhalten des Programms nicht ändern, sollen sie nicht mitgezählt werden.

Es werden vier Backends verglichen. Einerseits werden die in dieser Arbeit entstandenen Backends nach Python und nach C betrachtet. Andererseits wird ebenfalls die Komplexität der

Backends zweier vorheriger Arbeiten, *Cam* (übersetzt nach Java) und *Hurry* (nach JavaScript) untersucht. Auf einen Vergleich mit *KiCS2* (nach Haskell), *PAKCS* (nach Prolog) oder *Sprite* (nach LLVM-IR) wird verzichtet, da die Zielsprachen dieser Systeme anderen Paradigmen folgen.

Für jeden dieser drei Compiler werden die Codezeilen dreier Bestandteile gezählt. Erstens ist das der Übersetzer, der das ICurry-Zwischenformat in die Zielsprache übersetzt. Darunter fällt sämtlicher Code, der notwendig ist, um von einem abstrakten ICurry-Programm zum Zielprogramm in konkreter Syntax zu gelangen, nicht aber Funktionen zum Lesen oder Schreiben von Quelltextdateien oder zur Verwaltung des Buildvorgangs. Zweitens wird die Größe des Laufzeitsystems verglichen. Darunter wird hier solcher handgeschriebener Quelltext in Zielsprache verstanden, der den Ablauf des Programms koordiniert und insbesondere noch keine Implementierungen externer Funktionen enthält. Zusätzliche Funktionalität, die nicht zum Ausführen eines Curry-Programms notwendig ist, wird dabei, soweit dies durch Ausschließen ganzer Dateien möglich ist, ebenfalls nicht mitgezählt. Die externen Implementierungen werden als dritter Teil gesondert untersucht. Dabei wird sich auf die `Prelude` beschränkt, da die anderen Module der Standardbibliothek nicht in allen Systemen implementiert sind.

Zum Zählen der Curry-Codezeilen der Übersetzerteile wird ein Curry-Skript verwendet, das leere und Kommentarzeilen überliest und die übrigen Zeilen zählt. Für alle anderen Dateien wird das Programm *cloc*[2] in der Version 1.72 genutzt, das ebenfalls nur solche Zeilen zählt, die das Programmverhalten beeinflussen.

Tabelle 7.4. Anzahl Codezeilen verschiedener Backends

	Python	C	Cam	Hurry
Übersetzer	342	441	790	632
Laufzeitsystem	422	1074	4595	319
externe Implementierungen	294	672	2175	228

In diesem Vergleich fällt zunächst auf, dass sich die Größen der Übersetzer nicht sehr voneinander unterscheiden. Der kleine Unterschied zwischen Python und C lässt sich leicht durch den zusätzlichen Bedarf einer Headerdatei neben jeder Quelldatei sowie den zusätzlichen Maßnahmen zum Sicherstellen eindeutiger Namen erklären. Der Unterschied zwischen den Übersetzern dieser Arbeit zu den vorherigen Arbeiten ist darauf zurückzuführen, dass sowohl Cam als auch Hurry je eine zusätzliche abstrakte Darstellung eines Java- bzw. JavaScript-Programms verwenden. Darauf verzichten die Übersetzer dieser Arbeit bewusst, da zwischen ICurry und der jeweiligen Zielsprache kein großer struktureller Unterschied existiert. Eine Umformung in ein zusätzliches abstraktes Format würde so nur minimale Änderungen durchführen und die meisten Konstrukte direkt auf je ein passendes Konstrukt der abstrakten Darstellung der Zielsprache abbilden. Dies verschiebt den Prozess, das Programm in einer

[2] https://github.com/AlDanial/cloc, zuletzt abgerufen 2018-09-30

konkreten Syntax darzustellen, lediglich. Zusätzlich ist zu erwähnen, dass Hurry keinen konkreten JavaScript-Code erzeugt, sondern das Programm in einem weiteren Zwischenformat, *esTree*[3], ablegt, aus dem erst mit einem externen Werkzeug konkreter JavaScript-Code generiert wird.

Bei curry2python, curry2c sowie Hurry ist das Laufzeitsystem etwa 40-60% größer als die externen Implementierungen der Prelude. Das C-Backend zeigt dabei von den drei Systemen den größten Unterschied zwischen diesen Teilen. Dies ist nicht verwunderlich, da C keine Datenstrukturen bereitstellt. Diese müssen selbst implementiert werden und vergrößern so das Laufzeitsystem. Auch deshalb zeigt sich wieder das bereits gesehene Bild, das das C-Backend aufwändiger ist als das Python-Backend. Überraschenderweise sind sowohl Hurry als auch Cam die Extreme in diesem Vergleich. Hurry bietet das kleinste Laufzeitsystem und die kleinste Größe an externen Implementierungen. Während keine Begründung für das kleine Laufzeitsystem angegeben werden kann, zeigt ein Blick in die externen Implementierungen, dass diese sämtlichen Umgang mit `IO` vermissen lassen, also unvollständig sind. Eine vollständige Implementierung der Prelude würde einen besseren Vergleich erlauben. Als grober Anhaltspunkt kann verwendet werden, dass sowohl im Python- als auch im C-Backend dieser Arbeit etwa ein Drittel der Codezeilen auf die Implementierung der Funktionen zur Unterstützung von `IO` entfallen. Wird die Anzahl der Codezeilen der externen Hurry-Implementierungen um dieses Drittel erhöht, hat dieses mit 342 Codezeilen etwas mehr als das Python-Backend.

