Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Biologie und Informatik
Institut für Informatik

Diplomarbeit

Realisierung der Ein-/Ausgabe in einem Compiler für Haskell bei Verwendung einer nichtdeterministischen Semantik

David Sabel

15. September 2003

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie

Danksagung

Ich möchte mich hiermit bei allen bedanken, die mich während der Entstehung dieser Arbeit begleitet und unterstützt haben.

Mein besonderer Dank gilt Matthias Mann und Prof. Dr. Manfred Schmidt-Schauß für Ihre hervorragende Betreuung und Ihre unzähligen hilfreichen Anregungen.

Außerdem möchte ich mich bei den Mitgliedern der „Glasgow Haskell Users Mailing List“ und den Entwicklern des Glasgow Haskell Compiler für die nützlichen Antworten auf meine Fragen bedanken.

David Sabel

Erklärung gemäß DPO §11 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Offenbach am Main, den 15. September 2003

D a v i d S a b e l

Inhaltsverzeichnis

Abkürzungsverzeichnis

bzgl.	bezüglich
CSE	Common Subexpression Elimination
Forts.	Fortsetzung
gdw.	genau dann wenn
GHC	Glasgow Haskell Compiler
HasFuse	Haskell with FUNDIO-based side effects
IO	Input and Output
O.B.d.A.	Ohne Beschränkung der Allgemeinheit
STG	Shared Term Graph
WHNF	Weak Head Normal Form

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Funktionale Programmiersprachen weisen viele Eigenschaften auf, die zur modernen Softwareentwicklung benötigt werden: *Zuverlässigkeit*, *Modularisierung*, *Wiederverwendbarkeit* und *Verifizierbarkeit*.

Als schwer vereinbar mit diesen Sprachen stellte sich die Einbettung von Seiteneffekten in diese Sprachen heraus. Nach einigen mehr oder weniger gescheiterten Ansätzen[1] hat sich mittlerweile für die nichtstrikte funktionale Sprache Haskell der monadische Ansatz durchgesetzt, bei dem die Seiteneffekte geschickt verpackt werden, so dass zumindest IO-behaftete Teile eines Haskell-Programms dem klassischen imperativen Programmierstil ähneln.

S. Peyton Jones bringt dieses in [Pey01, Seite 3] auf den Punkt, indem er schreibt

> „...Haskell is the world's finest imperative programming language."

Dies ist einerseits vorteilhaft, denn die klassischen Programmiertechniken können angewendet werden, andererseits bedeutet dies auch eine Rückkehr zu altbekannten Problemen in diesen Sprachen: Der Programmcode wird unverständlicher, die Wiederverwendbarkeit von Code verschlechtert sich, Änderungen im Programm sind aufwendig. Zudem erscheint die monadische Programmierung teilweise umständlich.

Deshalb wurde im Zuge der Entwicklung in fast jede Implementierung von Haskell ein Konstrukt eingebaut, dass von monadischem IO zu direktem IO führt. Da dieses nicht mit dem bisherigen Ansatz vereinbar schien, wird es als „unsafe" bezeichnet, die entsprechende Funktion heißt `unsafePerformIO`. Zahlreiche Anwendungen benutzten diese Funktion, teilweise scheint die Verwendung zumindest aus Effizienzgründen unabdingbar[2] .

[1] Einige davon werden in [HS89] dargestellt.

[2] `unsafePerformIO` wird beispielsweise in [PME99] als einer von vier Mechanismen zur effizienten Implementierung von so genannten „weak pointers" benötigt.

Allerdings ist die Frage, wann dieses verwendet werden darf, d.h. so angewendet wird, dass es nicht „unsicher“ ist, nur unzureichend geklärt. Das Zitat in Abbildung 1.1 gibt wenig Aufschluss über die korrekte Anwendung, zumal immer wieder Diskussionen entstehen, ob die Verwendung von `unsafePerformIO` in diesem oder jenem Spezialfall korrekt ist[3].

> When is it safe to use `unsafePerformIO`?
>
> > We'll give two answers to this question, each of which may be helpful. These criteria are not rigorous in any real sense (you'd need a formal semantics for Haskell in order to give a proper answer to this question), but should give you a feel for the kind of things you can and cannot do with `unsafePerformIO`.
> >
> > - It is safe to implement a function or API using `unsafePerformIO` if you could imagine also implementing the same function or API in Haskell without using `unsafePerformIO` (forget about efficiency, just consider the semantics).
> > - In pure Haskell, the value of a function depends only on the values of its arguments (and free variables, if it has any). If you can implement the function using `unsafePerformIO` and still retain this invariant, then you're probably using `unsafePerformIO` in a safe way. Note that you need only consider the observable values of the arguments and result.
>
> [...]

Abbildung 1.1: Ausschnitt aus [The03, Chapter 13]

1.2 Zielsetzung

Ziel dieser Arbeit ist es, einen Prototypen zu entwickeln, der es ermöglicht um `unsafePerformIO` erweiterte Haskell-Programme zu compilieren und dabei Korrektheit bezüglich einer festgelegten Semantik zu garantieren. Die Verwendung von `unsafePerformIO` soll dabei uneingeschränkt möglich sein und sich nicht, wie bisher, auf Spezialfälle konzentrieren, um somit die Möglichkeit zu schaffen Haskell-Programme mit nicht-monadischem deklarativem IO zu definieren.

Hierfür dient als theoretische Grundlage der FUNDIO-Kalkül aus [Sch03a], der direktes IO mittels Nichtdeterminismus simuliert, die praktische Umsetzung soll durch Modifikation des Glasgow Haskell Compiler (GHC) [PHH+93, The03], der bereits

[3] Zahlreiche solcher Diskussionen findet man im Archiv der Haskell-Mailinglist unter `http://haskell.org/mailman/listinfo/haskell`

das `unsafePerformIO`-Konstrukt anbietet, derart erfolgen, so dass während des Übersetzungsvorgangs nur solche Programmtransformationen stattfinden, die der Semantik des FUNDIO-Kalküls entsprechen.

1.3 Überblick

Im Folgenden geben wir einen Überblick über die weiteren Kapitel.

In *Kapitel 2* stellen wir die Grundlagen funktionalen Programmierens dar und geben einen kurzen Überblick über die Programmiersprache Haskell und den GHC. Danach gehen wir auf monadisches IO und `unsafePerformIO` in Haskell und der entsprechenden Implementierung im GHC ein. Das Kapitel endet mit einem Beispiel zur Problematik der bisherigen Verwendung von `unsafePerformIO`.

Kapitel 3 beginnt mit der Definition des FUNDIO-Kalküls aus [Sch03a], danach werden die wichtigsten Ergebnisse aus [Sch03a] dargestellt. Anschließend definieren wir einige neue Programmtransformationen für den FUNDIO-Kalkül, die für spätere Beweise benötigt werden, und beweisen deren Korrektheit.

Kapitel 4 stellt den Zusammenhang zwischen dem GHC und dem FUNDIO-Kalkül her, indem wir eine Übersetzung definieren, die Ausdrücke einer Variante der Compiler-internen Kernsprache in Ausdrücke des FUNDIO-Kalküls übersetzt. Mithilfe dieser Übersetzung wird im Anschluss ein Großteil der im GHC angewendeten Programmtransformationen auf Korrektheit bzgl. der FUNDIO-Semantik geprüft.

In *Kapitel 5* wird die praktische Umsetzung der Ergebnisse aus Kapitel 4 beschrieben, indem die vorgenommenen Modifikationen des GHC kurz dargestellt werden.

In *Kapitel 6* wird eine Zusammenfassung der Arbeit sowie ein Ausblick auf mögliche weitere Untersuchungen bzgl. des in der Arbeit behandelten Themas gegeben.

Kapitel 2

Grundlagen

2.1 Funktionales Programmieren

In diesem Abschnitt wird – in ähnlicher Weise wie in [Sch00] und [Sch03b] – eine kurze Einführung in funktionales Programmieren und die dazu gehörigen Konzepte gegeben.

Funktionales Programmieren trägt diese Bezeichnung, da Programme in den zugehörigen Sprachen im Wesentlichen aus *Funktionen* und Anwendung von Argumenten auf diese bestehen.

Eine Funktion K_and_double, die zwei Argumente erwartet und als Ergebnis das erste Argument verdoppelt zurück liefert, kann z.B. folgendermaßen definiert werden

$$K\_and\_double(x, y) = x + x$$

Hierbei wird im Folgenden die rechte Seite einer solchen *Funktionsdefinition* als *Funktionsrumpf* bezeichnet.

Man kann nun den Wert von $K\_and\_double(3, 5)$ berechnen (was wir im Folgenden auch als *Auswerten* des *Ausdrucks* $K\_and\_double(3, 5)$ bezeichnen), indem der Parameter x durch 3 und y durch 5 ersetzt wird, also etwa in folgender Weise:

$$K\_and\_double(3, 5) \rightarrow 3 + 3 \rightarrow 6$$

Während beim imperativen Programmieren Probleme gelöst werden, indem die zugehörige Berechnung als Folge von Anweisungen dargestellt wird, wird in funktionalen Programmiersprachen von der Definition der Berechnungsschritte abstrahiert und das erwartete Ergebnis in deklarativem Stil mithilfe von Funktionen spezifiziert, wobei meist eine Zerlegung des eigentlichen Problems in kleinere Teile notwendig ist. Dieses wesentliche Konzept und dessen Vorteile werden in [Hug89] gut dargestellt.

Die verschiedenen Implementierungen von funktionalen Programmiersprachen unterscheiden sich vor allem in der *Auswertungsstrategie* und darin, ob sie *pure* oder *impure* funktionale Programmiersprachen sind. Diese Begriffe werden im Folgenden erklärt.

2.1.1 Auswertungsstrategien

Die „call-by-value“-Strategie

Diese Auswertungsstrategie verlangt, dass Argumente erst dann in den Funktionsrumpf eingesetzt werden dürfen, wenn sie Werte[1] sind, d.h. vor der Einsetzung müssen die Argumente ausgewertet werden. So wird beispielsweise der Ausdruck $K\_and\_double(4-1, 2+2)$ wie folgt ausgewertet:

$$\begin{aligned} & K\_and\_double(4-1, 2+2) \\ \rightarrow \quad & K\_and\_double(3, 2+2) \\ \rightarrow \quad & K\_and\_double(3, 4) \\ \rightarrow \quad & 3+3 \\ \rightarrow \quad & 6 \end{aligned}$$

Diese Art der Auswertung nennt man auch *applikative Reihenfolge*. Programmiersprachen, die diese Strategie benutzen, werden auch als *strikt* bezeichnet.

Der Nachteil dieser Auswertung wird an obigem Beispiel deutlich: Das zweite Argument wird ausgewertet, obwohl es im Funktionsrumpf nicht benutzt wird.

Insbesondere führt dies dazu, dass die Berechnung des gesamten Ausdrucks nicht terminiert, wenn die Auswertung des zweiten Arguments nichtterminierend ist.

Die „call-by-name“-Strategie

Die „call-by-name“-Strategie setzt die Argumente in den Funktionsrumpf ein, ohne diese vorher auszuwerten und berechnet dann den Wert des Gesamtausdrucks. Eine Auswertung des obigen Beispiels hat dann die Form:

$$\begin{aligned} & K\_and\_double(4-1, 2+2) \\ \rightarrow \quad & (4-1)+(4-1) \\ \rightarrow \quad & 3+(4-1) \\ \rightarrow \quad & 3+3 \\ \rightarrow \quad & 6 \end{aligned}$$

Beim Benutzen dieser Strategie wird das zweite Argument $(2+2)$ nie ausgewertet, da es für die Berechnung des Ergebnisses nicht benötigt wird.

Ebenso terminiert die Auswertung des Ausdrucks $K\_and\_double(4-1, \bot)$, wobei $\bot$ ein Ausdruck sei, dessen Auswertung nicht terminiert.

[1] Genauer wäre hier der Begriff „strikte Normalformen“, der beispielsweise in [Sch03b, Definition 5.4.33] definiert wird.

Die „call-by-name“-Strategie hat jedoch den Nachteil, dass u.U. Ausdrücke mehrfach ausgewertet werden. So wird im Beispiel der Ausdruck $(4-1)$ zweimal ausgewertet, während bei der „call-by-value“-Strategie dieser Ausdruck nur einmal ausgewertet wurde.

Funktionale Sprachen die die „call-by-name“-Strategie verwenden, werden als *nicht-strikt* bezeichnet.

Die „call-by-need“-Strategie

Die „call-by-need“-Strategie kombiniert die Vorteile der beiden vorher genannten Strategien und umgeht deren Nachteile.

Bei ihr werden, ebenso wie bei der „call-by-name“-Strategie, die Argumente ohne vorheriges Auswerten in den Funktionsrumpf eingesetzt, wobei jedoch Mehrfachauswertungen mithilfe von *Sharing* vermieden werden. Intuitiv kann man sich vorstellen, dass die Auswertungsmaschine die Argumente markiert und bei der ersten Auswertung alle gleich markierten Ausdrücke durch das Ergebnis ersetzt. Am Beispiel kann man das wie folgt darstellen, wobei $M1$ und $M2$ Markierungen seien:

$$\begin{aligned} & K\_and\_double((4-1)^{M1}, (2+2)^{M2}) \\ \rightarrow \quad & (4-1)^{M1} + (4-1)^{M1} \\ \rightarrow \quad & 3+3 \\ \rightarrow \quad & 6 \end{aligned}$$

Die Implementierung dieser Strategie erfolgt, indem Ausdrücke als Graphen – im Gegensatz zu Bäumen – repräsentiert werden. In Abbildung 2.1 (a) wird der Funktionsrumpf von *K_and_double* als Baum, in Abbildung 2.1 (b) als Graph, dargestellt[2], wobei Knoten mit der Markierung @ Anwendungen repräsentieren.

Abbildung 2.1 (c) stellt die „call-by-name“-Auswertung und Abbildung 2.1 (d) die „call-by-need“-Auswertung des Beispiels nach Einsetzung in den Funktionsrumpf von *K_and_double* dar.

Funktionale Programmiersprachen, die die „call-by-need“-Strategie benutzen, werden auch als *lazy* bezeichnet.

Allgemein gilt, dass die „call-by-name“-Strategie terminiert, wenn ein Wert existiert, und die „call-by-need“-Strategie darüber hinaus höchstens so viele Auswertungsschritte wie die „call-by-value“-Strategie benötigt.

2.1.2 Pure und impure funktionale Programmiersprachen

Pure funktionale Programmiersprachen bauen auf dem klassischen Lambda-Kalkül auf, wie er in [Bar84] definiert wird. Ein wichtiger Aspekt ist hierbei, dass für pure

[2]Die Darstellung ist analog zu [Sch01].

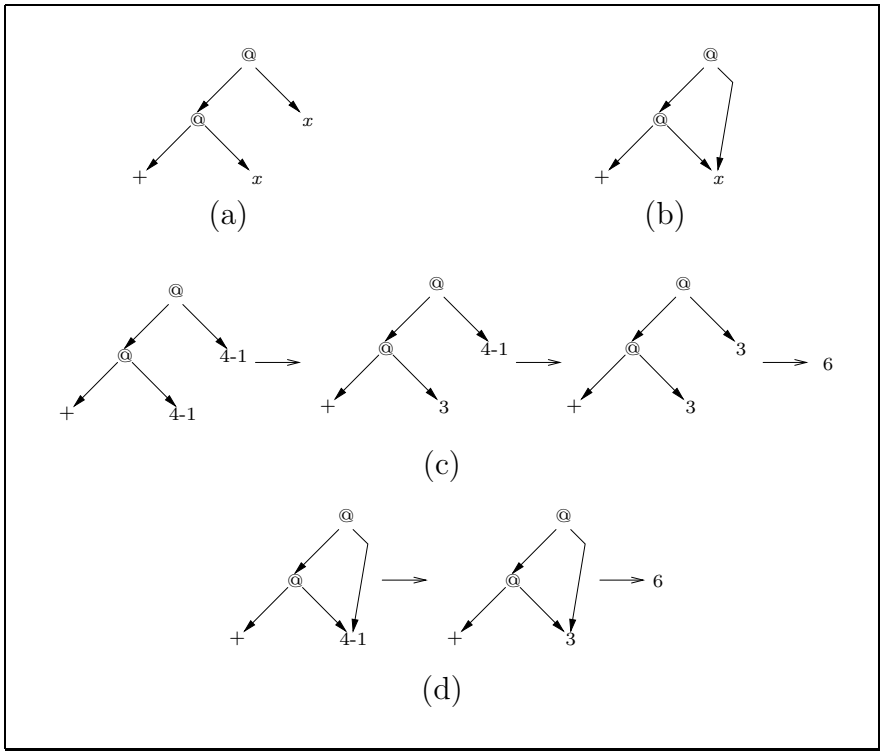

Abbildung 2.1: Darstellung von Ausdrücken als Bäume und Graphen

funktionale Programmiersprachen das Prinzip der *referentiellen Transparenz* gilt, das analog zu [Sch03b, Kapitel 1, Seite 6] wie folgt formuliert werden kann: Der Wert einer Anwendung von Argumenten auf eine (feste) Funktion hängt ausschließlich vom Wert der Argumente ab, d.h. insbesondere ergibt eine Anwendung von gleichen Argumenten auf eine gleiche Funktion immer den gleichen Wert. [BW88, Seite 4] definiert selbiges aus Sicht der Termstruktur, indem er schreibt, dass der Wert eines Ausdrucks einzig von den Werten seiner Teilausdrücke abhängt.