Die Zeilenzahl des Laufzeitsystems und der externen Implementierungen Cams sind etwa um einen Faktor 4,3 und 3,2 größer als die des C-Backends. Damit sind diese Java-Implementierungen Ausreißer in dieser Messung, für die keine einfache Begründung angegeben werden kann. Am wahrscheinlichsten ist das Zusammentreffen verschiedener Faktoren, wie der Geschwätzigkeit Javas, einer Unterteilung in viele kleine Klassen in je einer eigenen Datei und die Verwendung gängiger Software-Entwicklungsmuster.

[3] https://github.com/estree/estree/blob/master/es5.md, zuletzt abgerufen 2018-09-30

Kapitel 8

Ausblick

Im Rahmen dieser Arbeit sind das ICurry-Zwischenformat sowie Backends für die Sprachen Python und C entwickelt worden. Das Zwischenformat sowie die Übersetzungen in die Zielsprachen können dabei als vollständig angesehen werden. Es sollte sich jedes Curry-Programm nach ICurry und weiter nach Python oder C übersetzen lassen. Die nativen Implementierungen externer Funktionen sind aber nur für je eine unterschiedlich große Teilmenge der benötigten Funktionen vorhanden. Für Python stehen Implementierungen für einen Großteil der Standardbibliothek zur Verfügung. In C ist lediglich die `Prelude` implementiert.

Insbesondere verzichten beide Backends bewusst auf jede Art von Unterstützung eingekapselter Suche. Diese ist in ihrer derzeitigen Unterstützung vergleichsweise aufwändig. Vor Kurzem ist jedoch eine neue Methode vorgestellt worden, wie sich *Set Functions*, also solche Funktionen, die mit einer deterministischen Menge von Werten arbeiten, anstatt sie durch Nichtdeterminismus auszudrücken, bereits im Compiler-Frontend synthetisieren lassen [AHT18]. Diese benötigt, abgesehen von einer Funktion, die prüft, ob ein Term ein Fehlschlag ist, keine zusätzliche Unterstützung durch externe Funktionen. Sobald das verwendete Curry-Frontend diese Synthese beinhaltet, kann die eingekapselte Suche leicht durch Implementierung dieser einfachen Funktion unterstützt werden.

Weder das Python- noch das C-Backend enthalten viele Optimierungen bezüglich ihrer Laufzeit. Daher ist es denkbar, dass durch besser geeignete Datenstrukturen beispielsweise für die *Decision Map* oder eine intelligentere Speicherverwaltung die Laufzeiten noch merklich verkürzt werden können. Genauso könnte eine wählbare Suchstrategie eine bessere Anpassung an die jeweilige Situation erlauben und so zumindest in solchen Programmen, die nicht auf die hier implementierte vollständige Breitensuche angewiesen sind, einen weiteren Leistungszuwachs erreichen.

Schließlich muss sich auch durch die Verwendung in weiteren, neuen Backends für andere Sprachen zeigen, ob Schwächen im Zwischenformat oder den bereitgestellten Bibliotheken zum Umgang mit diesem verblieben sind.

Anhang A

Nutzungsanleitung

In diesem Dokument soll kurz beschrieben werden, wie *icurry* und die beiden Backends für Python und C installiert und in einem einfachen Projekt verwendet werden können. Dazu wird vorausgesetzt, dass die folgenden Softwarepakete installiert und funktionsfähig sind:

- Aktuelles GNU/Linux Betriebssystem. Andere Betriebssysteme können funktionieren, sind aber nicht getestet.
- KiCS2 ab Version 2.0.0-b13.
- Python3 ab Version 3.5. Getestet sind CPython und PyPy.
- Ninja ab Version 1.8.2. Ältere Versionen können funktionieren, sind aber nicht getestet.
- GCC in beliebiger Version (mit Unterstützung für C99). Nur bei Verwendung der Übersetzung nach C. Andere C-Compiler sind auch möglich, erfordern aber weitere Anpassungen der Build-Skripte eines Programms.
- md5. Nur bei Verwendung der Übersetzung nach C.
- Make. Nur für die automatische Installation. Getestet ist GNU Make 4.2.1.

A.1. Installation

Die Installation erfolgt wie in der `INSTALL.txt` beschrieben.

```
mkdir build
cd $_
../configure     #ggf. mit Optionen, siehe ../configure --help
make
make install     #ggf. mit erhöhten Privilegien
```

Wird kein `prefix` an das `configure`-Skript übergeben, so wird nach `/usr/local` installiert. Der `binpath` muss im `PATH` zu finden sein, damit die ausführbaren Dateien dieses Programms gefunden werden können.

Eine manuelle Installation ist möglich, indem – in dieser Reihenfolge – die Pakete `ninja`, `icurry`, `python`, `c` jeweils zum Paketindex hinzugefügt und dann installiert werden.