Wenn referentielle Transparenz gegeben ist, besteht der Vorteil, dass beliebige Teilausdrücke durch Ausdrücke mit gleichem Wert ersetzt werden können. Pure Sprachen benötigen besondere Vorgehensweisen (eine solche wird in Abschnitt 2.4.1 vorgestellt) um Seiteneffekte zu modellieren, da diese die referentielle Transparenz nicht verletzen dürfen.

Impure funktionale Programmiersprachen erlauben (eingeschränkt) Seiteneffekte und können somit einfacher Ein- und Ausgabe und andere Seiteneffekt-behaftete Operationen darstellen. Allerdings ist die Beweisbarkeit von Aussagen bei Sprachen mit Seiteneffekten schwieriger als bei puren funktionalen Programmiersprachen.

2.2 Haskell

Haskell (definiert in [ABB+99]) ist eine pure nicht-strikte funktionale Programmiersprache, die statisch polymorph getypt ist. Im Folgenden werden wesentliche Konzepte dieser Sprache vorgestellt, da genau diese Sprache um direkt aufgerufene Ein-/Ausgabe-Operatoren erweitert werden soll.

2.2.1 Programme

Ein Haskell-Programm besteht aus einem oder mehreren *Modulen* (siehe [ABB+99, Kapitel 5]), wobei ein solches jeweils innerhalb einer Datei definiert wird. Ein Modul muss den Namen `Main` tragen und den Wert (bzw. die Funktion) `main` exportieren. Dieser Wert ist der Wert des Haskell-Programms und er wird bei Ausführung berechnet. Dieses Hauptmodul hat somit die Form

```
module Main(main) where
 ...
 main = ...
 ...
```

Module dienen zur Strukturierung einzelner Teile eines Programms und zur einfachen Wiederverwendung von Code.

Innerhalb eines Moduls werden neben dem Import anderer Module (mit dem Schlüsselwort `import`) Datentypen sowie Funktionen definiert.

2.2.2 Datentypen

Zur Deklaration neuer Datentypen stellt Haskell das `data`-Konstrukt (siehe [ABB+99, Abschnitt 4.2]) zur Verfügung, mit dem z.B. ein Datentyp für die drei Grundfarben der additiven Farbmischung in folgender Form definiert werden kann:

```
data Farbe = Rot | Gruen | Blau
```

Hierbei ist `Farbe` der neue Datentyp; `Rot`, `Gruen` und `Blau` sind die Datenkonstruktoren.

Ferner stellt Haskell *Typklassen* zur Verfügung, die insbesondere dazu dienen Operatoren zu überladen.

So ist z.B. für alle Datentypen, die Instanzen der Klasse `Eq` sind, der Gleichheitsoperator `(==)` definiert. Eine Instanz der Klasse `Eq` für den oben definierten Datentyp für Farben kann wie folgt definiert werden:

```
instance Eq Farbe where
    (==) Rot Rot     = True
    (==) Gruen Gruen = True
    (==) Blau Blau   = True
    (==) _   _       = False
```

Mithilfe des Schlüsselworts `type` lassen sich Typsynonyme definieren, z.B. könnte man damit einen Typ `Farbenpaar` definieren, der ein Typsynonym für den Typ `(Farbe,Farbe)` sei:

```
type Farbenpaar = (Farbe, Farbe)
```

Für solche Typsynonyme können keine Instanzen für Typklassen definiert werden. Will man dies erreichen, so muss man das Schlüsselwort `newtype` benutzen, dass einen neuen Datentyp definiert, indem eine Kopie des existierenden Typs benutzt wird. Die Definition mit `newtype` verlangt zusätzlich einen neuen Datenkonstruktor. Für das Farbenpaar könnte man

```
newtype Farbenpaar = FPaar (Farbe, Farbe)
```

definieren, wobei `FPaar` der neue Datenkonstruktor ist.

2.2.3 Pattern matching

Im Beispiel für die `Eq`-Instanz wurde bereits *Pattern matching*[3] (siehe [ABB+99, Abschnitt 3.17]) benutzt. Hierbei wurden in jeder Zeile der Definition für `(==)` unterschiedliche Muster bzw. *Pattern* angegeben. Die erste Zeile verlangt für beide Argumente das Muster `Rot`. Nur wenn die Funktion (==) mit Argumenten aufgerufen wird, die mit dem Muster übereinstimmen (das Pattern „matcht“ dann), ist diese Zeile gültig für den Aufruf und der Ausdruck `(==) Rot Rot` wird durch den Ausdruck `True` ersetzt. Die letzte Zeile der Definition enthält für beide Argumente von `(==)` das Pattern `_`, das auch als *Wildcard* bezeichnet wird. Dieses Pattern matcht immer.

Pattern matching ist jedoch weitaus mächtiger, als es im obigen Beispiel gezeigt wird, da auch geschachtelte Pattern möglich sind.

Listen sind in Haskell als Datentyp mit dem nullstelligen Konstruktor `[]`[4] für das Listenende, sowie dem zweistelligen Konstruktor `(:)`[5] zur Konstruktion von Listen definiert. Z.B. kann die Liste bestehend aus den Zahlen 1,2 und 3 in Haskell als `1:(2:(3:[]))` oder in kürzerer Syntax als `[1,2,3]` dargestellt werden. Eine Funktion, die das erste Element der ersten Liste einer Liste von Listen berechnet, kann mithilfe von Pattern matching wie folgt definiert werden:

[3] engl. für Mustererkennung
[4] gesprochen als „Nil“
[5] gesprochen als „Cons“

```
doubleHead :: [[a]] -> a
doubleHead ((a:as):xs) = a
```

Beim Aufruf von `doubleHead [[10,11],[20]]` wird `(10:(11:[])):((20:[]):[])` gegen das Pattern `((a:as):xs)` gematcht, so dass an die Variable `xs` der Ausdruck `((20:[]):[])`, an `as` der Ausdruck `(11:[])` und an `a` der Ausdruck `10` gebunden wird. Diese Bindungen werden dann für die Auswertung des Funktionsrumpfs verwendet.

Das Pattern matching kann soweit aufgelöst werden, so dass der Ausdruck nur mithilfe von `case`-Ausdrücken und einfachen (nicht verschachtelten) Pattern dargestellt wird. Obige Definition hat dann die Form:

```
doubleHead ys = case ys of
                  (x:xs) -> case x of
                              (a:as) -> a
```

2.2.4 Guards

Guards[6] sind Prädikate über den Argumenten einer Funktion, wobei für jeden Guard ein eigener Funktionsrumpf definiert werden muss. Bei der Auswertung der Funktion wird der Rumpf gewählt dessen Prädikat zuerst erfüllt ist, wobei die Abarbeitung der Guards sequentiell von oben nach unten erfolgt.

Eine Funktion, die prüft ob eines der beiden ersten Elemente einer Liste größer als 10 ist, kann mit Guards folgendermaßen definiert werden.

```
fstOrSndGreater10 (x1:(x2:_))
               | x1 > 10   = True
               | x2 > 10   = True
               | otherwise = False
```

Der Ausdruck `otherwise` ist in Haskell vordefiniert als

```
otherwise = True
```

und somit ist dieser Guard immer wahr.

Auf die Verwendung von Guards kann verzichtet werden, indem man verschachtelte `if-then-else`- Konstrukte benutzt. Die Definition von `fstOrSndGreater10` hätte dann die folgende Form:

```
fstOrSndGreater10 (x1:(x2:_)) = if x1 > 10 then True
                                else if x2 > 10 then True
                                     else False
```

[6] engl. für Wächter

2.2.5 List comprehensions

List comprehensions dienen zur einfachen Definition komplexer Listen:

Nach [ABB+99, Abschnitt 3.11] ist eine List comprehension eine Liste der Form `[`e `|` q_1`,...,` q_n`]`, wobei jedes der q_i entweder

- ein *Generator* der Form p `<-` e_i ist, wobei p ein Pattern vom Typ t und e_i eine Liste vom Typ `[`t`]`[7] ist,
- ein Guard ist oder
- eine lokale Definition ist, die in e oder nachfolgenden Guards oder Generatoren benutzt werden kann.

Mit List comprehensions kann z.B. eine Liste, die alle Elemente des kartesischen Produktes $\mathbb{N} \times \mathbb{N}$ der natürlichen Zahlen enthält, definiert werden:

```
[(x,y) | x <- [1..], y <- [1..]]
```

2.3 Der Glasgow Haskell Compiler

In diesem Abschnitt wird ein Überblick über den Glasgow Haskell Compiler ([PHH+93]) gegeben und anschließend genauer auf die Kernsprache des Compilers eingegangen, da auf dieser Sprache Programmtransformationen durchgeführt werden, die wir später auf Korrektheit bezüglich des FUNDIO-Kalküls überprüfen werden.

2.3.1 Compilerphasen des GHC

Gegeben sei Haskell-Code, wir beschreiben den Weg, den der Code durch den Compiler nimmt, wie in [PS98, PHH+93, CFM+02] dargestellt. Abbildung 2.2 zeigt die einzelnen Phasen und die zugehörige Ausgabe der jeweiligen Phase.

1. **Front-End**:
   - Nach der lexikalischen Analyse mittels des *Lexers* wird die Syntax des Quellcodes analysiert. Der dafür verwendete *Parser* ist mittels einer Parserspezifikation für den Parser-Generator Happy (siehe [MG01]) definiert. Die Ausgabe dieser Phase ist ein Syntaxbaum, der sämtliche Konstrukte von Haskell explizit darstellt, ohne Vereinfachungen vorzunehmen.

[7] In Haskell ist `[]` der Typkonstruktor für Listen.

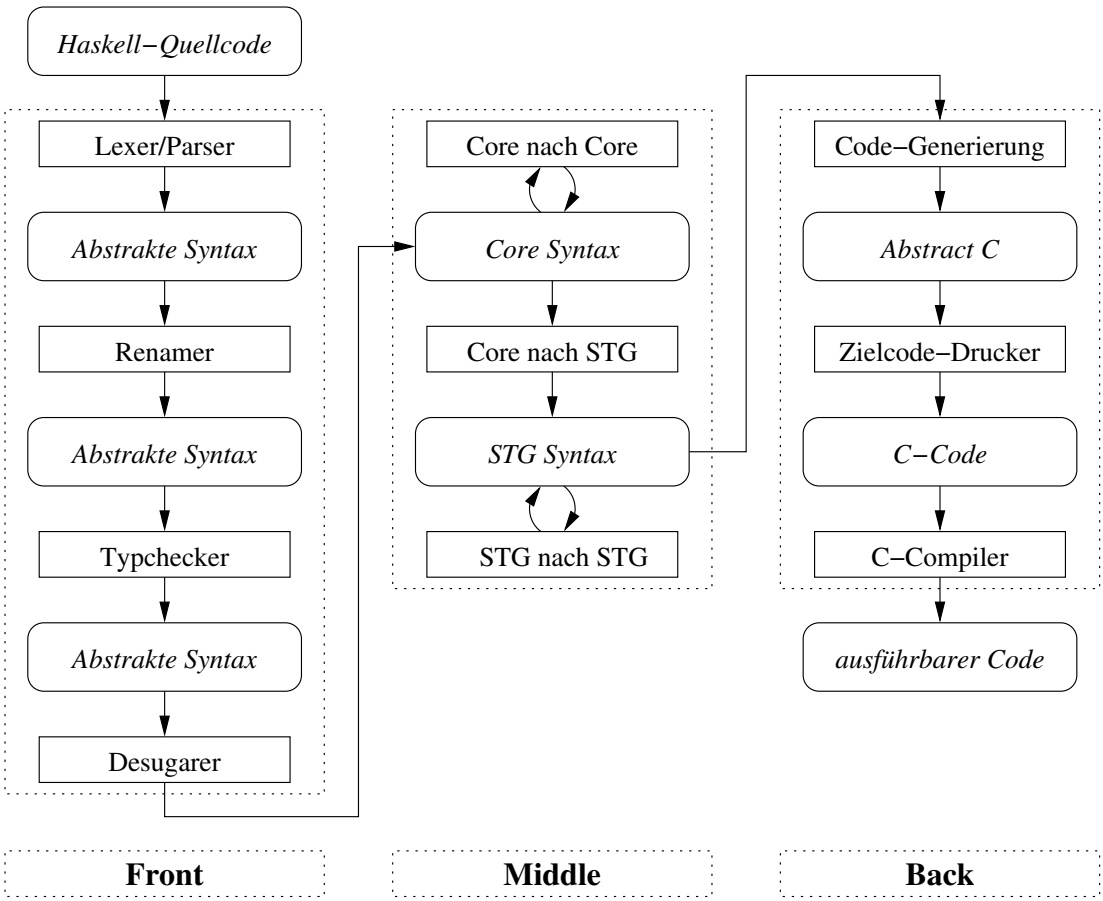

Abbildung 2.2: Phasen des GHC

- Der *Renamer* löst Bereichs- und Namenskonflikte insbesondere bezüglich des Modulimports und -exports auf.
- Der *Typchecker* fügt dem Programm Typinformationen hinzu, führt eine Typüberprüfung durch und löst die Typüberladung auf. Genauer wird dies in [WB89] erklärt.
- Der *Desugarer* übersetzt den Haskell-Code in eine einfachere funktionale Sprache, die als GHC-Kernsprache bezeichnet wird. Insbesondere werden syntaktische Spezialitäten wie „Pattern matching“ und „List comprehensions“ in einfachere Ausdrücke übersetzt [8].

2. Middle:

- Hier finden Transformationen auf der GHC-Kernsprache statt, die der Optimierung dienen, wobei je nach Optimierungsstufe unterschiedlich viele

[8] Solche Konstrukte einer höheren Programmiersprache werden häufig als „syntaktischer Zucker“ bezeichnet, woher sich der Name des „Entzuckerers“ ergibt.

Programmtransformationen durchgeführt werden.

- Übersetzung der GHC-Kernsprache in die STG[9]-Sprache, die eine noch einfachere funktionale Sprache darstellt als die Kernsprache des GHC.
- Optimierungstransformationen auf der STG-Sprache.

3. Back-End:

- Die Codegenerierung basiert auf der STG-Maschine (siehe [Pey92]), welche eine abstrakte Maschine ist. Der Zustand der Maschine besteht aus sechs Komponenten (siehe [Pey92, Seite 32]): Drei Stacks, der Code für die Durchführung des nächsten Schrittes der Maschine, einer globalen Umgebung und dem Heap. Der Heap stellt eine Abbildung von Adressen zu Heap-Zellen dar. In diesen Heap-Zellen werden so genannte „closures“ abgelegt. Diese stellen entweder Daten oder Funktionen dar, wobei die Repräsentation einheitlich für beide Objekte ist[10].
- Der *Codegenerator* erstellt nun „Abstract C“-Code, was nichts als ein interner Datentyp ist, der C-Code repräsentiert.
- Mittels des *Zielcode-Druckers* wird echter C-Code ausgegeben. Optional kann auch direkt Maschinencode erzeugt werden, wenn die jeweilige Maschinenarchitektur unterstützt wird.
- Dieser C-Code wird schließlich mit einem C-Compiler zu ausführbarem Code übersetzt.

Im Folgenden treffen wir die Annahme, dass das Back-End des GHC in dem Sinne korrekt arbeitet, dass die STG-Maschine eine Reduktionsmaschine ist, die ausschließlich in Normalordnung reduziert und dabei die call-by-need-Strategie verwendet. Deshalb werden diesen Teil des Compilers aus weiteren Untersuchungen ausklammern.

2.3.2 Die Kernsprache des GHC

Definition 2.1. ($L_{GHCCore}$)

In Abbildung 2.3 wird die Sprache $L_{GHCCore}$ definiert, die wir im Folgenden auch als GHC-Kernsprache *bezeichnen.*

Hierbei sind fett gedruckte Symbole Nichtterminale deren Definition angegeben ist, kursive Symbole sind sonstige Nichtterminale und die restlichen Symbole sind Terminale. Ein Programm wird mit dem Startsymbol **Prog** *abgeleitet.*

Zusätzlich zur dargestellten Grammatik gelten folgende Regeln und Konventionen:

[9] Die Abkürzung STG wurde zunächst für „Spinless Tagless G-Maschine“ benutzt, mittlerweile steht sie für „Shared Term Graph“, da sie unabhängig vom benutzten Maschinenmodell ist.

[10] Daher der Name „tagless“, da keine Markierung notwendig ist, um Daten von Funktionen zu unterscheiden.

Prog	::=	**Binding**$_1$; ...; **Binding**$_n$	$n \geq 1$
Binding	::=	**Bind**	
	|	`rec` **Bind**$_1$; ...; **Bind**$_n$	
Bind	::=	*Var* `=` **Expr**	
Expr	::=	**Expr Expr**	(Applikation)
	|	λ *Var*$_1$ ... *Var*$_n$ `->` **Expr**	(Lambda-Abstraktion)
	|	`case` **Expr** `of` **Alts**	(case-Ausdruck)
	|	`let` **Binding** `in` **Expr**	(lokale Definition)
	|	*Var*	(Variable)
	|	*Con*	(Konstruktor)
	|	**Literal**	(unboxed Objekt)
	|	*Prim*	(primitiver Operator)
Literal	::=	*Int* | *Char* | ...	
Alts	::=	**Calt**$_1$; ...; **Calt**$_n$; [**Default**]	$n \geq 0$
	|	**Lalt**$_1$; ...; **Lalt**$_n$; [**Default**]	$n \geq 0$
Calt	::=	*Con Var*$_1$ ... *Var*$_n$ `->` **Expr**	$n \geq 0$
Lalt	::=	**Literal** `->` **Expr**	
Default	::=	*Var* `->` **Expr**	

Abbildung 2.3: $L_{GHCCore}$ - Die Kernsprache des GHC

- *Klammern werden benutzt, um Zweideutigkeiten zu vermeiden.*
- *Die Applikation ist links assoziativ und bindet stärker als jeder andere Operator.*
  Beispiele:
  - $(a_1\ a_2 \ldots\ a_n)$ *ist eine Abkürzung für* $(\ldots((a_1\ a_2)\ldots)\ a_n)$.
  - *Der Ausdruck* $a_1\ a_2 + a_3$ *ist gleich zum Ausdruck* $(a_1\ a_2) + a_3$ *im Gegensatz zu dem Ausdruck* $a_1\ (a_2 + a_3)$.
- *Der Rumpf eines* λ*-Ausdrucks geht soweit wie möglich. So ist z.B. der Ausdruck* $\lambda x.\ e\ f$ *eine Abkürzung für* $\lambda x.\ (e\ f)$*, im Gegensatz zu* $(\lambda x.\ e)\ f$.
- *Ein gültiges Programm besitzt eine Bindung auf oberer Ebene, deren linke Seite den Namen* `main` *hat.*

- *Konstruktoranwendungen oder Anwendungen auf primitive Operatoren müssen nicht gesättigt sein, allerdings darf die Anzahl der Argumente die Stelligkeit nicht überschreiten. Innerhalb von Pattern in* `case`*-Alternativen muss jedoch die Anzahl der Argumente der Stelligkeit des Konstruktors entsprechen.*

- *Die* `case`*-Alternativen sind derart, dass alle Konstruktoren (des zugehörigen Typs), zu denen das erste Argument des* `case`*-Ausdrucks auswerten kann, abgedeckt sind. Dafür ist es u.U. nicht notwendig, dass für jeden Konstruktor des Typs eine eigene Alternative oder eine default-Alternative vorhanden ist, wie folgendes Beispiel zeigt, wobei wir den Datentyp* `Farbe` *aus Abschnitt 2.2.2 verwenden.*

```
case x of
   Rot  -> False
   Gruen -> False
   y -> case x of
          Blau -> True
```

  Hierbei sind die Alternativen des inneren `case`*-Ausdrucks ausreichend, obwohl sie den Typ* `Farbe` *nicht vollständig abdecken, aber durch den übergeordneten* `case`*-Ausdruck ist sichergestellt, dass x nur zum Konstruktor* `Blau` *auswerten kann.*

- *Es gibt keinen* `case`*-Ausdruck mit Alternativen für Konstruktoren unterschiedlichen Typs. Eine Ausnahme hierbei ist der* `case`*-Ausdruck, der ausschließlich eine default-Alternative enthält.*

- *Arithmetische Operatoren werden auch infix verwendet.*

- *Wenn die Bedeutung eindeutig ist, werden Semikola zwischen Bindungen und* `case`*-Alternativen weggelassen.*

- *Funktionen werden manchmal in der Form* $f\ a_1\ \ldots\ a_n = e$ *dargestellt, wobei stets* $f = \lambda\ a_1 \ldots a_n$ `->` e *gemeint ist.*

Die Darstellung von $L_{GHCCore}$ ist an jener in [San95] und [PS94] angelehnt, wobei sie jedoch entsprechend der aktuell verwendeten Kernsprache des GHC angepasst wurde, deren Beschreibung aus [Apt] und [PM02, Seite 400] sowie dem Quellcode[11] entnommen wurde. Im Folgenden werden einige Unterschiede zwischen $L_{GHCCore}$ und der Compiler-internen Sprache beschrieben:

- Die Sprache innerhalb des GHCs ist explizit (durch weitere Sprachkonstrukte) getypt. Wir betrachten die Sprache jedoch als ungetypt.

[11] Die Kernsprache wird im Modul `ghc/compiler/coreSyn/CoreSyn.lhs` definiert. Wir geben innerhalb dieser Arbeit Module des Quellcodes mit ihrem kompletten Verzeichnispfad entsprechend der Verzeichnisstruktur der Quellcode-Distribution des GHC an.

Wir haben zwar innerhalb der Syntax keinerlei Typen mehr, wir nehmen jedoch an, dass die Menge der Konstruktoren in $L_{GHCCore}$ derart partitioniert ist, dass wir die Konstruktoren den jeweils ursprünglichen Typen in der getypten Kernsprache zuordnen können. Beispielsweise bilden die Konstruktoren `True` und `False` eine Partition des ehemaligen Typs `Bool`. Somit lässt sich die Bedingung prüfen, dass alle Konstruktoren des Typs des ersten Arguments eines `case`-Ausdrucks durch die Alternativen abgedeckt werden.

- `case`-Ausdrücke haben in der Realität eine leicht veränderte Darstellung: Sie haben ein weiteres Argument, den „binder“, der eine Variable ist, an den der ausgewertete Ausdruck gebunden wird. Dementsprechend wird die default-Alternative durch `DEFAULT ->` **Expr** dargestellt, wobei `DEFAULT` eine Konstante ist.

- Die Sprache enthält ein weiteres Konstrukt **Note Expr** um Ausdrücke mit Markierungen zu versehen.

Die Kernsprache hat einige Besonderheiten, die wir im Folgenden darstellen und erklären.

Unboxed Objekte

Die Kernsprache enthält Konstrukte, die als „unboxed“[12] bezeichnet werden.

Zum einen gibt es „unboxed values“ (siehe [PL91] und [Apt, Seite 12]). Dies sind Werte für primitive Datentypen. Herkömmliche Datenobjekte werden innerhalb der STG-Maschine wie oben erwähnt in einer Heap-Zelle abgelegt und mittels eines Zeigers adressiert. „Unboxed values“ hingegen werden direkt durch einen Bit-String dargestellt. Der Vorteil besteht darin, dass bei Auswertung des Objektes der Wert direkt in ein Register geladen werden kann, ohne dass der Heap besucht werden muss.

Primitive Operatoren arbeiten auf Argumenten, die „unboxed values“ sind, und der Rückgabewert ist zumeist auch „unboxed“.

Gibt eine Funktion mehrere Werte zurück, so müssten diese mit einem Konstruktor verpackt werden, so dass letztlich nur ein Objekt zurück gegeben wird. Dieses würde dann als „closure“ im Heap abgelegt. Werden diese mehreren Werte jedoch sofort weiterverarbeitet, so würden wiederum unnötige Zugriffe auf den Heap stattfinden. Aufgrund dessen gibt es „unboxed tuples“ wie sie in [Apt, Seite 11ff.] beschrieben sind. Hierbei handelt es sich um Tupel, die nicht mit einem Datenkonstruktor verpackt auf dem Heap abgelegt werden, sondern direkt in Register geladen oder auf dem Stack abgelegt werden können.

[12] engl. für „nicht eingepackt“

Diese „unboxed“ Konstrukte sind sowohl auf der Ebene der STG-Sprache als auch auf der Ebene der GHC-Kernsprache vorhanden, damit sie in den zugehörigen Compilerphasen zwecks Optimierung manipuliert werden können. Die Vorteile dieser Konstrukte werden ausführlich in [PL91] beschrieben.

Operationale Semantik von `let` und `case`

Die Konstrukte `let und case` haben direkte operationale Interpretationen bezüglich der Auswertung mittels der STG-Maschine: `let` bedeutet Heap-Allokation, d.h. ein Ausdruck `let v = e in e'` veranlasst, dass eine „closure“ für `e` im Heap angelegt wird und die Variable `v` daran gebunden wird. `case` bedeutet Auswertung des ersten Arguments, d.h. `case e of v -> e'` bedeutet, dass `e` ausgewertet wird und der erhaltene Wert an `v` gebunden wird. In Haskell haben diese Ausdrücke dieselbe Bedeutung.

2.3.3 Haskell-Erweiterungen des GHC

Der GHC bietet Erweiterungen der Sprache Haskell an, die automatisch verfügbar sind, wenn man den Compiler mit der Option `-fglasgow-exts` aufruft.

Für eine komplette Auflistung der Erweiterungen sei auf [The03, Kapitel 7] verwiesen. Im Folgenden werden diejenigen Erweiterungen dargestellt, die innerhalb dieser Arbeit verwendet werden.

Unboxed values, unboxed tuples und primitive Operationen

Wie bereits erwähnt gibt es unboxed values und unboxed tupels. Diese sind auch auf der Ebene des Haskell-Quellcodes verfügbar.

In Funktionsdefinitionen werden Variablen, die einen zugehörigen „unboxed type“ (siehe [The03, Abschnitt 7.2.1]) haben, meist mit dem Symbol `#` am Ende des Namens versehen, dies ist jedoch nicht zwingend notwendig.

Die zugehörigen Typen und Konstruktoren enden jedoch alle mit dem Symbol `#`. So ist z.B. `Int#` der Typ für primitive Ganzzahlen mit beschränkter Größe und `1#` der Konstruktor für die Zahl 1 vom Typ `Int#`.

In [The03, Abschnitt 7.2] werden auch einige Operationen auf diesen primitiven Werten dargestellt.

Die Syntax von unboxed tupels (siehe [The03, Abschnitt 7.2.2]) ist an jene von herkömmlichen Tupeln in Haskell angelehnt. Die Komponenten werden durch Kommata getrennt, und von Klammern umschlossen, wobei die öffnende Klammer durch `(#` und die schließende Klammer durch `#)` dargestellt wird.

Datentypen ohne Konstruktoren

In [The03, Abschnitt 7.3.1] wird eine Erweiterung beschrieben, die es ermöglicht, Datentypen zu definieren, die keine Datenkonstruktoren besitzen, indem die rechte Seite der `data`-Syntax weggelassen wird. So kann z.B. ein Datentyp `WithoutDataCons` durch

```
data WithoutDataCons
```

definiert werden.

Hierarchische Modulstruktur

Diese in [The03, Abschnitt 7.5.1] beschriebene Erweiterung erlaubt es Modulnamen mit Punkten zu versehen. So kann z.B. ein Modul `A.B.C` definiert werden. Allerdings ist dies eine rein syntaktische Erweiterung des Namens und es bestehen keinerlei Verbindungen zwischen einem Modul mit dem Namen `A.B` und `A.B.C`.

Die Verwendung dieser Syntax hat lediglich Auswirkungen beim Modulimport, wie der GHC nach der zu importierenden Datei im Dateisystem sucht, dies ist genauer in [The03, Abschnitt 4.9.3] dargestellt.

Pragmas

Die in [The03, Anschnitt 7.6] beschriebenen *Pragmas* dienen dazu, die Optimierung des Compilers individuell zu steuern. Mit `INLINE`-Pragmas kann der Compiler dazu angewiesen werden Funktionsnamen durch die Funktionsdefinition zu ersetzen. Mit `RULES`-Pragmas können so genannte „rewrite rules“ definiert werden, mit denen zahlreiche Optimierungen durchgeführt werden können.

2.4 IO in Haskell und Realisierung im GHC

Im Folgenden wird die Realisierung[13] von IO im GHC dargestellt. Zunächst wird ein Überblick über monadisches IO analog zu [Pey01] gegeben, danach folgt eine detaillierte Darstellung der Haskell-Erweiterung `unsafePerformIO`. Nach genaueren Implementierungsdetails endet dieser Abschnitt mit einem Beispiel über die Problematik von Transformationen bei Anwendung von `unsafePerformIO`.

[13] Die Ausführungen beziehen sich auf Version 5.04.

2.4.1 Monadisches IO

In [ABB+99, Seite 62] ist festgelegt, dass jedes Haskell-Programm eine Funktion `main` enthalten muss, die bei Ausführung des Programms aufgerufen wird. Diese Funktion hat den Typ `main :: IO()`. Im Folgenden soll geklärt werden, was sich hinter dem `IO`-Typ verbirgt, und wie man mit IO in Haskell umgehen kann.

Die Grundidee von monadischem IO kann nach [Pey01, Seite 5] folgendermaßen formuliert werden:

> *Ein Wert vom Typ* `IO a` *ist eine „Aktion“, die beim Ausführen eine Ein- oder Ausgabe durchführt, bevor sie einen Wert vom Typ* `a` *liefert.*

Man kann diesen Zusammenhang mit folgender Typdefinition darstellen.

```
type IO a = Welt -> (Welt, a)
```

Dies ist nicht die im GHC verwendete Typdefinition, sondern eine vereinfachte Darstellung, die hier jedoch benutzt wird, da sie ausreicht, um die Grundideen zu präsentieren.

Ein Wert vom Typ `IO a` ist eine Funktion[14], die, wenn sie ein Argument vom Typ `Welt` bekommt, ein Tupel liefert. Dieses besteht aus einem Wert des Typs `Welt` und einem Wert des Typs `a`. Dies ist genau das erwartete Verhalten eines Programms: Es startet in einem Zustand der Welt, führt eine Ein-/Ausgabe durch, und liefert einen neuen Zustand der Welt und ein Ergebnis der IO-Aktion.

Im Folgenden wird angenommen, dass spezielle IO-Operationen `getChar` und `putChar` definiert sind. Deren Typsignatur können wir nun wie folgt angeben.

- `getChar` ist eine IO-Aktion, die ein Zeichen von der Standardeingabe liest und dieses als Resultat der IO-Aktion zurück gibt.

  ```
  getChar :: IO Char
  ```

- `putChar` erwartet ein Zeichen als erstes Argument und gibt dieses auf der Standardausgabe aus. Der Rückgabewert der IO-Aktion besteht aus dem trivialen Null-Tupel `()`.

  ```
  putChar :: Char -> IO ()
  ```

Angenommen, man will eine Funktion `echo` implementieren, die zunächst ein Zeichen liest und dieses dann ausgibt, so kann dies mit den Funktionen `getChar` und `putChar` bewerkstelligt werden.

Hierfür wird allerdings ein Verknüpfungsoperator benötigt, der IO-Aktionen verbindet. Dafür stellt Haskell den Kombinator `(>>=)` zur Verfügung, der folgende Typsignatur besitzt:

[14] In Realität ist dies keine Funktion, sondern eine mit einem Konstruktor verpackte Funktion.

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

Der `(>>=)`-Kombinator wird „bind“ ausgesprochen, er implementiert sequentielle Komposition, d.h. wenn die verbundene IO-Aktion `a >>= f` ausgeführt wird, wird zunächst die IO-Aktion `a` ausgeführt, dann wird das Ergebnis auf `f` angewendet, um eine neue IO-Aktion zu erhalten, die dann ausgeführt wird.

Nun kann die `echo`-Funktion definiert werden:

```
echo :: IO ()
echo =  getChar >>= putChar
```

Es wird zunächst die IO-Aktion `getChar` ausgeführt, um ein Zeichen c zu erhalten, dann wird `putChar` c ausgeführt.

Wollte man `echo` zweimal hintereinander ausführen, so kann man dies nicht mittels `echo >>= echo` tun, da `(>>=)` als zweites Argument eine Funktion erwartet, und keine IO-Aktion. Genauer gesagt soll das Ergebnis `()` des ersten Aufrufs von `echo` verworfen werden. Deswegen bietet Haskell einen weiteren Verknüpfungsoperator `(>>)` an, der mithilfe von `(>>=)` formuliert werden kann:

```
(>>) :: IO a -> IO b -> IO b
(>>) a1 a2 = a1 >>= \_ -> a2
```

Doppeltes Ausführen von `echo` kann somit wie folgt definiert werden:

```
echoTwice :: IO ()
echoTwice = echo >> echo
```

Es ist noch ein weiterer Kombinator notwendig, der es ermöglicht, einen beliebigen Wert in eine IO-Aktion zu verpacken, ohne jedoch irgendwelche Ein- oder Ausgaben zu tätigen. Dies leistet der Kombinator `return`:

```
return :: a -> IO a
```

Nun kann z.B. eine Funktion definiert werden, die zwei Zeichen liest und dieses als Paar zurück gibt:

```
getTwoChars :: IO (Char, Char)
getTwoChars :: getChar >>= \c1 ->
               getChar >>= \c2 ->
               return (c1, c2)
```

Der `(>>=)`-Kombinator ist der einzige, der IO-Aktionen miteinander verknüpft, und er behandelt die Welt „single-threaded“, d.h. er nimmt die Welt und reicht sie von IO-Aktion zu IO-Aktion weiter. Dabei wird die Welt niemals weggeworfen oder dupliziert, unabhängig davon, welchen Code der Programmierer schreibt. Diese Eigenschaft erlaubt es, primitive IO-Operationen wie `getChar` derart zu implementieren, dass die Operation sofort ausgeführt wird[15].