A.2. Schnellstart

Für Python und C finden sich minimale Beispielprogramme unter `python-example` und `c-example`. Sie beinhalten je ein durch *cpm* erzeugtes Projekt, einen Einstiegspunkt in der jeweiligen Sprache, der eine Instanz des Laufzeitsystems erzeugt, und darin die Curry-Funktion auswertet, sowie eine kurze Anleitung, wie das Beispiel übersetzt und ausgeführt werden kann. Im Fall des C-Beispiels ist dazu eine Steuerdatei (`curry2c.build`) nötig, die einige Informationen über die zu übersetzenden C-Quelldateien enthält.

A.3. Nutzung von Curry-Funktionen aus Python

In Python finden Interaktionen mit dem Laufzeitsystem hauptsächlich durch die Klassen `CurryRuntime.CurryRuntime`, `CurryRuntime.Progstate`, `CurryTypes.Node` und `CurryTypes.FCall` statt.

Dazu wird immer ein `CurryRuntime`-, sowie ein `Progstate`-Objekt erzeugt. In das `Progstate`-Objekt werden mittels `addRoot` Aufrufe von Curry-Funktionen eingereiht. Diese können mit der `run`-Methode des `CurryRuntime`-Objekts abgearbeitet werden. Um `IO`-Aktionen in die Warteschlange einzureihen, kann `CurryInterface.RunIO` verwendet werden.

Weitere Methoden zur Interaktion finden sich in den Beschreibungen des `CurryRuntime`-Modules.

A.4. Nutzung von Curry-Funktionen aus C

In C wird das Laufzeitsystem durch die Struktur `CurryRuntime` aus dem `curry_runtime.h`-Header dargestellt. Es kann durch die Prozedur `curry_runtime_new` erzeugt werden. Funktionsaufrufe können durch `curry_add_eval` in die Ausführungswarteschlange hinzugefügt werden. Mittels `curry_run` kann diese abgearbeitet werden. Für die Einreihung von `IO`-Aktionen steht die Prozedur `curry_add_eval_io` zur Verfügung.

Weitere Prozeduren zum Umgang mit dem Laufzeitsystem finden sich in `curry_runtime.h`

A.5. Anbindung eigener externer Funktionen

Um eine selbst definierte externe Funktion im Curry-Teil des Projekts zu nutzen, muss diese zunächst in einem Curry-Modul bekanntgemacht werden. Dabei ist eine Angabe der Signatur notwendig.

```
module MyModule where
-- ...
extDouble :: Int -> Int
extDouble = external
-- ...
```

Sie kann nun im Curry-Code verwendet werden. Die tatsächliche Implementierung unterscheidet sich natürlich, je nachdem, welches Backend verwendet werden soll. In jedem Fall muss die externe Implementierung die Laziness der Sprache Curry beachten. Das bedeutet, dass eine Funktion nicht davon ausgehen darf, dass ein Argument einen Wert enthält. Sie muss zunächst prüfen, ob ein Argument, das sie verwenden will, bereits ausgewertet ist und –falls nicht– zunächst die Auswertung des Arguments fordern.

A.5.1. Python

In Python wird erwartet, dass externe Implementierungen unter `currynative.`*Modulename* zu finden sind, ihren Namen `C_` vorangestellt wird, und alle Sonderzeichen sowohl im Modul- als auch im Funktionsnamen durch Digraphen ersetzt werden.

Für die oben deklarierte Funktion wird also ein Modul benötigt, das im `PYTHONPATH` unter `currynative.MyModule` zu finden ist und folgendes enthält:

```
import CurryInterface

@SimpleFuncAllStrict
def C_extDouble(x):
  return 2*x
```

Hier ist bereits mit `@SimpleFuncAllStrict` ein Dekorator des `CurryInterface`-Moduls verwendet. In diesem Modul finden sich weitere solche Dekoratoren und Funktionen, die das Implementieren externer Funktionen vereinfachen können.

A.5.2. C

In C wird erwartet, dass ein Prototyp der Funktion in einer Headerdatei zu finden ist, die im *Include-Path* unter `currynative/`*Modulename*`.h` gefunden werden kann.

Ihr Name muss, da C kein Namespacing erlaubt, alle Komponenten des qualifizierten Namens enthalten. Er entsteht, indem in `currynative.f.`*Modulename*`.`*Funcname* wiederum alle Sonderzeichen ersetzt werden. Im obigen Beispiel sind das nur die Punkte, sodass der Name `currynative_df_dMyModule_dextDouble` entsteht.

Die Datei `currynative/MyModule.h` sollte also folgendes enthalten:

```
int currynative_df_dMyModule_dextDouble(Node *node);
```

Während auf den Include-Guard hier nur aus Lesbarkeitsgründen verzichtet wird, ist das Inkludieren des `curry_types.h`-Headers in den `currynative`-Headerdateien nicht nötig.