Die Operatoren `(>>=)` und `return`, zusammen mit dem Datentyp `IO`, erfüllen die monadischen Gesetze und formen somit eine *Monade*. Mithilfe von Monaden können „Aktionen“ miteinander kombiniert werden, wobei dies einen imperativen Programmierstil verlangt. Diese Nützlichkeit von Monaden wird z.B. in [Wad92] dargestellt.

In Haskell (siehe [ABB+99, Abschnitt 6.3.6]) ist eine Typklasse `Monad` sowie eine Instanz für den `IO`-Typen implementiert.

Für alle Instanzen von `Monad` ist die `do`-Notation (siehe [ABB+99, Abschnitt 3.14]) verfügbar, die eine einfachere Syntax erlaubt, so kann z.B. die Funktion `getTwoChars` mit der `do`-Notation wie folgt definiert werden:

```
getTwoChars :: IO (Char, Char)
getTwoChar = do
               c1 <- getChar
               c2 <- getChar
               return (c1,c2)
```

2.4.2 Unsicheres IO - `unsafePerformIO`

Für die bisher vorgestellten primitiven IO-Operationen (`(>>=)`, `return`, `putChar`, `getChar`) gilt:

- Alle Operationen außer `(>>=)` haben eine IO-Aktion als Ergebnis, aber haben keine IO-Aktion als Argument.
- Die einzige Operation, die IO-Aktionen kombiniert, ist `(>>=)`.
- Keine der Operationen hat Argumente vom IO-Typ und liefert ein Ergebnis von einem nicht-IO-Typ zurück, d.h. man kommt aus der IO-Monade nicht mehr heraus, sobald man sich einmal in ihr befindet.

Solche Einschränkungen sind manchmal lästig, wie folgendes aus [Pey01, Seiten 13f.] entnommenes Beispiel zeigt. Will man eine Konfigurationsdatei lesen, um bestimmte Optionen für ein Programm zu erhalten, so könnte man versucht sein, folgenden Code zu benutzen:

[15] S. Peyton Jones bezeichnet dies als eine Art von „update in place“(siehe [Pey01, Seite 9]).

```
configFileContents :: [String]
configFileContents = lines (readFile "config") -- FALSCH!!!

useOptimisation :: Bool
useOptimisation = "optimise" `elem` configFileContents
```

Der Code ist nicht korrekt getypt, denn `readFile` hat den Typ

```
readFile :: FilePath -> IO String
```

und `lines` hat den Typ

```
lines :: String -> [String]
```

Um dieses Problem zu lösen, gibt es folgende Möglichkeiten:

- Man führt sämtliche Operationen innerhalb der IO-Monade aus. Dann hätte `configFileContents` den Typ `IO String` und `useOptimisation` den Typ `IO Bool`. Dies bedeutet, man kann nur dann `useOptimisation` testen, wenn man innerhalb der IO-Monade ist.

  Man kann diesen Ansatz verbessern, indem man das Programm in einen monadischen Teil (das Hauptprogramm) und einen rein funktionalen Teil trennt. Allerdings erfordert dies einen erhöhten Programmieraufwand.

- Man führt einen primitiven Operator ein, der aus der IO-Monade herausführt. Dieser Ansatz wird im Folgenden dargestellt.

Was im obigen Beispiel wünschenswert wäre, ist ein Weg von `IO String` zu `String`, d.h. einen Weg aus der IO-Monade heraus. Aber dies ist aus folgendem Grund zunächst nicht möglich:

Das Lesen der Datei ist eine IO-Aktion, und deswegen ist es prinzipiell entscheidend, *wann* die Datei gelesen wird in Relation zu allen anderen IO-Aktionen des Programms. Aber in dem Fall, in dem sichergestellt ist, dass die Datei `config` nicht während der Laufzeit des Programms geändert wird, ist es unerheblich, wann die Datei gelesen wird. Deswegen gibt es eine weitere, als *unsicher* bezeichnete, primitive IO-Operation:

```
unsafePerformIO :: IO a -> a
```

Nun kann das Beispiel wie folgt programmiert werden:

```
configFileContents :: [String]
configFileContents = lines (unsafePerformIO (readFile "config"))

useOptimisation :: Bool
useOptimisation = "optimise" `elem` configFileContents
```

Im Folgenden stellen wir dar, welche Sichtweise in [Pey01] für die Verwendung von `unsafePerformIO` vertreten wird.

Benutzt man den Kombinator, so sollte man dem Compiler zusichern, dass die Ausführungszeit der IO-Operation, relativ zu allen anderen IO-Operationen des Programms, unbedeutend ist. Das Wort „unsafe“ weist darauf hin, dass diese Bedingung nicht durch den Compiler geprüft werden kann, sondern der Programmierer deren Erfüllung zusichern muss.

Da die durch `unsafePerformIO` entkapselten Ein-/Ausgaben zu unvorhersagbaren Zeitpunkten (oder auch gar nicht) geschehen können, muss man genau prüfen, ob die Anwendung des Operators wirklich erlaubt ist.

Der Kombinator kann sogar dazu benutzt werden, das Typsystem von Haskell zu umgehen (siehe [LLC99, Seite 63]), indem man eine Funktion `cast :: a -> b` definiert.

S. Peyton Jones nennt in [Pey01, Seite 15] drei Fälle, für die er das Benutzen von `unsafePerformIO` empfiehlt:

- Das Ausführen einer Ein-/Ausgabe, die nur einmal während der Laufzeit ausgeführt wird.
- Allozieren einer globalen veränderlichen Variablen (siehe [Pey01, Abschnitt 2.5] sowie [LP95] und [PME99]).
- Trace-Nachrichten während eines Debugging-Prozesses ausgeben.

2.4.3 Implementierung von monadischem IO

Für die Implementierung der IO-Monade sind in [Pey01, Seiten 18f.] zwei Alternativen angegeben, die in den folgenden Abschnitten vorgestellt werden.

Monade in der Kernsprache

Die IO-Monade wird durch den Compiler bis hin zum Code-Generator befördert. Die Compiler-interne Kernsprache wird um die monadischen Konstrukte erweitert (z.B. werden `(>>=)` und `return` als Primitive implementiert). Allerdings müssten somit sämtliche Programmtransformationen auf dieser erweiterten Kernsprache stattfinden. Aufgrund des damit verbundenen Aufwands wurde dieser Ansatz nicht im GHC implementiert.

Funktionales Codieren der Monade

Der im Folgenden beschriebene Ansatz wird im GHC benutzt. Aus funktionaler Sicht ist eine IO-Aktion ein Wert des Typs `IO`, der bereits folgendermaßen definiert wurde:

```
type IO a = Welt -> (a, Welt)
```

Wenn das „Welt-Argument“ durch eine Konstante[16] des Typs `Welt` dargestellt wird, können `return` und `(>>=)` direkt implementiert werden:

```
return :: a -> IO a
return a =  \w -> (a,w)

(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) m k = \w -> case m w of
                    (r, w') -> k r w'
```

In der Definition von `(>>=)` sieht man, dass die durch die erste IO-Aktion erhaltene Welt an die zweite IO-Aktion weitergegeben wird.

Die primitiven IO-Operationen, wie z.B. `getChar`, müssen auch implementiert werden, aber dies kann ebenso wie im Falle von anderen primitiven Operationen, etwa wie z.B. die Addition von zwei Integerzahlen, durch Aufruf externer Bibliotheken geschehen.

Dieser Ansatz ist ökonomischer als der erste, allerdings bezeichnet S. Peyton Jones ihn in [Pey01, Seite 17] als „a bit of a hack“, da die Implementierung nur dann korrekt ist, wenn niemals ein Ausdruck, dessen Vervielfältigung Anlass zu zusätzlicher Arbeit geben könnte ([Pey01, Seite 18] bezeichnet einen solchen Ausdruck als „redex“), durch den Compiler dupliziert wird.

Man betrachte den Ausdruck:

```
getChar >>= \c -> (putChar c >> putChar c)
```

Mit der oben angegebenen Definition von `(>>=)` könnte man den Ausdruck folgendermaßen transformieren:

```
\w -> case getChar w of
        (c, w1) -> case putChar c w1 of
                     (_,w2) -> putChar c w2
```

Nun könnte der Compiler den Code durch folgenden Code ersetzen:

```
\w -> case getChar w of
        (c, w1) -> case putChar c w1 of
                     (_,w2) -> putChar (fst(getChar w)) w2
```

[16] S. Peyton Jones bezeichnet diese als ein „un-forgeable token“ in [Pey01, Seite 17].

Hier wurde das letzte `c` durch den Ausdruck `(fst(getChar w))` ersetzt. Dadurch wurde jedoch die Konstante für die Welt dupliziert und es gibt nun zwei statt einem Aufruf von `getChar` und damit kein single-threadness mehr.

In der Realität ersetzt der GHC niemals einen einfachen Ausdruck durch einen komplizierteren wie im Beispiel. Da Redexe (im Sinne von [Pey01]) niemals dupliziert werden, ist die Implementierung korrekt, womit sich in [AS98] beschäftigt wurde.

2.4.4 Implementierung im GHC

Im Folgenden stellen wir die Realisierung des vorher beschriebenen Ansatzes zur Implementierung von monadischem IO im GHC dar.

Der Zustandstyp

Details über die Typen `State` und `RealWorld` finden sich in [The03, Seite 151].

Der primitive Typ `State#` repräsentiert den Zustand eines „state transformer“[17]. Er ist abhängig vom gewünschten Typ des Zustands parametrisiert, um Zustände von unterschiedlichen Threads unterscheiden zu können. Der einzige Effekt dieser Parametrisierung befindet sich jedoch im Typsystem: Alle Werte des Typs `State#` werden in der selben Weise dargestellt. Tatsächlich sind sie durch nichts dargestellt. Der Code-Generator “weiß“, dass er z.B. keinen Code für primitive Zustände generieren und keine Register für primitive Zustände allozieren soll.

```
type State# s
```

Die Welt

Über dem Typ `RealWorld` sind keine Werte definiert, und es existieren keine Operationen auf ihm. Seine einzige Rolle ist es, der Typ zu sein, der den „IO state transformer“ von anderen „state transformern“ unterscheidet.

```
data RealWorld
```

Der Zustand der Welt

Für den Zustand der Welt wird ein einziger primitiver Wert des Typs `State# RealWorld` zur Verfügung gestellt:

```
realWorld# :: State# RealWorld
```

[17] engl. für „Zustandsveränderer“

Die IO-Monade

Im Modul `GHC.IOBase`[18] findet sich die Definition des `IO a`-Typs:

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

Der Rückgabewert, bestehend aus dem Zustand und dem Ergebnis der IO-Aktion, ist ein unboxed tupel, da, wie wir im Folgenden sehen werden, beide Komponenten des Tupels innerhalb des monadischen Operators `(>>=)` als auch innerhalb von `unsafePerformIO` direkt weiter verarbeitet werden.

Ebenso in `GHC.IOBase` finden sich Definitionen für `(>>=)`, `(>>)` und `return` für die IO-Monade:

```
instance  Monad IO  where
    m >> k      = m >>= \ _ -> k
    return x    = returnIO x
    m >>= k     = bindIO m k

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO (\ s -> case m s of
                               (# new_s, a #) -> unIO (k a) new_s)

returnIO :: a -> IO a
returnIO x = IO (\ s -> (# s, x #))

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))
unIO (IO a) = a
```

Die Klasse `Monad` wird im Modul `GHC.Base`[19] definiert:

```
class  Monad m  where
    (>>=)       :: m a -> (a -> m b) -> m b
    (>>)        :: m a -> m b -> m b
    return      :: a -> m a
    fail        :: String -> m a

    m >> k      = m >>= \_ -> k
    fail s      = error s
```

[18] Der Quellcode ist in der Datei `libraries/base/GHC/IOBase.lhs` zu finden.
[19] Der Quellcode befindet sich in `libraries/base/GHC/Base.lhs`.

Primitive IO-Operationen

Primitive IO-Operationen, wie z.B. `getChar`, können genauso wie andere primitive Operationen (wie z.B. die Addition zweier Integer) implementiert werden. Das Laufzeitsystem muss bei dem vorgestellten Ansatz keine Unterscheidung zwischen IO- und nicht-IO-Auswertung machen. Allerdings gibt es für die Wahl der primitiven IO-Operationen beliebig viele Möglichkeiten. Aufgrund dessen wird ein Mechanismus benötigt, der es erlaubt beliebige IO-Bibliotheken aufzurufen. Dies leistet das *Foreign Function Interface* (FFI) [Cha03]. Die Definition von `putChar` kann dann z.B. folgendermaßen erfolgen:

```
foreign import ccall putChar :: Char -> IO()
```

Das Schlüsselwort `foreign` kennzeichnet den Bezug zu externen Funktionen, `import` bedeutet, dass eine externe Funktion nach Haskell importiert wird und `ccall` bedeutet, dass es sich um eine C-Funktion handelt und die entsprechende „Calling Convention“ benutzt werden soll.

unsafePerformIO

Die Implementierung von `unsafePerformIO` findet sich in `GHC.IOBase`:

```
unsafePerformIO :: IO a -> a
unsafePerformIO (IO m) = case m realWorld# of (# _, r #)   -> r
```

Man sieht, dass eine neue Welt benutzt wird, die sofort wieder verworfen wird.

2.4.5 Beispiel zur Problematik des unsafePerformIO

Die von S. Peyton Jones genannten Fälle, in denen er das Benutzen von `unsafePerformIO` empfiehlt, sind sehr eingeschränkt. Zudem ist keine eindeutige Semantik definiert, anhand derer man prüfen kann, ob die Verwendung von `unsafePerformIO` erlaubt ist (siehe hierzu auch Abbildung 1.1).

Aufgrund dessen soll der modifizierte Compiler erlauben, `unsafePerformIO` in beliebigen Situationen anzuwenden, zumal damit ein deklarativerer Programmierstil auch für Programme mit Seiteneffekten möglich ist.

Die bisherige Implementierung des GHC ist nicht in der Lage dies zu leisten, da je nach Compileroption und den damit verbundenen Programmtransformationen bei Verwendung von `unsafePerformIO` Programme mit unterschiedlichem Verhalten entstehen.

Zwei offensichtlich problematische Transformationen sind „Inlining“:

```
let f = e in ... f ...
⇒ let f = e in ... e ...
```

und „Common Subexpression Elimination“ (CSE):

```
 ... e ... e ...
⇒ let v = e in ... v ... v ...
```

Das folgende Beispiel zeigt die Problematik:

Beispiel 2.2. *Folgendes Programm*[20] *benutzt* `unsafePerformIO`*:*

```
module Main where

 data Pack a = C a
 unpack (C x) = x

 a    = unsafePerformIO getChar
 b x  = unsafePerformIO getChar
 c    = C (unsafePerformIO getChar)

 main = do
        putChar a
        putChar a
        putChar (b 5)
        putChar (b 5)
        putChar (unpack c)
        putChar (unpack c)
```

Auf obigem Programm wurden mehrere Testläufe durchgeführt, wobei

- *unterschiedliche Optimierungsstufen des Compilers getestet wurden,*
- *die CSE-Transformation aus- und eingeschaltet wurde,*
- *für die Funktionen* `a, b, c` *Inlining ge- oder verboten wurde und*
- *auf dem Eingabepuffer dem Programm stets die Folge 123456 zur Verfügung gestellt wurde.*

Die Resultate sind in Tabelle 2.1 dargestellt:

Zunächst zeigt die Tabelle, dass für dieses Beispiel kein Unterschied zwischen Optimierungsstufe 1 und 2 zu sehen ist. Deswegen wird die letzte Spalte nicht weiter betrachtet. Das explizite Ausschalten der CSE-Transformation auf Optimierungsstufe 0 führt zu keinem anderen Ergebnis wie ohne diese Option. Der Grund hierfür ist,

		Optimierungsstufe		
		0	1	2
		aa bb cc	aa bb cc	aa bb cc
	Inline	11 23 45	11 11 11	11 11 11
	no Inline	11 23 44	11 11 11	11 11 11
no-cse	Inline	11 23 45	11 23 45	11 23 45
	no Inline	11 23 44	11 22 33	11 22 33

Tabelle 2.1: Ergebnisse der Ausführung des Programms aus Beispiel 2.2

		Optimierungsstufe	
		0	1
		aa bb cc	aa bb cc
	Inline		11 11 11
	no Inline		11 11 11
no-cse	Inline	11 23 45	11 23 45
	no Inline	11 23 44	11 22 33

Tabelle 2.2: Wesentliche Resultate der Ausführung des Programms aus Beispiel 2.2

dass in Stufe 0 keine CSE durchgeführt wird. Aufgrund dessen genügt es, Tabelle 2.2 zu betrachten.

Die Tabelle zeigt, dass die Anzahl der ausgeführten IO-Aktionen je nach Optimierungsstufe und Zulassen/Verbieten von Inlining und CSE unterschiedlich ist.

In den folgenden Kapiteln sollen diejenigen Transformationen ermittelt werden, bei deren Anwendung das Verhalten eines Programms, das `unsafePerformIO` benutzt, verändert wird. Wir stellen hierfür den FUNDIO-Kalkül aus [Sch03a] im nächsten Kapitel dar, um anhand seiner Semantik die Korrektheit von Programmtransformationen festzulegen.

[20] Im Anhang in Abschnitt B.1 ist der vollständige Programmcode, sowie weitere Testläufe, auch mit dem in dieser Arbeit modifizierten Compiler, angegeben.

Kapitel 3

Der FUNDIO-Kalkül

Der FUNDIO-Kalkül [Sch03a] ist ein um ein `letrec`, `case`, Konstruktoren und ein `IO`-Konstrukt erweiterter Lambda-Kalkül, der Sharing beachtet. Die Auswertung des `IO`-Konstruktes wird dabei mittels Nichtdeterminismus realisiert. Der Kalkül eignet sich besonders gut als Semantik für das `unsafePerformIO`, da ein Haskell-Ausdruck der Form (`unsafePerformIO` *monadischer IO-Ausdruck*) bezüglich einer später definierten Übersetzung und sehr ähnlich zu einem `IO`-Ausdruck des FUNDIO-Kalküls ist. Auch hierfür werden später Beispiele gegeben.

Im Folgenden geben wir einen Überblick über den FUNDIO-Kalkül wie er in [Sch03a] definiert ist, indem wir die zugrunde liegende Sprache sowie die zugehörige Reduktion definieren und schließlich die Resultate aus [Sch03a] über korrekte Programmtransformationen wiedergeben. Das Kapitel endet mit einigen zusätzlichen Programmtransformationen, die wir als korrekt beweisen, indem wir analog zu [Sch03a] das Hilfsmittel von vollständigen Vertauschungs- und Gabeldiagrammen in Verbindung mit einem Kontextlemma benutzen.

3.1 Syntax

Die Definition der Sprache des FUNDIO-Kalküls erfolgt analog zu [Sch03a, Abschnitt 4].

Definition 3.1. (L_{FUNDIO})

Es gibt eine endliche Menge $\mathcal{C}$ von Konstruktoren, wobei $|\mathcal{C}| = N > 2$. Die Konstruktoren sind nummeriert, wobei c_i den i-ten Konstruktor bezeichnet. Der N-te Konstruktor c_N ist die spezielle Konstante `lambda`*, die nur innerhalb von Pattern auftreten darf.*

Mit $ar(c_i)$ bezeichnen wir die Stelligkeit des Konstruktors c_i. In Abbildung 3.1 ist die Syntax der im FUNDIO-Kalkül verwendeten Kernsprache angegeben. Diese Sprache bezeichnen wir im Folgenden mit L_{FUNDIO}. Gültige Ausdrücke können beginnend mit dem Nichtterminal **E** *hergeleitet werden, wobei folgende Zusatzbedingungen gelten:*

- *Für den* `case`*-Ausdruck:*

  - *Für jeden Konstruktor $c \in \mathcal{C}$ gibt es genau eine Alternative.*

- *Für den* `letrec`*-Ausdruck:*
  - *Die V_i sind paarweise verschiedene Variablen.*
  - *das* `letrec` *ist kommutativ, d.h. die Reihenfolge der Bindungen ist austauschbar.*
  - *Eine leere Menge von Bindungen ist erlaubt, d.h. Ausdrücke der Form* (`letrec` {} `in` s) *sind zulässig.*

- *Für das Pattern:*
  - *Die V_i sind paarweise verschiedene und neue Variablen.*

E	::=	V	(Variable)
	|	$(c_i\ \mathbf{E}_1 \ldots \mathbf{E}_{ar(c)})$	(Konstruktoranwendung)
	|	(`IO` **E**)	(IO-Ausdruck)
	|	(`case` **E** $\mathbf{Alt}_1 \ldots \mathbf{Alt}_N$)	(case-Ausdruck)
	|	$(\mathbf{E}_1\ \mathbf{E}_2)$	(Applikation)
	|	$(\lambda \mathrm{V}.\mathbf{E})$	(Lambda-Abstraktion)
	|	(`letrec` $\mathrm{V}_1 = \mathbf{E}_1, \ldots, \mathrm{V}_n = \mathbf{E}_n$ `in` **E**)	(letrec-Ausdruck)
Alt	::=	(**Pat** $\rightarrow$ **E**)	(Alternative)
Pat	::=	$(c_j\ \mathrm{V}_1 \ldots \mathrm{V}_{ar(c_j)})$	(Pattern)

wobei $i \in \{1, \ldots, N-1\}$ und $j \in \{1, \ldots N\}$.

Abbildung 3.1: L_{FUNDIO} - Die Sprache des FUNDIO-Kalküls

Konvention 3.2. *Wir verwenden folgende Schreibweisen, um verschiedene Ausdrücke verkürzt darzustellen:*

- *Statt* (`letrec` $x_1 = E_1, \ldots, x_n = E_n$ `in` t) *schreiben wir auch* (`letrec` Env `in` t).
- *Statt* (`case` s Alt_1 $\ldots$ Alt_n) *schreiben wir auch* (`case` s $Alts$).
- *Wenn die Bedeutung eindeutig ist, lassen wir Klammern weg. Die Applikation ist linksassoziativ, d.h.* $(a_1 \ \ldots \ a_n)$ *ist eine Abkürzung für* $(\ldots((a_1\ a_2)\ldots)\ a_n)$.
- *Statt* $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.s))\ldots)$ *schreiben wir auch* $(\lambda x_1 \ldots x_n.s)$.

Wir definieren nun in einer ähnlichen Weise wie [Sch00, Kapitel 1], [Sch02, Kapitel 2] und [Bar84] Bindungsbereiche, freie und gebundene Vorkommen von Variablen sowie freie und gebundene Variablen. Danach vereinbaren wir eine Konvention über die Namen der Variablen in Ausdrücken aus L_{FUNDIO}.

Definition 3.3. (Bindungsbereich)
Abstraktionen, `letrec`*-Ausdrücke und Pattern* binden *Variablen.*

Der Bindungsbereich *der Variablen v_i in*

- $(\lambda v_i.t)$ *ist* t.
- $(\texttt{letrec}\ v_1 = e_1, \ldots, v_i = e_i, \ldots, v_n = e_n\ \texttt{in}\ t)$ *ist* $e_1, \ldots, e_n$ *und* t.
- $((c\ v_1 \ldots v_i \ldots v_n) \to t)$ *ist* t.

Das Vorkommen einer Variablen v in einem Ausdruck bezeichnen wir als gebunden, *wenn das Vorkommen in einem Bindungsbereich der Variablen v ist. Ansonsten bezeichnen wir das Vorkommen der Variablen v als* frei.

Definition 3.4. (Freie Variablen)
Die Menge $\mathcal{FV}(t)$ der freien Variablen *eines Ausdrucks t ist wie folgt induktiv definiert.*

$$
\begin{aligned}
\mathcal{FV}(x) &= \{x\}, \text{ wenn } x \text{ eine Variable ist.}\\
\mathcal{FV}(s\ t) &= \mathcal{FV}(s) \cup \mathcal{FV}(t)\\
\mathcal{FV}(c\ s_1 \ldots s_n) &= \bigcup_{i=1}^{n} \mathcal{FV}(s_i)\\
\mathcal{FV}(\texttt{IO}\ s) &= \mathcal{FV}(s)\\
\mathcal{FV}(\lambda x.s) &= \mathcal{FV}(s) \setminus \{x\}\\
\mathcal{FV}(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e_{n+1}) &= \left(\bigcup_{i=1}^{n+1} \mathcal{FV}(e_i)\right) \setminus \{x_1, \ldots, x_n\}\\
\mathcal{FV}(\texttt{case}\ s\ Alt_1 \ldots Alt_N) &= \mathcal{FV}(s) \cup \bigcup_{i=1}^{N} \mathcal{FV}(Alt_i)\\
\mathcal{FV}((c\ x_1 \ldots x_n) \to s) &= \mathcal{FV}(s) \setminus \{x_1, \ldots, x_n\}
\end{aligned}
$$

Definition 3.5. (Gebundene Variablen)
Die Menge $\mathcal{GV}(t)$ der gebundenen Variablen *eines Ausdrucks t ist wie folgt induktiv definiert.*

$$
\begin{aligned}
\mathcal{GV}(x) &= \emptyset, \text{ wenn } x \text{ eine Variable ist.}\\
\mathcal{GV}(s\ t) &= \mathcal{GV}(s) \cup \mathcal{GV}(t)
\end{aligned}
$$

$$\mathcal{GV}(c\ s_1 \ldots s_n) = \bigcup_{i=1}^{n} \mathcal{GV}(s_i)$$
$$\mathcal{GV}(\texttt{IO}\ s) = \mathcal{GV}(s)$$
$$\mathcal{GV}(\lambda x.s) = \{x\} \cup \mathcal{GV}(s)$$
$$\mathcal{GV}(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e_{n+1}) = \{x_1, \ldots, x_n\} \cup \bigcup_{i=1}^{n+1} \mathcal{GV}(e_i)$$
$$\mathcal{GV}(\texttt{case}\ s\ Alt_1 \ldots Alt_N) = \mathcal{GV}(s) \cup \bigcup_{i=1}^{N} \mathcal{GV}(Alt_i)$$
$$\mathcal{GV}((c\ x_1 \ldots x_n) \to s) = \{x_1, \ldots, x_n\} \cup \mathcal{GV}(s)$$

Definition 3.6. (Geschlossene und offene Ausdrücke)
Sei t ein Ausdruck, dann bezeichnen wir t als geschlossen*, gdw. $\mathcal{FV}(t) = \emptyset$. Ansonsten nennen wir t* offen*.*

Die folgende Konvention entspricht im Wesentlichen [Bar84, Convention 2.1.13.] und [Kut00, Konvention 2.1.1 und 2.1.2].

Konvention 3.7. *Wir nehmen im Folgenden zur Vereinfachung stets an, dass sämtliche gebundene Variablen Namen haben, die sich von den freien Variablen unterscheiden. Zudem sollen alle durch ein* `letrec` *oder eine Abstraktion formulierte Bindungen durch unterschiedliche Variablen erfolgen.*

Die Konvention vermeidet, dass freie Variablen durch eine Termersetzung eingefangen werden, d.h. sich nach der Ersetzung in einem Bindungsbereich befinden und somit ungewollt gebunden werden.

Definition 3.8. *Sei s ein Ausdruck und x eine Variable. Dann bezeichnen wir mit $s[y/x]$ den Ausdruck der entsteht, wenn wir alle freien Vorkommen von x in s durch die Variable y ersetzen. Wir bezeichnen dies als eine* Umbenennung *der Variablen x.*

Damit Konvention 3.7 nicht durch eine Umbenennung verletzt wird, sollte y stets eine neue Variable sein. Andererseits können wir mithilfe der Umbenennung Ausdrücke, die der Konvention nicht entsprechen, transformieren, so dass sie konform zur Konvention sind.

Definition 3.9. *. Wir bezeichnen Terme, die sich nur dadurch unterscheiden, dass gebundene Variablen unterschiedliche Namen haben, als* α-äquivalent *und notieren dies mit dem Symbol $=_\alpha$.*

Im Folgenden betrachten wir analog zu [Bar84, Convention 2.1.12] α-äquivalente Terme als syntaktisch gleich, da die Umbennung von gebundenen Variablen (mit Beachtung von Konvention 3.7) keine Auswirkung auf das Verhalten der Terme hat.

3.2 Kontexte

Kontexte sind Ausdrücke, die an einer Stelle ein „Loch“ besitzen, welches wir im Folgenden mit $[\cdot]$ kennzeichnen. Wir definieren zunächst allgemeine Kontexte für Ausdrücke von L_{FUNDIO}. Danach geben wir Definitionen für die speziellen Reduktions- und Oberflächenkontexte an.

3.2.1 Allgemeine Kontexte

Definition 3.10. (Kontext)
Ein Kontext C *ist*[1] *durch die folgende Grammatik definiert:*

$$
\begin{aligned}
C \;::=\; & [\cdot] \\
\mid\; & (\lambda x.C) \\
\mid\; & (C\ E) \\
\mid\; & (E\ C) \\
\mid\; & (\texttt{IO}\ C) \\
\mid\; & (c\ E\ \ldots\ E\ C\ E\ \ldots\ E) \\
\mid\; & (\texttt{case}\ C\ Alts) \\
\mid\; & (\texttt{case}\ E\ Alt_1 \ldots (Pat \to C) \ldots Alt_n) \\
\mid\; & (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ C) \\
\mid\; & (\texttt{letrec}\ x_1 = E_1, \ldots, x_{i-1} = E_{i-1}, x_i = C, x_{i+1} = E_{i+1}, \ldots x_n = E_n\ \texttt{in}\ E)
\end{aligned}
$$

Sei D ein Kontext, dann bezeichnen wir mit $D[t]$ den Ausdruck, der entsteht, wenn wir im Kontext D das Loch durch den Ausdruck t ersetzen.

3.2.2 Reduktionskontexte

Reduktionskontexte sind solche Kontexte, in denen wir später reduzieren und die insbesondere zur Definition der Normalordnungsreduktion notwendig sind. Die Klasse der Reduktionskontexte besitzt eine Unterklasse, die Menge der schwachen Reduktionskontexte, mithilfe derer diese Klasse definiert wird[2].

Definition 3.11. (Reduktionskontexte)

[1] analog zu [Sch03a, Seite 9]
[2] Die Definition entspricht [Sch03a, Definition 4.1]

$$
\begin{array}{lcl}
R^- & ::= & [\cdot] \\
 & | & (R^-\ E) \\
 & | & (\texttt{case}\ R^-\ \mathit{Alts}) \\
 & | & (\texttt{IO}\ R^-) \\
R & ::= & R^- \\
 & | & (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ R^-) \\
 & | & (\texttt{letrec}\ x_1 = R_1^-, \ldots, x_j = R_j^-[x_{j-1}], \ldots\ \texttt{in}\ R^-[x_j]) \\
 & & \textit{wobei } R^-, R_i^- \textit{ Kontexte der Klasse } R^- \textit{ sind.}
\end{array}
$$

R wird als Reduktionskontext*, R^- als* schwacher Reduktionskontext *bezeichnet.*

Sei t ein Ausdruck mit $t = R^-[t_0]$, dann nennen wir R^- maximal (für t)*, wenn es keinen größeren schwachen Reduktionskontext mit dieser Eigenschaft gibt.*

Sei $t = R[t_0]$ ein Ausdruck, so nennen wir R einen maximalen Reduktionskontext (für t)*, wenn R*

- *ein maximaler schwacher Reduktionskontext R^- ist,*
- *von der Form* $(\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ R^-)$ *ist, wobei R^- ein maximaler schwacher Reduktionskontext ist und $\forall j : t_0 \neq x_j$, oder*
- *von der Form* $(\texttt{letrec}\ x_1 = R_1^-, x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], \ldots\ \texttt{in}\ R^-[x_j])$ *ist, wobei $t = R[t_0] = (\texttt{letrec}\ x_1 = R_1^-[t_0], \ldots\ \texttt{in}\ R^-[x_j])$, R_1^- ein maximaler schwacher Reduktionskontext für $R_1^-[t_0]$ und die Anzahl j von beteiligten Bindungen maximal ist.*

3.2.3 Oberflächenkontexte

Oberflächenkontexte sind solche Kontexte, die kein Loch im Rumpf einer Abstraktion besitzen. Wir definieren diese im Folgenden[3]:

Definition 3.12. (Oberflächenkontext)

[3] analog zu [Sch03a, Definition 4.2]

Ein Oberflächenkontext S *wird durch folgende Grammatik definiert:*

$$
\begin{aligned}
S \;::= {} & [\cdot] \\
\mid {} & (S\ E) \\
\mid {} & (E\ S) \\
\mid {} & (\texttt{IO}\ S) \\
\mid {} & (c\ E\ \ldots\ E\ S\ E\ \ldots\ E) \\
\mid {} & (\texttt{case}\ S\ Alts) \\
\mid {} & (\texttt{case}\ E\ Alt_1 \ldots (Pat \to S) \ldots Alt_n) \\
\mid {} & (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ S) \\
\mid {} & (\texttt{letrec}\ x_1 = E_1, \ldots, x_{i-1} = E_{i-1}, x_i = S, x_{i+1} = E_{i+1}, \ldots x_n = E_n\ \texttt{in}\ E)
\end{aligned}
$$

3.2.4 Beispiele

Zum besseren Verständnis geben wir einige Beispiele für die verschiedenen Kontextarten an:

- Jeder schwache Reduktionskontext ist auch ein Reduktionskontext, jeder Reduktionskontext ist auch ein Oberflächenkontext und jeder Oberflächenkontext ist auch ein Kontext.
- Die Ausdrücke mit Loch $(\lambda[\cdot].s)$ und $(\texttt{case}\ s\ \ldots(c\ a_1\ \ldots[\cdot]\ldots\ a_n\ \to\ t)\ldots)$ sind keine Kontexte.
- Der Kontext $C = (\lambda x.([\cdot]\ x))$ ist weder ein Reduktions- noch ein Oberflächenkontext. Sei $t = (\lambda y.y)$, dann ist $C[t] = (\lambda x.((\lambda y.y)\ x))$.
- Der Kontext $S = ((\lambda x.x)\ [\cdot])$ ist ein Oberflächenkontext aber kein Reduktionskontext.
- Für den Ausdruck $t = ((\texttt{IO}\ c)\ (\lambda x.x))$ mit $c \in \mathcal{C}$ ist $R_1 = ([\cdot]\ (\lambda x.x))$ ein schwacher Reduktionskontext, aber nicht maximal.