Die Implementierung dieser Funktion kann an beliebiger Stelle geschehen, solange sie während des Linkens des Zielprogramms einbezogen wird:

```
#include<curry_types.h>
#include<currynative/MyModule.h>
#include<curry_helpers.h>
```

```
CURRY_FUNC(currynative_df_dMyModule_dextDouble) {
  STRICT_ARG(0);
  CURRY_RETURN(
    CONTENT_LIT_INT(
      2*node_getIntLiteral(ARG(0))
    )
  );
}
```

Hier sind bereits viele Hilfsmakros der `curry_helpers.h` verwendet. In dieser Headerdatei finden sich noch weitere solche Makros und Funktionen, die das Implementieren externer Funktionen vereinfachen können.

Anhang B

Implementation Guide

This document shall give as much help as possible to implement a new backend on top of the ICurry intermediate format. It includes descriptions for all necessary functions that make up a unoptimized and slow but still complete runtime system, a description on how to translate ICurry constructs and how to implement some of the more complex external functions.

B.1. Runtime System

The runtime system's purpose is to coordinate the execution of functions with respect to their laziness and non-determinism. It shall track all branches of computation, cycle through them to enable some form of concurrent evaluation and dispatch the correct functions.

The runtime system is an implementation of *The Fair Scheme* [AJ13]. It uses a queue of stacks, each representing a branch in the non-deterministic computation.

In this document, we give an abstract API, which can hopefully serve as a guide on how this runtime system could be implemented in any language. We try to keep it language-agnostic, favoring (semi-)formal or informal descriptions over constructs which may be influenced by some programming languages being absent from others.

B.1.1. Data Structures

The complete program state is one big, directed graph. Many operations deal with a single node and its direct children, so the chosen data structure should allow to access this kind of information cheaply. There are no operations, which deal with all nodes or all edges at once, so a direct implementation of the usual mathematical definition of a graph using a set of vertices and a set of edges is not recommended. A faster approach would likely be to encode all outgoing edges directly into each node. While the number of children theoretically is not limited, a suitably high maximum number of children may be selected. This is reasonable as there will never be more children than the arity of a function or constructor. A number of arguments that high is not expected in any program.

This section highlights the important parts of this graph and some structures which hold information used to correctly and efficiently transform this graph.

Node

A node provides the identity for a (sub-)term. This means that evaluating its represented term will not create a new node, but rather update this existing one. Hence, a node needs to be mutable. In addition to its children, it needs to encode a label. This label determines what kind of term this node represents. It can be one of the following:

1. A failure.
2. A function call.
3. A constructor call.
4. A literal.
5. A choice.

Function calls, constructor calls and literals also need to encode, which function or constructor is called or which literal is meant. For functions, this can be achieved by either adding a reference to the function, using a *function pointer*, *callable object* or similar mechanism, or by using a custom dispatcher and some function identifiers. Constructors need to include their type-unique identifier. If the node is labeled to contain a literal, this literal is to be included. Nodes containing a choice additionally need a mechanism to match multiple choice nodes representing the same choice. This will further be explained in section B.1.1.

The runtime system needs to be able to differentiate between a function call and a constructor. However, literals can be seen as a set of constructors and thus a differentiation between a constructor call and a literal is not strictly needed, but may be convenient in many places.

Edge

Edges need to encode a source and a target node. Also, edges are sorted as seen from the outgoing node. There are no labels or additional information. Thus, a useful representation for edges is to include a list of all child nodes into each node.

Matching Choices

When doing a *pull-tab* step in the presence of shared nodes, single nodes have their label copied. While this is not a problem for the other types of labels, it eventually causes all choice nodes to be copied into completely independent nodes. This manifests as a run-time choice semantic, which is not desired. The solution is to mark all copies of a choice as belonging together and to make the same decision for each of these nodes.

A simple mechanism for doing this is to include an incrementing integer as *choice identifier* into a choice-labeled node. This choice identifier is equal for all nodes representing the same choice and different between any two nodes belonging to different choices. However, it may be

convenient to assign these identifiers lazily. By that we avoid having to pass a the mechanism for obtaining new identifiers to all parts of the program but rather lets a few functions in the runtime system deal with organizing these identifiers. Furthermore, lazy assignments reduce wasting identifiers on choices which will never be evaluated. These identifiers are often a finite resource, so avoiding waste is desirable.

Stack

The stack, or rather *a* stack, is a classic stack over references to nodes. It is not necessary for correctly implementing the runtime system, as its information can staightforwardly be computed by walking the graph. However, it is one of the most important parts for making the runtime system efficient.

Map of Decisions

As hinted at in section B.1.1, for nodes representing the same choice the same decision has to be made. Thus, we need to keep the information which way each choice was decided. If the matching is done using the proposed *choice identifiers*, this information can be held as a set of $\{(c, n) \mid c \in \mathbb{N}, n \in \mathbb{N}\}$, where c shall be the choice identifier and n the position of the chosen child. Preferably, this data structure shall allow fast lookup by the *choice id*, so a structure called *map* in many languages is a good fit.

Queue

The queue, being the main component of a breadth-first search in the graph is the core mechanism to achieve computational completeness within non-deterministic programs.