  Der Kontext $R_2 = ((\texttt{IO}\ [\cdot])\ (\lambda x.x))$ hingegen ist ein maximaler schwacher Reduktionskontext für t.
- Der Kontext

  $$(\texttt{letrec}\ x_1 = ([\cdot]\ c), x_2 = x_1, x_3 = (\texttt{case}\ x_2\ Alts)\ \texttt{in}\ (\texttt{IO}\ x_3))$$

  ist der maximale Reduktionskontext für den Ausdruck

  $$(\texttt{letrec}\ x_1 = ((\lambda y.y)\ c), x_2 = x_1, x_3 = (\texttt{case}\ x_2\ Alts)\ \texttt{in}\ (\texttt{IO}\ x_3)).$$

3.3 Reduktionsregeln

Definition 3.13. (Reduktionsregeln)
In den Abbildungen 3.2 und 3.3 sind die Reduktionsregeln[4] *des FUNDIO-Kalküls definiert. Wobei eine Regel*

$$(name) \qquad a \longrightarrow b$$

wie folgt zu lesen ist: Ein Ausdruck der Form a kann durch einen Ausdruck der Form b ersetzt werden, dabei wird die Regel mit der Bezeichnung (name) verwendet.

Die Vereinigung der Regeln (cp-in) und (cp-e) nennen wir (cp). Die Vereinigung der Regeln (llet-in), (llet-e) bezeichnen wir mit (llet). Die Vereinigung der Regeln (case-c),(case-in),(case-e) und (case-lam) bezeichnen wir mit (case). Die Vereinigung der Regeln (IOr-c), (IOr-in) und (IOr-e) bezeichnen wir mit (IOr).

Wenn notwendig, notieren wir die verwendete Reduktionsregel oder auch den verwendeten Kontext über dem Reduktionspfeil, z.B. ist $\xrightarrow{R,case}$ eine (case)-Reduktion in einem Reduktionskontext.

Die transitive Hülle bezeichnen wir mit einem +, die reflexiv-transitive Hülle mit $*$, so ist z.B. $\xrightarrow{(llet)^+}$, die transitive Hülle von $\xrightarrow{llet}$. Wir verwenden des Weiteren die Notation $\xrightarrow{red^n}$, wenn n Reduktionen der Regel red folgen, sowie die Notation $\xrightarrow{red^{a \vee b}}$, wenn a oder b Reduktionen der Regel red folgen.

Man beachte, dass die (IOr)-Reduktion nichtdeterministisch ist. Sie modelliert IO-Aktionen im FUNDIO-Kalkül in folgender Weise: Wenn (`IO` c) $\xrightarrow{IOr}$ d, dann wird zunächst der *Ausgabewert c* ausgegeben und danach der *Eingabewert d* nichtdeterministisch erhalten. Die Vorstellung dabei ist, dass der Benutzer den Eingabewert eingibt, das Programm somit nicht weiß, welcher Wert als Ergebnis der (IOr)-Reduktion zu erwarten ist.

3.4 Normalordnungsreduktion

Definition 3.14. (Normalordnungsreduktion)
Sei t ein geschlossener Ausdruck. Sei R der maximale Reduktionskontext, so dass $t = R[t']$ *für einen Term* t'. *Die* Normalordnungsreduktion $\xrightarrow{n}$ *ist*[5] *durch einen der folgenden Fälle definiert:*

[4] analog zu [Sch03a, Definition 4.3]
[5] analog zu [Sch03a, Defintion 5.1]

(lbeta) $((\lambda x.\ s)\ t) \longrightarrow (\texttt{letrec}\ x = t\ \texttt{in}\ s)$

(cp-in) $(\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, Env\ \ \texttt{in}\ C[x_j])$
$\longrightarrow (\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, Env\ \texttt{in}\ C[s_1])$
wobei s_1 eine Abstraktion ist

(cp-e) $(\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, x_{j+1} = C[x_j], Env\ \ \texttt{in}\ s)$
$\longrightarrow (\texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, x_{j+1} = C[s_1], Env\ \texttt{in}\ s)$
wobei s_1 eine Abstraktion ist

(llet-in) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ (\texttt{letrec}\ \ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r))$
$\longrightarrow (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r)$

(llet-e) $(\texttt{letrec}\ x_1 = s_1, \ldots,$
$\qquad x_i = (\texttt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ s_i), \ldots,$
$\qquad x_n = s_n$
$\texttt{in}\ r)$
$\rightarrow (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \texttt{in}\ r)$

(lapp) $((\texttt{letrec}\ Env\ \texttt{in}\ t)\ s) \longrightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$

(lcase) $(\texttt{case}\ (\texttt{letrec}\ Env\ \texttt{in}\ t)\ Alts) \longrightarrow (\texttt{letrec}\ \ Env\ \texttt{in}\ (\texttt{case}\ t\ Alts))$

(case-c) $(\texttt{case}\ (c_i\ t_1 \ldots t_n)\ \ldots\ ((c_i\ y_1 \ldots y_n) \rightarrow t) \ldots)$
$\longrightarrow (\texttt{letrec}\ y_1 = t_1, \ldots, y_n = t_n\ \texttt{in}\ t)$

(case-lam) $(\texttt{case}\ (\lambda x.\ s)\ \ldots\ \texttt{lambda} \rightarrow t) \ldots) \longrightarrow\ (\texttt{letrec}\ \{\}\ \texttt{in}\ t)$

(case-in) $(\texttt{letrec}\ x_1 = (c_i\ t_1\ \ \ldots\ \ t_n), x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots$
$\texttt{in}\ C[\texttt{case}\ x_m\ \ \ldots((c_i\ z_1\ \ \ldots\ \ z_n) \rightarrow t)])$
$\longrightarrow (\texttt{letrec}\ x_1 = (c_i\ y_1\ \ \ldots\ \ y_n), y_1 = t_1, \ldots, y_n = t_n$
$\qquad\qquad x_2 = x_1, \ldots, x_m = x_{m-1, \ldots}$
$\qquad \texttt{in}\ C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ t)])$

(case-e) $(\texttt{letrec}\ x_1 = (c_i\ t_1\ \ \ldots\ \ t_n), x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots$
$\qquad u = C[\texttt{case}\ x_m\ \ \ldots((c_i\ z_1\ \ \ldots\ \ z_n) \rightarrow r_1)]$
$\texttt{in}\ r_2)$
$\longrightarrow (\texttt{letrec}\ x_1 = (c_i\ t_1\ \ \ldots\ \ t_n),$
$\qquad\qquad y_1 = t_1, \ldots, y_n = t_n,$
$\qquad\qquad x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots$
$\qquad\qquad u = C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ r_1)]$
$\qquad \texttt{in}\ r_2)$
wobei die y_i neue Variablen sind

Abbildung 3.2: Reduktionsregeln des FUNDIO-Kalküls

(IOlet) $(\texttt{IO}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)) \longrightarrow (\texttt{letrec}\ Env\ \ \texttt{in}\ (\texttt{IO}\ s))$

In den folgenden drei Regeln sind c und d Konstanten und (c, d) ist das *IO-Paar* der Reduktion.

(IOr-c) $(\texttt{IO}\ c) \longrightarrow d$

(IOr-in) $(\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ C[(\texttt{IO}\ \ x_m)])$
$\longrightarrow (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ C[d])$

(IOr-e) $(\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, u = C[(\texttt{IO}\ \ x_m)], Env\ \texttt{in}\ r)$
$\longrightarrow (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, u = C[d], Env\ \ \texttt{in}\ r)$

Abbildung 3.3: IO-Reduktionsregeln des FUNDIO-Kalküls

1. *t' ist ein* `letrec`*-Ausdruck und R ist nicht trivial. Sei R_0 ein Reduktionskontext, so können folgende Fälle auftreten:*
   a) *$R = R_0[(\texttt{IO}\ [\cdot])]$. Dann reduziere $(\texttt{IO}\ t')$ mit der Regel (IOlet).*
   b) *$R = R_0[([\cdot]\ s)]$. Dann reduziere $(t'\ s)$ mit der Regel (lapp).*
   c) *$R = R_0[(\texttt{case}\ [\cdot]\ Alts)]$. Dann reduziere $(\texttt{case}\ t'\ Alts)$ mit der Regel (lcase).*
   d) *$R = (\texttt{letrec}\ Env\ \texttt{in}\ [\cdot])$. Dann reduziere t mit der Regel (llet-in).*
   e) *$R = (\texttt{letrec}\ x = [\cdot], Env\ \texttt{in}\ t'')$. Dann reduziere t mit der Regel (llet-e).*
2. *t' ist eine Konstruktoranwendung. Es können folgende Fälle auftreten:*
   a) *$R = R_0[(\texttt{case}\ [\cdot]\ Alts)]$, wobei R_0 ein Reduktionskontext ist. Dann reduziere $(\texttt{case}\ t'\ Alts)$ mit der Regel (case-c).*
   b) *$R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[(\texttt{case}\ x_m\ Alts)])$, wobei R_0^- ein schwacher Reduktionskontext ist. Dann reduziere t mit der Regel (case-in).*
   c) *$R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[(\texttt{case}\ x_m\ Alts)], Env\ \texttt{in}\ t'')$, wobei R_0^- und R_1^- schwache Reduktionskontexte sind und y in einem Reduktionskontext ist. Dann reduziere t mit der Regel (case-e).*
3. *t' ist eine Konstante und keiner der vorhergehenden Fälle passt. Es können folgende Fälle auftreten:*
   a) *$R = R_0[(\texttt{IO}\ [\cdot])]$, wobei R_0 ein Reduktionskontext ist. Dann reduziere $(\texttt{IO}\ t')$ mit der Regel (IOr-c).*

b) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[(\texttt{IO}\ x_m)])$, *wobei* R_0^- *ein schwacher Reduktionskontext ist. Dann reduziere* t *mit der Regel (IOr-in).*

c) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[(\texttt{IO}\ x_m)], Env\ \texttt{in}\ t'')$, *wobei* R_0^- *und* R_1^- *schwache Reduktionskontexte sind und* y *in einem Reduktionskontext ist. Dann reduziere* t *mit der Regel (IOr-e).*

4. t' *ist eine Abstraktion. Es können folgende Fälle auftreten:*

a) $R = R_0[(\texttt{case}\ [\cdot]\ Alts)]$, *wobei* R_0 *ein Reduktionskontext ist. Dann reduziere* $(\texttt{case}\ t'\ Alts)$ *mit der Regel (case-lam).*

b) $R = R_0[([\cdot]\ t'')]$, *wobei* R_0 *ein Reduktionskontext ist. Dann reduziere* $(t'\ t'')$ *mit der Regel (lbeta).*

c) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[x_m])$, *wobei* R_0^- *ein schwacher Reduktionskontext ist. Dann reduziere* t *mit der Regel (cp-in), so dass* $R_0^-[x_m]$ *durch* $R_0^-[t']$ *ersetzt wird.*

d) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[(x_m\ s)], Env\ \texttt{in}\ t'')$, *wobei* R_0^- *und* R_1^- *schwache Reduktionskontexte sind und* y *in einem Reduktionskontext ist. Dann reduziere* t *mit der Regel (cp-e), so dass* $R_0^-[(x_m\ s)]$ *durch* $R_0^-[(t'\ s)]$ *ersetzt wird.*

e) $R = (\texttt{letrec}\ x_1 = [\cdot], x_2 = x_1, \ldots, x_m = x_{m-1}, y = R_0^-[(\texttt{case}\ x_m\ Alts)], Env\ \texttt{in}\ t'')$, *wobei* R_0^- *und* R_1^- *schwache Reduktionskontexte sind und* y *in einem Reduktionskontext ist. Dann reduziere* t *mit der Regel (cp-e), so dass* $R_0^-[(\texttt{case}\ x_m\ Alts)]$ *durch* $R_0^-[(\texttt{case}\ t'\ Alts)]$ *ersetzt wird.*

Der *Normalordnungsredex* ist der gesamte Unterausdruck, auf den die entsprechende Reduktionsregel angewendet wird.

Beispiel 3.15. *Wir reduzieren den folgenden Ausdruck* t *in Normalordnung: Seien* $c, d \in \mathcal{C}$ *Konstanten.*

$$
\begin{aligned}
t &= (\texttt{letrec}\ x_1 = ((\lambda y.y)\ c), x_2 = x_1, x_3 = (\texttt{case}\ x_2\ \ldots(c \to c)\ldots)\ \texttt{in}\ (\texttt{IO}\ x_3)) \\
&\xrightarrow{n,lbeta} (\texttt{letrec}\ x_1 = (\texttt{letrec}\ y = c\ \texttt{in}\ y), x_2 = x_1, x_3 = (\texttt{case}\ x_2\ \ldots(c \to c)\ldots) \\
&\quad \texttt{in}\ (\texttt{IO}\ x_3)) && \text{(Fall 4.b))} \\
&\xrightarrow{n,llet-e} (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = (\texttt{case}\ x_2\ \ldots(c \to c)\ldots)\ \texttt{in}\ (\texttt{IO}\ x_3)) && \text{(Fall 1.e))} \\
&\xrightarrow{n,case-e} (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = (\texttt{letrec}\ \{\}\ \texttt{in}\ c)\ \texttt{in}\ (\texttt{IO}\ x_3)) && \text{(Fall 2.c))} \\
&\xrightarrow{n,llet-e} (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = c\ \texttt{in}\ (\texttt{IO}\ x_3)) && \text{(Fall 1.e))} \\
&\xrightarrow{n,IOr-in} (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = c\ \texttt{in}\ d) && \text{(Fall 3.b))}
\end{aligned}
$$

Nun ist keine Regel mehr anwendbar.

Im Folgenden definieren wir Werte und schwache Kopfnormalformen[6].

Definition 3.16. *Ein* Wert *ist eine Konstruktoranwendung oder eine Abstraktion.*

Definition 3.17. *Eine* schwache Kopfnormalform (WHNF[7]) *ist*

- *ein Wert oder*
- *ein Ausdruck der Form* (`letrec` *Env* `in` t)*, wobei* t *ein Wert ist, oder*
- *ein Ausdruck der Form* (`letrec` $x_1 = (c\ t_1\ \ldots\ t_{ar(c)}), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$ `in` x_m).

Beispiel 3.18. *Der letzte Ausdruck aus Beispiel 3.15, auf den keine Regel mehr anwendbar ist, ist eine WHNF, da* d *ein Wert ist.*

Man beachte, dass für eine WHNF keine Normalordnungsreduktion existiert.

Wir definieren ferner Fehlerterme[8].

Definition 3.19. *Ein* Fehlerterm *ist ein Ausdruck, der*

1. *keine WHNF ist und*
2. *keinen Normalordnungsredex besitzt und*
3. *nicht von der Form* $R[x]$ *ist, wobei* R *ein Reduktionskontext und* x *eine freie Variable ist.*

Beispiel 3.20. *Der Ausdruck* (`letrec` $x = x$ `in` x) *ist keine WHNF aber ein Fehlerterm, denn der maximale Reduktionskontext ist* (`letrec` $x = [\cdot]$ `in` x) *und es ist keine der Regeln aus Definition 3.14 anwendbar.*

3.5 Kontextuelle Äquivalenz

Eine *Reduktionsfolge* $s_1 \to \ldots \to s_n$ ist eine Folge von Reduktionen des FUNDIO-Kalküls, wenn sie nicht anders spezifiziert werden. Eine Reduktionsfolge beginnend mit einem Ausdruck t, die ausschließlich aus Normalordnungsreduktionen besteht, bezeichnen wir als *no-Reduktionsfolge von* t.

Im Folgenden definieren wir IO-Multimengen und -Sequenzen sowie Terminierung[9] für den FUNDIO-Kalkül, um im Anschluss die kontextuelle Äquivalenz[10] zu definieren.

[6] analog zu [Sch03a, Defintion 5.2 und Definition 5.3]
[7] Abkürzung für **w**eak **h**ead **n**ormal **f**orm (engl.).
[8] analog zu [Sch03a, Defintion 5.4]
[9] analog zu [Sch03a, Definition 6.1]
[10] analog zu [Sch03a, Definition 6.3]

3.5.1 IO-Multimengen und IO-Sequenzen

Definition 3.21. (IO-Paare, IO-Multimengen und IO-Sequenzen)

Ein IO-Paar *ist ein Paar* (a, b) *wobei* a *und* b *beliebige Konstanten aus* $\mathcal{C}$ *sind:*

- *Das IO-Paar einer* (IOr)*-Reduktion ist das Paar* (c, d) *bestehend aus dem Ausgabe- und dem Eingabewert, wie es in Abbildung 3.3 definiert ist.*
- *Reduktionen von Typ* a *mit* $a \notin \{(\text{IOr-c}), (\text{IOr-in}), (\text{IOr-e})\}$ *besitzen kein IO-Paar.*

Eine IO-Sequenz *ist eine endliche Folge von IO-Paaren.*

Die IO-Sequenz $IOS(s_1 \to \ldots \to s_n)$ *einer Reduktionsfolge* $s_1 \to \ldots \to s_n$ *ist wie folgt definiert:*

- *Falls* $s_1 \to s_2$ *eine* (IOr)*-Reduktion mit IO-Paar* (a, b) *ist, dann ist* $IOS(s_1 \to \ldots \to s_n) := (a, b), IOS(s_2 \to \ldots \to s_n)$.
- *Falls* $s_1 \to s_2$ *keine* (IOr)*-Reduktion ist, dann ist* $IOS(s_1 \to \ldots \to s_n) := IOS(s_2 \to \ldots \to s_n)$.