It shall contain pairs of a stack (see B.1.1) and a decision map (see B.1.1). As this structure is a queue, taking from one end and inserting at the other end should be fast. Other operations are not needed to implement the core of the runtime system. Some more advanced mechanisms like the *concurrent and* (Curry: `(&) :: Bool -> Bool -> Bool`) or encapsulated search may want to have finer control of the queue. However, this is currently not in the scope of this document.

B.1.2. Memory Management

The runtime system constantly updates a graph by adding new nodes, creating links between existing nodes and breaking these links. As it does not define, when a node may be discarded, there needs to be a system that can detect whether a node is still in use or can be freed from memory. Simple reference counting is not enough for this task, as the graph is allowed to contain cycles which can be discarded at once, and there is no point where these cycles could be broken using weak references.

Rather, a complete system for garbage collection is needed. If the target language itself relies on a garbage collector, it is wise to utilize this for our data structures. If the target language

requires explicit memory management, a full garbage collector must to be implemented for nodes. It needs to be able to detect stale cycles and free them.

B.1.3. Procedures

Now we will give some procedures that form the runtime system when implemented as described. Depending on the target language, some of these are just different names for built-in functions dealing with data structures.

All procedures are given as `name :: arguments -> return-type`, with `argument` being a comma-separated list of argument definitions. An argument is given as `name:type`. A type can be one of the data structures described above, a base type such as `Int` or `Bool`, a tuple of two types, given as `(type1, type2)`, a list of a type, denoted as `[type]`, or a type with a hint that data shall be given by reference `type*`.

Depending on the procedure, being marked as pass-by-reference has at least one of two implications for an argument or return value. Modifying the argument shall modify the whole state (sometimes called an *out*-argument). Or some data shall be referenced from somewhere else (*sharing*). Thus, all reference-marked types shall adhere to some reference-semantics.

Not being marked as reference, however, does not require passing by value. Those pieces of data may be passed by reference as well, if this holds out the prospect of faster execution.

For brevity, the decision map will be abbreviated as `DM` in type signatures.

`updateNode :: n:Node*, l:Label, c:[Node*] -> Void`

`updateNode` shall write the label `l` into the node `n`, disconnect all child nodes from `n` and then make all nodes in `c` children of `n`.

`push :: s:Stack*, n:Node* -> Void`, **`pop :: s:Stack* -> Node*`**, **`peek :: s:Stack* -> Node*`**

`push`, `pop` and `peek` shall be standard stack operations.

`next :: q:Queue* -> (Stack*, DM*)`, **`enqueue :: q:Queue*, (Stack*, DM*) -> Void`**

`next` and `enqueue` shall return the next element at one end of a queue, while removing it from the queue, or respectively insert an element at the other end of a queue.

`ensureHasChoiceId :: n:Node*`

`ensureHasChoiceId` uses a global supply of identifiers to assign one to a *choice*-labeled node, if this node does not yet have an identifier.

It does not need to check if the passed node is a choice, as it will only be called on choices. It shall execute the following:

```
1 if n.choice_id is <unset> then
2     n.choice_id ← nextId;
3     nextId ← nextId + 1;
```

```
pull :: n:Node*, p:Int -> Void
```

`pull` executes a *pull-tab* step. This is an operation that pulls a choice towards the root of an expression. Instead of passing the choice-labeled node to this procedure, its arguments are a parent node and a position `p`. A node can be referenced from multiple positions, so a position is strictly required to correctly determine, over which edge this pull-tab step shall be executed over.

`pull` shall execute the following:

```
 1 s ← n.label;
 2 cs ← n.children;
 3 c ← cs[p];
 4 ensureHasChoiceId(c);
 5 cid ← c.choice_id;
 6 n.label ← <choice>;
 7 n.choice_id ← cid;
 8 n.children ← [];
 9 forall i ← c.children do
10     n′ ← new Node;
11     n′.label ← s;
12     n′.children ← copy of cs;
13     n′.children[p] ← i;
14     n.children ← n.children + +[n′];
```

```
step :: s:Stack* -> Bool
```

`step` does a single step towards the *Head Normal Form*.

It shall execute the following:

```
1  n ← pop(s);
2  if n.label is constructor or n.label is literal then
3      return |s| > 0;
4  if n.label is <fail> then
5      return False;
6  if n.label is function then
7      f ← n.label;
8      r ← f(n);
9      if r is <no_argument_needed> then
10         push(s, n);
11         return True;
12     if n.children[r] is <choice> then
13         pull(n, r);
14         push(s, n);
15     if n.children[r] is function then
16         push(s, n);
17         push(s, n.children[r]);
18     return True;
19 if n.label is <choice> then
20     return True;
21 return False;
```

`step`'s return value indicates whether the runtime system shall enqueue this nondeterministic branch again. A return value of `False` shows that this branch requires the head-normal form of a node that is a failure, and thus is a failure itself.

```
dispatch :: q:Queue* -> Void
```

`dispatch` is the main procedure coordinating the execution.

It shall execute the following:

```
1  (s, m) ← next(q);
2  n ← peek(s);
3  if n.label is <choice> and |s| = 1 then
4      ensureHasChoiceId(n);
5      b ← m[n.choice_id];
6      if b is <not_found> then
7          for (i, n′) ← zip([0..], n.children) do
8              m′ ← copy of m;
9              m′[n.choice_id] ← i;
10             s′ ← new Stack;
11             push(s′, n′);
12             enqueue(q, (s′, m′));
13     else
14         updateNode(n, n.children[b]);
15         enqueue(q, (s, m));
16 else
17     r ← step(s);
18     if r then
19         enqueue(q, (s, m));
```

```
run :: q:Queue* -> Void
```

`run` shall repeatedly call `dispatch` (see 21) as long as the queue is non-empty:

```
1  while |q| > 0 do
2      dispatch(q);
```

B.2. The Backend

Most of the work necessary for translating a Curry program to a new target language is already done by the *curry-frontend* and the transformations from the *icurry* package. The emitted *ICurry* format is designed to be structured imperatively and shall enable a sufficiently easy translation to the desired target language by requiring only slightly more logic than what is needed for a pretty printer. Furthermore, the *Extended ICurry* format executes some more transformations on top of the *ICurry* representation, which only make implementing the actual backend a little more straightforward.

B.2.1. (Extended) ICurry Structure

An (Extended) ICurry program consists of four parts: a module name, a list of imported modules, a list of declared data types and a list of declared functions.

Apart from the module name, all these pieces of information should be needed to emit a complete program. The module name, however, may be required for some target languages to help naming the module or single functions.

The import list is a comprehensive list of all modules used in this module. Every function or datatype that is referenced in this module is defined either in this module or in one of the modules in this list. Still, this list may contain unused modules, as there is currently no mechanism to prune imported but unused modules.

B.2.2. Data Types

Data type declarations have a name, a number of type variables and a number of constructors. Every data type shall be compiled into a generator function.

A generator function looks just like any other function and it is used exactly the same. Its arguments arise from the data type's type arguments. They are used to parameterize this generator by passing an appropriate generator for every type argument.

The generator never requires the head-normal form of any argument. It shall only set its node to a choice. This choice shall contain one constructor call for every constructor of this type. The constructors' arguments are determined as follows:

If a type variable is referenced in the constructor's signature, a function call to the `Prelude.unshare` function, with the variable as only argument, shall be passed. `unshare` will be explained in section B.4.1. If a type constructor is given instead, a function call to this data type's generator is inserted. Its arguments are found by doing this recursively.

B.2.3. Functions

A function is either compiled or defined externally. If a function is defined externally, the external implementation shall be called and no further action is necessary.

Translating a function body is the more interesting case. All specified arguments shall be unpacked from the node the function operates on. Names for variables in the target program can directly be derived from each `IVarIndex`, for example by converting the index to a string and prepending a letter. They are meant to be sufficiently unique.

Then, the block defines the actual logic happening in the function. All blocks carry a number of local variables and a number of assignments. Before dealing with the logic specific to each different block, all these local variables shall be declared and defined to an empty node. This is needed because these new variables can be used before they are assigned, for example to allow cyclic data structures. Then it shall process each assignment by assigning the structure arising from the given expression to the given variable.

A simple block shall now set the contents of the current node to the graph structure constructed from the expression.

A case block shall examine the specified variable. If it is not in head-normal form, the function aborts by returning the position of this variable in the function's argument list. The variable is guaranteed to always be a function argument. If it is in head-normal form, it shall find the correct branch based on the constructor in this variable. Then it shall process the block given in this branch.

For a case differentiation over literals it may happen that no branch matches. In this case the node shall be set to a failure.

For a differentiation over constructors of a data type, this can never happen. All cases resulting in a failure are given explicitly. As constructors can have arguments, these must be unpacked to the variables by their position just like function arguments, before the block specified in this branch is processed.

Evaluating expressions builds a subgraph. `ILit`, `IFCall`, `ICCall` and `IOr` (respectively their `IE`-counterparts) generate new nodes labeled accordingly. `IVar` (respectively `IEVar`) just gives an existing node which is referenced by a variable.

B.3. Input/Output

As the described method of evaluation is lazy, the order of evaluation of every subterm is not set by the order of their appearance in the source program. While this is not a problem for pure functions, IO requires the ability to specify the order of execution. This is solved using the `IO` monad, which enforces every action to be executed in order. To do this, an implicit *world* object is passed between IO actions that are being executed. This world could be seen as actually containing all data for input and output, allowing to see IO actions as pure functions for theoretical arguments. In practice, however, this world object contains much less, nothing in fact. It is a mere dummy for guarding the access to the actual world (the user, file system, network, etc.) and hence, can be represented by the smallest applicable type.

An action in the source language with type `IO a` can be implemented as a function `() -> (a,())`, with the world being represented through a unit type. It is important to only execute the action and produce a result, once the world argument actually is evaluated to head-normal form. This small detail ensures the correct evaluation order.

As IO actions are composed using `Prelude.>>=$ :: IO a -> (a -> IO b) -> IO b`, this function has to properly pass the world between the two actions. A usual implementation would be to pass the world to the first action, wait for its evaluation to head-normal form, extract the world from the action's result, pass this world to the second action and then make this second action's result the whole result.

Although at first glance this may look like an additional translation step, all basic IO actions are defined as external functions and thus are not translated at all. IO actions defined in the source program are always composed from these basic actions using the previously explained bind operator, which takes care of handling the world.