Eine IO-Multimenge *ist eine endliche Multimenge von IO-Paaren. Die IO-Multimenge* $IOM(s_1 \to \ldots \to s_n)$ *der Reduktionsfolge* $s_1 \to \ldots \to s_n$ *ist die Multimenge bestehend aus den Elementen von* $IOS(s_1 \to \ldots \to s_n)$.

3.5.2 Terminierung

Definition 3.22. *Sei* t *ein Ausdruck und* P *eine endliche IO-Multimenge.*

- *Wir schreiben* $t{\Downarrow}(P)$*, wenn es eine no-Reduktionsfolge* Q *von* t *gibt, die mit einer WHNF endet und* $IOM(Q) = P$ *gilt. Wir sagen dann,* t terminiert für die IO-Multimenge P.
- *Falls es keine no-Reduktionsfolge mit vorheriger Eigenschaft gibt, schreiben wir* $t{\Uparrow}(P)$.

Beispiel 3.23. *Seien* c *und* d *Konstanten aus* $\mathcal{C}$ *und sei* t *wie in Beispiel 3.15 definiert, dann terminiert* t *für die IO-Multimenge* $\{(c, d)\}$*, bzw. für* $P = \{(c, d)\}$ *gilt* $t{\Downarrow}(P)$*, denn in Beispiel 3.15 ist eine no-Reduktionsfolge von* t *angegeben, die mit einer WHNF (siehe Beispiel 3.18) endet.*

Definition 3.24. *Sei* t *ein Ausdruck und* $\overrightarrow{P}$ *eine IO-Sequenz.*

- *Die no-Reduktionsfolge von t* entlang $\overrightarrow{P}$ *ist die längste no-Reduktionsfolge Q von t, mit der Eigenschaft, dass $IOS(Q)$ ein Prefix von $\overrightarrow{P}$ ist.*
- *Wir bezeichnen $\overrightarrow{P}$ als* gültig *für t, wenn die no-Reduktionsfolge von t entlang $\overrightarrow{P}$ mit einer WHNF endet oder die no-Reduktionsfolge von t entlang $\overrightarrow{P}$ die IO-Sequenz $\overrightarrow{P}$ hat.*
- *Wir schreiben $t{\downarrow}(\overrightarrow{P})$ für eine endliche IO-Sequenz $\overrightarrow{P}$, gdw. es eine no-Reduktionsfolge von t entlang $\overrightarrow{P}$ gibt, die mit einer WHNF endet.*
- *Wir schreiben $t{\Downarrow}(\overrightarrow{P})$ für eine endliche IO-Sequenz $\overrightarrow{P}$, gdw. mindestens eine der folgenden Bedingungen gilt:*
  1. *$\overrightarrow{P}$ ist nicht gültig für t.*
  2. *$\overrightarrow{P}$ ist gültig für t und es gibt eine no-Reduktionsfolge von t zu einer WHNF, deren IO-Sequenz ein Prefix von $\overrightarrow{P}$ ist.*
  3. *$\overrightarrow{P}$ ist gültig für t und es gibt eine no-Reduktionsfolge Q von t mit $t \xrightarrow{Q} t'$, $IOS(Q) = \overrightarrow{P}$ und es gibt eine IO-Multimenge P' mit $t'{\Downarrow}(P')$.*
- *Wir bezeichnen t als* fehlerfrei, *gdw. $\forall \overrightarrow{P}$: $t{\Downarrow}(\overrightarrow{P})$, ansonsten sagen wir t* hat einen Fehler *und schreiben dies als $t{\Uparrow}$.*

3.5.3 Kontextuelle Äquivalenz

Definition 3.25. (Kontextuelle Präordnung und Äquivalenz)
Die kontextuelle Präordnung $\leq_c$ *auf Termen s, t ist folgende binäre Relation:*

$$s \leq_c t \text{ gdw. } \forall C[\cdot] : \big((\forall P : C[s]{\Downarrow}(P) \implies C[t]{\Downarrow}(P)) \wedge (C[t]{\Uparrow} \implies C[s]{\Uparrow})\big)$$

Die kontextuelle Äquivalenz $\sim_c$ *auf Termen s, t ist die binäre Relation mit*

$$s \sim_c t \text{ gdw. } s \leq_c t \wedge t \leq_c s$$

Die zweite Bedingung $\forall C[\cdot] : C[t]{\Uparrow} \implies C[s]{\Uparrow}$ aus der Definition für die kontextuelle Präordnung kann auch als $\forall C[\cdot] : (\forall \overrightarrow{P} : C[s]{\Downarrow}(\overrightarrow{P})) \implies (\forall \overrightarrow{P} : C[t]{\Downarrow}(\overrightarrow{P}))$ formuliert werden.

Definition 3.26. *Eine* Präkongruenz *ist eine Präordnung $\preceq$ auf Ausdrücken, mit der Eigenschaft $s \preceq t \implies C[s] \preceq C[t]$ für alle Kontexte C.*

Eine Kongruenz *ist eine Präkongruenz, die zusätzlich eine Äquivalenzrelation ist.*

Lemma 3.27. *Die Relationen $\leq_c$ und $\sim_c$ besitzen folgende Eigenschaften:*

- $\leq_c$ *ist eine Präkongruenz.*
- $\sim_c$ *ist eine Kongruenz.*

Beweis. Siehe [Sch03a, Proposition 6.6] □

3.5.4 Kontextlemma

Im Folgenden geben wir das so genannte Kontextlemma aus [Sch03a] wieder. Es besagt, dass es genügt, Reduktionskontexte zu betrachten, um eine $\leq_c$-Beziehung zwischen zwei Ausdrücken nachzuweisen.

Lemma 3.28. (Kontextlemma)
Seien s und t Terme. Wenn für alle Reduktionskontexte R und alle IO-Multimengen P gilt, dass $(R[s]\Downarrow(P) \implies R[t]\Downarrow(P))$ und für alle Reduktionskontexte R gilt, dass $(\forall \overrightarrow{P} : R[s]\Downarrow(\overrightarrow{P})) \implies (\forall \overrightarrow{P} : R[t]\Downarrow(\overrightarrow{P}))$, dann gilt $s \leq_c t$.

Beweis. Siehe [Sch03a, Lemma 7.3]. □

Des Weiteren geben wir noch ein weiteres Kriterium an, dass wir benutzen, um kontextuelle Äquivalenz nachzuweisen.

Lemma 3.29. *Seien s und t Terme. Eine hinreichende Bedingung für $s \sim_c t$ ist:*

$$\forall R, \overrightarrow{P} :$$
$$\left(R[s]\downarrow(\overrightarrow{P}) \Leftrightarrow R[t]\downarrow(\overrightarrow{P})\right) \wedge \left(\overrightarrow{P} \text{ ist gültig für } R[s] \Leftrightarrow \overrightarrow{P} \text{ ist gültig für } R[t]\right)$$

Beweis. Siehe [Sch03a, Lemma 7.4] □

3.6 Programmtransformationen

Definition 3.30. (Korrekte Programmtransformation)
Eine Programmtransformation *ist eine binäre Relation zwischen Ausdrücken.*

Eine Programmtransformation T ist korrekt, *wenn für zwei beliebige Ausdrücke $s_1, s_2 \in L_{FUNDIO}$ gilt: Aus $s_1\ T\ s_2$ folgt $s_1 \sim_c s_2$.*

3.6.1 Programmtransformationen des FUNDIO-Kalküls

Im folgenden Satz halten wir kurz fest, welche der Reduktionsregeln korrekte Programmtransformationen sind.

Satz 3.31. *Die Reduktionsregeln* (lbeta), (lapp), (llet), (lapp), (lcase), (IOlet), (cp), (case) *sind korrekte Programmtransformationen.*

Die Reduktionsregeln (IOr-c), (IOr-in), (IOr-e) *sind keine korrekten Programmtransformationen, wenn* $|\mathcal{C}| \geq 2$.

Beweis. Siehe [Sch03a, Theorem 16.1] und [Sch03a, Proposition 16.2] □

Weitere Programmtransformationen aus [Sch03a] sind in Abbildung 3.4 definiert. Wir fassen die einzelnen Transformationen wie folgt zusammen:

- Mit (gc) bezeichnen wir die Vereinigung der Regeln (gc-1) und (gc-2).
- Mit (cpx) bezeichnen wir die Vereinigung der Regeln (cpx-in) und (cpx-e).
- Mit (cpcx) bezeichnen wir die Vereinigung der Regeln (cpcx-in) und (cpcx-e).
- Mit (ucp) bezeichnen wir die Vereinigung der Regeln (ucp-1) und (ucp-2).

In [Sch03a, Proposition 12.5, 13.4, 14.6, 17.5, 18.1, 18.2 und Lemma 14.2] wurde bewiesen:

Satz 3.32. (gc), (cpx), (cpcx), (llift), (ucp), (xch) *und* (betavar) *sind korrekte Programmtransformationen.*

Im Folgenden halten wir noch ein weiteres Resultat aus [Sch03a] fest, dass wir später verwenden möchten.

Satz 3.33. *Sei* $\perp$ *ein (geschlossener) nichtterminierender Ausdruck in* L_{FUNDIO}, *d.h.* $\perp\Uparrow$. *Dann gilt für jeden Reduktionskontext* $R[\perp] \sim_c \perp$.

Beweis. Siehe [Sch03a, Corollary 20.17]. □

Definition 3.34. *Wir definieren wie in [Sch03a, Abschnitt 10] die Reduktion (lll) als die Vereinigung der Reduktionen* (llet), (lapp), (lcase) *und* (IOlet).

Garbage Collection

(gc-1) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, Env\ \texttt{in}\ t) \longrightarrow (\texttt{letrec}\ Env\ \texttt{in}\ t)$
wenn für alle $i : x_i$ kommt weder in Env noch in t vor.

(gc-2) $(\texttt{letrec}\ \{\}\ \texttt{in}\ t) \longrightarrow t$

Kopieren von Variablen

(cpx-in) $(\texttt{letrec}\ x = y, Env\ \texttt{in}\ \ C[x]) \longrightarrow (\texttt{letrec}\ x = y, Env\ \ \texttt{in}\ \ C[y])$
wobei y eine Variable und $x \neq y$.

(cpx-e) $(\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ \ t)$
$\longrightarrow (\texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ \ t)$
wobei y eine Variable und $x \neq y$.

Kopieren von Konstruktoren

(cpcx-in) $(\texttt{letrec}\ x_1 = c\ t_1 \ldots t_m, Env\ \texttt{in}\ \ C[x])$
$\longrightarrow (\texttt{letrec}\ x_1 = c\ y_1 \ldots y_m,$
$y_1 = t_1, \ldots, y_m = t_m, Env\ \texttt{in}\ \ C[c\ y_1 \ldots y_m])$

(cpcx-e) $(\texttt{letrec}\ x_1 = c\ t_1 \ldots t_m, z = C[x], Env\ \texttt{in}\ t)$
$\longrightarrow (\texttt{letrec}\ x_1 = c\ y_1 \ldots y_m,$
$y_1 = t_1, \ldots, y_m = t_m, z = C[c\ y_1 \ldots y_m], Env\ \ \texttt{in}\ t)$

Lambda-Lifting

(llift) $C[s[z]] \longrightarrow C[(\lambda x.\ s[x])\ z]$, wobei z eine Variable ist

Unique-Copy

(ucp-1) $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ \ S[x]) \longrightarrow (\texttt{letrec}\ Env\ \texttt{in}\ \ S[s])$
wenn x nur einmal in $Env, S[x]$ und nicht in s vorkommt.

(ucp-2) $(\texttt{letrec}\ x = s, Env, y = S[x]\ \texttt{in}\ \ t)$
$\longrightarrow (\texttt{letrec}\ Env, y = S[s]\ \texttt{in}\ \ t)$
wenn x nur einmal in $Env, S[x], t$ und nicht in s vorkommt.

Andere Transformationen

(xch) $(\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r) \longrightarrow (\texttt{letrec}\ y = t, x = y, Env\ \texttt{in}\ r)$

(betavar) $C[(\lambda x.\ s)\ y] \longrightarrow C[s[y/x]]$, wenn y eine Variable ist.

Abbildung 3.4: Weitere Programmtransformationen aus [Sch03a]

3.7 Vertauschungs- und Gabeldiagramme

Wir möchten im Folgenden weitere Programmtransformationen definieren und als korrekt beweisen, damit wir diese in späteren Kapiteln benutzen können.

Um die Korrektheit von Programmtransformationen zeigen können, benötigen wir das Hilfsmittel der Vertauschungs- und Gabeldiagramme, wie sie auch in [Sch03a] benutzt werden. Aufgrund dessen definieren wir diese nun.

Definition 3.35. (interne Reduktion) *Eine interne Reduktion ist jede Reduktion, die keine Normalordnungsreduktion ist. Wir bezeichnen eine solche Reduktion mit* $\xrightarrow{i}$ *oder auch mit* $\xrightarrow{iC}$.

Eine Reduktion ist R-intern, wenn sie in einem Reduktionskontext stattfindet – also der Redex in einem Reduktionskontext ist –, aber keine Normalordnungsreduktion ist. Wir bezeichnen eine solche Reduktion mit $\xrightarrow{iR}$.

Eine Reduktion ist S-intern, wenn sie in einem Oberflächenkontext stattfindet, aber keine Normalordnungsreduktion ist. Wir bezeichnen eine solche Reduktion mit $\xrightarrow{iS}$.

Definition 3.36. (Transformationsdiagramm für eine Reduktionsfolge)

Ein Transformationsdiagramm für eine Reduktionsfolge *ist eine Regel der Form:*

$$\xleftarrow{K_1,red_1,P_1} \cdot \xleftarrow{K_2,red_2,P_2} \cdots \xleftarrow{K_k,red_k,P_k} \quad \rightsquigarrow \quad \xleftarrow{K_{k+1},red_{k+1},P_{k+1}} \cdot \xleftarrow{K_{k+2},red_{k+2},P_{k+2}} \cdots \xleftarrow{K_l,red_l,P_l}$$

wobei für $1 \leq i \leq l$ *das Symbol* $\xleftarrow{K_i,red_i,P_i}$ *eine Reduktion darstellt, die durch vier Eigenschaften zu spezifizieren ist:*

- *Die Richtung der Reduktion, d.h. anstelle von* $\longleftarrow$ *muss entweder* $\longrightarrow$ *oder* $\longleftarrow$ *stehen.*
- K_i *muss einer der Werte* iC, iR, iS *oder* n *sein, je nachdem, ob die Reduktion eine interne Reduktion (in einem Kontext, Reduktions- oder Oberflächenkontext) oder eine Normalordnungsreduktion ist.*
- red_i *gibt den Namen der Reduktion an. Dieser Name kann jedoch auch eine Variable sein, wenn an dieser Stelle jede Reduktion des FUNDIO-Kalküls, oder eine Reduktion aus einer gesondert zu spezifizierenden Menge von Reduktionen stehen kann.*
- P_i *gibt das IO-Paar der Reduktion an, falls diese ein solches Paar besitzt. Auch hier ist eine Variable zulässig.*
- ○ *Optional kann statt einer Reduktion auch die transitive (bzw. transitiv-reflexive) Hülle einer Reduktion stehen.*

Werden identische Variablen für red_i (bzw. P_i) für verschiedene Reduktionen benutzt, so bedeutet dies, dass die gleiche Reduktion (bzw. das gleiche IO-Paar) an allen Stellen mit identischen Variablen benutzt werden muss.

Obiges Transformationsdiagramm ist auf eine Reduktionsfolge

$$s \xleftarrow{K_1,red_1,P_1} x_1 \xleftarrow{K_2,red_2,P_2} x_2 \ldots x_{k-1} \xleftarrow{K_k,red_k,K_k} t$$

anwendbar*, wenn gilt:*

$$\exists y_1 \ldots y_{l-1} : s \xleftarrow{K_{k+1},red_{k+1},P_{k+1}} y_1 \xleftarrow{K_{k+2},red_{k+2},P_{k+2}} y_2 \ldots y_{l-1} \xleftarrow{K_l,red_l,K_l} t$$

Definition 3.37. (Vertauschungs- und Gabeldiagramme)

Ein Vertauschungsdiagramm *für die Reduktion $\xrightarrow{iX,red}$, mit $iX \in \{iC, iR, iS\}$ ist ein Transformationsdiagramm der Form:*

$$\xrightarrow{iX,red,P_0} \cdot \xrightarrow{n,a_1,P_1} \ldots \xrightarrow{n,a_k,P_k} \quad \rightsquigarrow \quad \xrightarrow{n,b_1,P'_1} \ldots \xrightarrow{n,b_m,P'_m} \cdot \xrightarrow{iX,red_1,P''_1} \ldots \xrightarrow{iX,red_h,P''_h}$$

wobei $k \geq 0, m \geq 1$.