B.3.1. `catch` and the World

In practice, IO actions can throw errors. In the source language, these errors have to be handled using the function `catch :: IO a -> (IOError -> IO a) -> IO a`. To properly allow this, it may be customary to have the world slightly more complex than the unit type. An equivalent of `data IOWorld = WorldOK | WorldError IOError` allows carrying an additional error.

`>>=$` then has to immediately return this error when receiving a `WorldError` from the first action. `catch` basically does the opposite of `>>=$`: It immediately returns if no error is seen and prepares and starts the second action (the error handler) if it receives an error from the action.

B.3.2. Starting an IO Action

Starting an IO action is required if the main function is an IO action or when invoking `unsafePerformIO`. Then, the action shall be copied, this copy shall receive a fresh world object and afterwards it can be assigned to a node, ready to be evaluated. In the case of `unsafePerformIO`, the action may not be directly assigned to the node that previously contained the call to this unsafe function. Instead it must be wrapped in another function that will extract the actual result from the world structure. Copying the action is necessary, as an action may be shared and every call needs to attach its own world object. Modifying the shared action duplicates a world and thus loses the enforcement of evaluation order.

B.4. External Functions

Many functions that need a native implementation in the target language are not very complex. Just by looking at their signature and maybe their documentation, one should be able to immediately have an idea how to implement them.

As there are many externally defined functions in the *curry-base*, this section will only focus on a few more complicated functions.

Most externally defined functions do not need to handle unevaluated nodes. Small stubs in the curry libraries ensure that these external implementations are only called on nodes which have already been evaluated to (head-)normal form.

B.4.1. `Prelude.unshare :: a -> a`

`unshare` is the only function added by the translation to *ICurry*. As there is no external documentation for this function, it is included in this document.

Its purpose is to separate a choice node from all other nodes with its choice identity. This is necessary to enable proper semantics for choices in the presence of generators. Without splitting the identities between a passed generator and its use, non-deterministic branches

between *different* free variables would be shared. Splitting these identities reduces these sharings to exactly those expected by call-time semantics.

Usually, `unshare` will create a shallow copy of the node which is the only argument that `unshare` is called on. The copy will reference the same nodes as children and no children are changed or copied. `unshare` will reset the copy's choice tag (see B.1.1), so it will be given a new one on occasion. This copy is the result of this function.

B.4.2. `Prelude.apply :: (a -> b) -> a -> b` and similar

`apply` looks like the identity function on a functional type. It is used, however, to construct a (potentially partial) function/constructor call from a partial function/constructor call and an argument.

The partial application needs to be copied to a new node. This new node can now be modified by adding a new child reference to the new argument. Copying is necessary, as a partial application may be used multiple times with different arguments, so the original partial application must be preserved.

Several other functions use (partial) applications in their mechanism (for example `$!`, `$#`). They can either use the `apply` function or just replicate this process in place.

B.4.3. `(Prelude.=:=) :: a -> a -> Bool`

`(=:=)` does a few things at the same time. Not only does it compare two data terms, but it also takes shortcuts by binding free variables and thus avoids lots of useless work.

The comparison recursively descends both arguments. This, however, may not be done strictly in the function body of `(=:=)`, as one of the data terms may be infinite or may contain unevaluated nodes. Strictly evaluating an inifinite structure would lead to an infinite computation in this function. This would immediately break the completeness of the Fair Scheme. An unevaluated node cannot be compared but must be evaluated first. Since a function can only request to evaluate those nodes which are function arguments, this would be a problem.

Instead, this function only compares both data terms' constructors, failing on inequality, and then returns a new subgraph, which represents the conjunction of the structural equalities of each argument. This conjunction shall preferably be realized with a mechanism like the one used in the function `&`, so the comparison is more like a breadth-first search than a depth-first search.

Binding a free variable is unique to this function. If one of this function's arguments is a generator, the generator's node can unconditionally be updated so that it bears the same label and children as the other argument. This requires the ability to distinguish a simple *choice* from a *generator*, as generators are choices as well. In this, generators are special choices and could (only for use in this function) be marked to be suitable for unification.

B.4.4. `(Prelude.$!!) :: (a -> b) -> a -> b` and `toNF :: a -> a`

`($!!)` evaluates the argument to normal form before passing it to the function. This can easily be implemented by using a helper function `toNF`, which is not part of a Curry library. `f $!! x` then only has to request the evaluation of `toNF x` to head-normal form, which can easily be achieved by passing this term as an argument to a helper function, which then can request the evaluation to head-normal form.

`toNF` then has to ensure that it only ever produces a head-normal form, if the result is also in normal form. This is achieved by recursively using this function and waiting for *all* arguments to be processed before constructing the result. In other words, this function can be seen to emulate a strict identity function in a lazy runtime system.

B.4.5. `Prelude.readFile :: String -> IO String`

While `writeFile` is easy to implement, because it only receives completely evaluated arguments and may do all the work in a single go, `readFile` should read lazily.

This can be done by obtaining a handle from opening the file, and then returning an applied function call to a `read :: Handle -> String` function as IO result. This is still a correct implementation, as the mechanism of sharing avoids trying to read the same position multiple times, although the function modifies the handle and, thus, is not purely functional. `read` then can be implemented as the equivalent of

```
1 if handle is eof then
2 |   close handle;
3 |   return [];
4 else
5 |   c ← readChar handle;
6 |   return c : read handle;
```

close and *readChar* denote the native functions to close a file handle and to get the next character from an open file handle.

B.4.6. The `Global` Module

The module `Global` allows to define and allow constructs that behave like global variables in imperative languages. While their usage is safe as reading and writing are performed in IO actions, global variables are defined using the pure function `global`. This is a problem, because global definitions are never shared in Curry [Han16]. No naive implementation will allow effective writing to a global variable, as a new instance is created everytime the global variable is referenced, which makes it impossible to read a modified value. Also there is no argument which would allow the `global` function to distinguish between different global variables.

To solve this, the translation can introduce a special handling for functions which are a mere call to the `global` function. These functions shall not be translated to a single function, but rather into two parts. The first part is an explicit subgraph containing the starting value (or data read from persistent storage), saved in the equivalent of a global variable. The second part is a function which simply returns a reference to this global structure.

This way, every reference to the global variable's definition will eventually be evaluated to the same structure in memory. A write to this variable can then be reflected in later reads from this global variable.

B.4.7. `IO.Handle`

The `IO` module defines a data type `Handle`. It also introduces functions to access files and file-like objects making use of this data type.

Other modules, especially the `Socket` module, however, also use handles. Consequently the `IO` functions working on a `Handle` not only need to be able to access files, but also other resources which may require different low-level interfaces. Thus, the `Handle` likely needs a mechanism to dispatch the correct functions for dealing with a handle at runtime.

Literatur

[AAF+10] Abdulla Alqaddoumi, Sergio Antoy, Sebastian Fischer und Fabian Reck. „The pull-tab transformation“. In: *Proceedings of the Third International Workshop on Graph Computation Models*. Enschede, The Netherlands, 2010, S. 127–133.

[AH06] Sergio Antoy und Michael Hanus. „Overlapping Rules and Logic Variables in Functional Logic Programs“. In: *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. 2006, S. 87–101. DOI: 10.1007/11799573_9.

[AHT18] Sergio Antoy, Michael Hanus und Finn Teegen. „Synthesizing Set Functions“. In: *Proceedings of the 26th International Workshop on Functional and (constraint) Logic Programming (WFLP 2018)*. 2018. arXiv: 1808.07401. URL: http://arxiv.org/abs/1808.07401.

[AJ13] Sergio Antoy und Andy Jost. „Compiling a Functional Logic Language: The Fair Scheme“. In: *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers*. 2013, S. 202–219. DOI: 10.1007/978-3-319-14125-1_12.

[AJ16] Sergio Antoy und Andy Jost. „A New Functional-Logic Compiler for Curry: Sprite“. In: *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*. 2016, S. 97–113. DOI: 10.1007/978-3-319-63139-4_6.

[BHP+11] Bernd Braßel, Michael Hanus, Björn Peemöller und Fabian Reck. „KiCS2: A New Compiler from Curry to Haskell“. In: *Functional and Constraint Logic Programming - 20th International Workshop, WFLP 2011, Odense, Denmark, July 19th, Proceedings*. 2011, S. 1–18. DOI: 10.1007/978-3-642-22531-4_1.

[BW88] Hans-Juergen Boehm und Mark Weiser. „Garbage Collection in an Uncooperative Environment“. In: *Softw., Pract. Exper.* 18.9 (1988), S. 807–820. DOI: 10.1002/spe.4380180902. URL: https://doi.org/10.1002/spe.4380180902.

[C99] *Programming Languages – C*. Standard ISO 9899:1999. Dez. 1999. URL: www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[EJ97] Rachid Echahed und Jean-Christophe Janodet. „On Constructor-based Graph Rewriting Systems“. In: *RESEARCH REPORT 985-I, IMAG*. 1997.

[Han13] Michael Hanus. „Adding Plural Arguments to Curry Programs“. In: *TPLP* 13.4-5-Online-Supplement (2013). URL: http://static.cambridge.org/resource/id/urn:cambridge.org:id:binary:20161018085635834-0697:S1471068413000112:tlp2013018.pdf.

[Han16] Michael Hanus (Hrsg.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Verfügbar unter http://www.curry-language.org, zuletzt abgerufen: 2018-09-16. 2016.

[Han18] Michael Hanus (Hrsg.) *The Portland Aachen Kiel Curry System - User Manual*. 2018-08-16. 2018.

[Kir17] Bastian Kirchmayr. „Übersetzung von Curry nach Java“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Juni 2017.

[Obe16] Jonas Oberschweiber. „A Package Manager for Curry“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Sep. 2016.

[Sik17] Jasper Paul Sikorra. „Integration von Curry-Programmen in Webseiten durch Übersetzung nach JavaScript“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Aug. 2017.

[Tee16] Finn Teegen. „Erweiterung von Curry um Typklassen und Typkonstruktorklassen“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Sep. 2016.