Ein Gabeldiagramm *für die Reduktion $\xrightarrow{iX,red}$, mit $iX \in \{iC, iR, iS\}$ ist ein Transformationsdiagramm der Form:*

$$\xleftarrow{n,a_1,P_0} \ldots \xleftarrow{n,a_k,P_k} \cdot \xrightarrow{iX,red,P_0} \quad \rightsquigarrow \quad \xrightarrow{iX,red_1,P'_1} \ldots \xrightarrow{iX,red_h,P'_h} \cdot \xleftarrow{n,b_1,P''_1} \ldots \xleftarrow{n,b_m,P''_m}$$

wobei $k \geq 0, m \geq 1$.

Abbildung 3.5 zeigt Darstellungen für die beiden Diagrammtypen. Dabei sind die Terme $t_1, \ldots, t_l$ die Terme einer bestehenden Reduktionsfolge, deren Pfeile durchgehend gezeichnet sind. Gestrichelte Pfeile stehen für die existenz-quantifizierten Reduktionen eines Transformationsdiagrammes auf Reduktionsfolgen, durch das Symbol $\circ$ sind die entsprechenden existenz-quantifizierten Terme dargestellt. Die Darstellungen zeigen die Anwendung eines Vertauschungsdiagramms auf einen Prefix und die Anwendung eines Gabeldiagramms auf einen Suffix einer jeweils gegebenen Reduktionsfolge.

Definition 3.38. (Vollständige Sätze von Vertauschungs- und Gabeldiagrammen)

Ein vollständiger Satz von Vertauschungsdiagrammen *für die Reduktion $\xrightarrow{iX,red}$ ist eine Menge von Vertauschungsdiagrammen für die Reduktion $\xrightarrow{iX,red}$, so dass für jede Reduktionsfolge $t_1 \xrightarrow{iX,red} t_2 \xrightarrow{n} \ldots \xrightarrow{n} t_l$, wobei t_l eine WHNF ist, mindestens eines der Diagramme auf einen Prefix der Reduktionsfolge anwendbar ist. Für den Fall $l = 2$ muss nur dann ein Diagramm enthalten sein, wenn t_1 keine WHNF ist.*

Ein vollständiger Satz von Gabeldiagrammen *für die Reduktion $\xrightarrow{iX,red}$ ist eine Menge von Gabeldiagrammen für die Reduktion $\xrightarrow{iX,red}$, so dass für jede Reduktionsfolge*

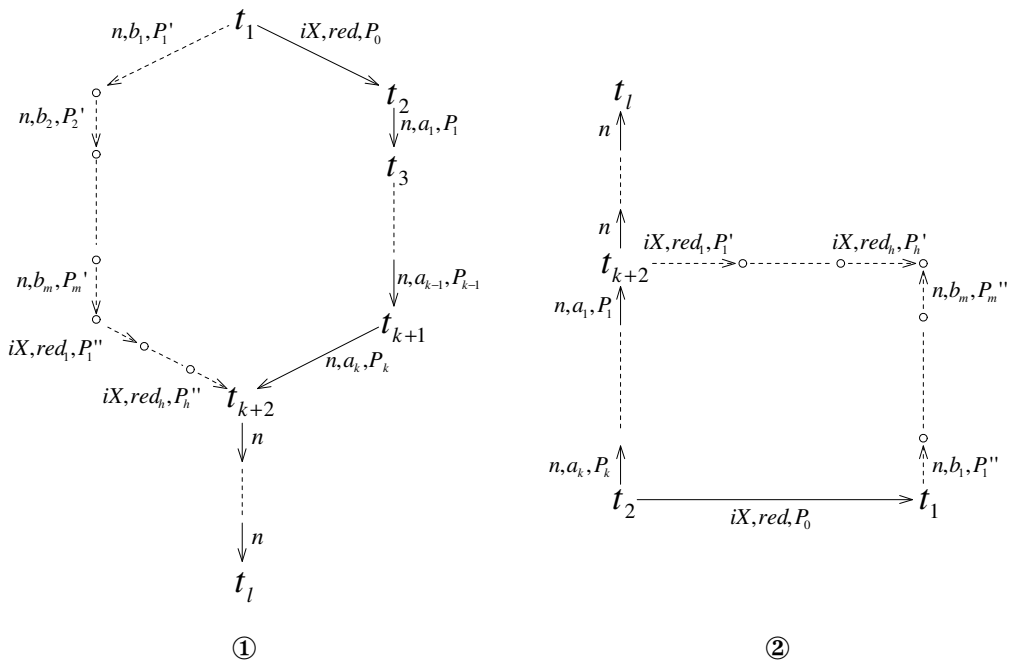

Abbildung 3.5: Darstellung des Vertauschungs- (①) und Gabeldiagramms (②).

$t_l \overset{n}{\leftarrow} \ldots t_3 \overset{n}{\leftarrow} t_2 \xrightarrow{iX,red} t_1$, *wobei* t_l *eine WHNF ist, mindestens eines der Diagramme auf einen Suffix der Reduktionsfolge anwendbar ist. Für den Fall* $l = 2$ *muss nur dann ein Diagramm enthalten sein, wenn* t_l *keine WHNF ist.*

Wir geben noch ein Lemma aus [Sch03a] an, dass wir später verwenden werden:

Lemma 3.39. *Es gilt:*

1. $R^-[(\texttt{letrec}\ Env\ \texttt{in}\ t)] \xrightarrow{n,(lll)^*} (\texttt{letrec}\ Env\ \texttt{in}\ R^-[t])$.

2. *Wenn* $R = [(\texttt{letrec}\ x_1 = R_1^-[\cdot], \ldots, x_j = R_{j-1}[x_{j-1}], Env_1\ \texttt{in}\ R^-[x_j])]$ *ein Reduktionskontext ist, dann gilt:*

$$\begin{aligned}&(\texttt{letrec}\ x_1 = R_1^-[(\texttt{letrec}\ Env_2\ \texttt{in}\ t)], \ldots, x_j = R_{j-1}[x_{j-1}], Env_1\ \texttt{in}\ R^-[x_j])\\ \xrightarrow{n,(lll)^*}\ &(\texttt{letrec}\ x_1 = R_1^-[t], \ldots, x_j = R_{j-1}[x_{j-1}], Env_1, Env_2\ \texttt{in}\ R^-[x_j])\end{aligned}$$

3. $(\texttt{letrec}\ Env_1\ \texttt{in}\ R^-[(\texttt{letrec}\ Env_2\ \texttt{in}\ t)]) \xrightarrow{n,(lll)^*} (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ R^-[t])$

Beweis. Siehe [Sch03a, Lemma 10.2]. □

3.8 Transformationen auf `case`-Ausdrücken

Wir definieren nun einige neue Programmtransformationen, die allesamt spezielle Umformungen von `case`-Ausdrücken darstellen.

Definition 3.40. *In Abbildung 3.6 werden die Transformationen* (capp)*,* (ccpcx)*,* (ccase)*,* (ccase-in)*,* (lcshift) *und* (crpl) *definiert.*

Mit der Regel (capp) können Anwendungen auf `case`-Ausdrücke nach innen verschoben werden. Die Regel ist analog zur (lapp)-Reduktion, die Anwendungsverschiebung für `letrec`-Ausdrücke durchführt.

Die (ccpcx)-Transformation erlaubt das Kopieren eines Patterns in die rechte Seite einer `case`-Alternativen, wenn das erste Argument des `case`-Ausdrucks eine Variable ist.

Mit der (lcshift)-Regel können äußere Bindungen in `case`-Alternativen verschoben werden, wobei der Ausdruck eine spezielle Form besitzen muss. Die Regel wird vor allem für den Beweis der (ccase-in)-Transformation benötigt.

Die Transformation (ccase) kann auf ineinander verschachtelte `case`-Ausdrücke angewendet werden und stellt die Reihenfolge der `case`-Ausdrücke gewissermaßen um. Die (ccase-in)-Regel ist eine erweiterte Variante der (ccase)-Regel, die wir später aufgrund dessen benötigen, dass verschachtelte `case`-Ausdrücke in der Kernsprache des GHC zu Ausdrücken in L_{FUNDIO} übersetzt werden, deren Form dem Ausdruck auf der linken Seite der (ccase-in)-Transformation entspricht.

Die (crpl)-Regel erlaubt es, rechte Seiten von `case`-Alternativen durch beliebige Ausdrücke zu ersetzen, wenn sichergestellt ist, dass diese Alternativen durch die Auswertung nicht erreicht werden.

In den folgenden Abschnitten beweisen wir die Korrektheit der soeben definierten Programmtransformationen.

3.8.1 Korrektheit von (capp)

Lemma 3.41. *Wenn $s \xrightarrow{iR,capp} t$, dann gilt: s ist in WHNF gdw. t ist in WHNF.*

Beweis. Nehmen wir an, dass s in WHNF ist. Zusätzlich gelte $s \xrightarrow{iR,capp} t$. Dies ist jedoch nicht möglich, da eine WHNF keine Applikation in einem Reduktionskontext besitzt.

Nehmen wir nun an, dass t in WHNF ist. Zusätzliche gelte $t \xleftarrow{iR,capp} s$, aber auch dies ist nicht möglich, da eine WHNF keinen `case`-Ausdruck in einem Reduktionskontext besitzt. □

(capp) $((\texttt{case}\ s\ (p_1 \to t_1)\ \ldots\ (p_N \to t_N))\ t)$
$\longrightarrow (\texttt{case}\ s\ (p_1 \to (t_1\ t))\ \ldots\ (p_N \to (t_N\ t)))$

(ccpcx) $(\texttt{case}\ x\ (p_1 \to t_1) \ldots ((c_i\ y_1 \ldots y_{ar(c_i)}) \to C[x]) \ldots (p_N \to t_N))$
$\longrightarrow (\texttt{case}\ x$
$(p_1 \to t_1) \ldots$
$((c_i\ y_1 \ldots y_{ar(c_i)}) \to C[(c_i\ y_1 \ldots y_{ar(c_i)})]) \ldots$
$(p_N \to t_N))$
wobei x eine Variable ist und $1 \leq i < N$.

(lcshift) $(\texttt{letrec}\ y = s, Env\ \texttt{in}\ R^-[(\texttt{case}\ y\ (p_1 \to t_1) \ldots (p_N \to t_N))])$
$\longrightarrow (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ s$
$(p_1 \to (\texttt{letrec}\ y = p_1\ \texttt{in}\ t_1))$
$\ldots$
$(p_N \to (\texttt{letrec}\ y = p_N\ \texttt{in}\ t_N)))])$
wenn y nicht frei in s, Env und R^- vorkommt.

(ccase) $(\texttt{case}\ (\texttt{case}\ s\ (p_1 \to t_1)\ \ldots\ (p_N \to t_N))\ Alts)$
$\longrightarrow (\texttt{case}\ s\ (p_1 \to (\texttt{case}\ t_1\ Alts))\ \ldots\ (p_N \to (\texttt{case}\ t_N\ Alts)))$

(ccase-in) $(\texttt{letrec}\ y = (\texttt{case}\ s\ (p_1 \to t_1) \ldots (p_N \to t_N))\ \texttt{in}\ (\texttt{case}\ y\ Alts))$
$\longrightarrow (\texttt{case}\ s$
$(p_1 \to (\texttt{letrec}\ y = t_1\ \texttt{in}\ (\texttt{case}\ y\ Alts)))$
$\ldots$
$(p_N \to (\texttt{letrec}\ y = t_N\ \texttt{in}\ (\texttt{case}\ y\ Alts))))$
wenn y nicht frei in $(\texttt{case}\ s\ (p_1 \to t_1) \ldots (p_N \to t_N))$vorkommt.

(crpl) $(\texttt{case}\ s\ (p_1 \to t_1) \ldots ((c_i\ y_1 \ldots y_{ar(c_i)}) \to t_i) \ldots (p_N \to t_N))$
$\longrightarrow (\texttt{case}\ s\ (p_1 \to t_1) \ldots ((c_i\ y_1 \ldots y_{ar(c_i)}) \to q) \ldots (p_N \to t_N))$
kann in einem Kontext C angewendet werden, wenn dieser Kontext die freien Variablen $\mathcal{FV}(s)$ nicht derart bindet, so dass s bezüglich C zu einer Konstruktoranwendung $(c_i\ a_1 \ldots a_n)$ auswerten kann. Dabei ist $1 \leq i < N$ und q ein beliebiger geschlossener Ausdruck.

Abbildung 3.6: case-Programmtransformationen

Lemma 3.42. *Ein vollständiger Satz von Vertauschungsdiagrammen für* $\xrightarrow{iR,capp}$ *ist:*

$$\xrightarrow{iR,capp} \cdot \xrightarrow{n,a,P} \quad \rightsquigarrow \quad \xrightarrow{n,a,P} \cdot \xrightarrow{iR,capp}$$

$$\xrightarrow{iR,capp} \cdot \xrightarrow{n,(lll)^+} \quad \rightsquigarrow \quad \xrightarrow{n,(lll)^+} \cdot \xrightarrow{iR,capp}$$

$$\xrightarrow{iR,capp} \cdot \xrightarrow{n,case} \quad \rightsquigarrow \quad \xrightarrow{n,case} \cdot \xrightarrow{n,lll}$$

Beweis. Da die Transformation (capp) keine Reduktionsregel des FUNDIO-Kalküls ist, ist jede Anwendung von (capp) in einem Reduktionskontext intern.

Sei $s = R[s'] = R[((\texttt{case}\ s_0\ (p_1 \to t_1) \ldots (p_N \to t_N))\ t_0)]$ und $t = R[t'] = R[(\texttt{case}\ s_0\ (p_1 \to (t_1\ t_0)) \ldots (p_N \to (t_N\ t_0)))]$, d.h. $s \xrightarrow{iR,capp} t$. Lemma 3.41 zeigt, dass es nicht möglich ist, dass s keine WHNF, aber t eine WHNF ist.

Aufgrund dessen genügt es zu zeigen, dass für jede Reduktionsfolge $s \xrightarrow{iR,capp} t \xrightarrow{n,a,P} r$ ein anwendbares Diagramm im Satz enthalten ist, oder die Reduktionsfolge mittels Normalordnungsreduktionen derart verlängert werden kann, so dass ein Diagramm anwendbar ist.

Wir unterscheiden nun danach, welche Normalordnungsreduktion auf t folgt.

I. Auf t folgt eine (n, a, P)-Reduktion mit $a \in \{\text{IOlet}, \text{lapp}, \text{IOr-c}, \text{lbeta}\}$.

Betrachten wir die entsprechenden Fälle 1.a), 1.b), 3.a) und 4.b) aus Definition 3.14, so zeigt sich, dass der maximale Reduktionskontext von t die Form $R_0[R_A^-[\cdot]]$ hat, wobei $R_A^-[t_A]$ der Normalordnungsredex sowie t_A ein `letrec`-Ausdruck, eine Konstante oder eine Abstraktion und R_A^- ein schwacher Reduktionskontext der Form $(\texttt{IO}\ [\cdot])$ oder $([\cdot]\ s_A)$ ist.

Da $t = R[t']$ und $t = R_0[R_A^-[t_A]]$ gibt es folgende Möglichkeiten für t:

i) $t = R[(\texttt{case}\ R_B^-[R_A^-[t_A]]\ (p_1 \to (t_1\ t_0)) \ldots)]$

ii) $t = (\texttt{letrec}\ x_1 = R_B^-[R_A^-[t_A]], x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env$
$\texttt{in}\ R_{j+1}^-[x_j])$
mit $R_i^-[x_{i-1}] = R_{i'}^-[(\texttt{case}\ R_D^-[x_{i-1}]\ (p_1 \to (t_1\ t_0)) \ldots)] = R_{i'}^-[t']$
für $2 \leq i \leq j+1$

Die (n, a, P)-Reduktion führt zu einem Ausdruck r, der sich von t darin unterscheidet, dass der Unterausdruck $R_A^-[t_A]$ durch einen anderen Ausdruck ersetzt wurde.

Betrachtet man für die Fälle i) und ii) den zugehörigen Ausdruck s, so zeigt sich, dass der Normalordnungsredex von s ebenfalls $R_A^-[t_A]$ ist und die gleiche (n, a, P)-Reduktion auch auf s anwendbar ist. Sei somit $s \xrightarrow{n,a,P} r'$. Die Ausdrücke r' und r können mittels einer $(iR, capp)$-Reduktion ineinander überführt

werden. Das bedeutet, dass die Reduktionsfolge $s \xrightarrow{iR,capp} t \xrightarrow{n,a,P} r$ durch die Folge $s \xrightarrow{n,a,P} r' \xrightarrow{iR,capp} r$ ersetzt werden kann.

Wir zeigen dies exemplarisch an einer (n,IO-let)-Reduktion in Fall i).

$$\begin{array}{rll}
 & R[((\texttt{case}\ R_A[(\texttt{IO}\ (\texttt{letrec}\ Env\ \texttt{in}\ s''))]\ (p_1 \to t_1)\ldots)\ t_0)] & (= s) \\
\xrightarrow{iR,capp} & R[(\texttt{case}\ R_A[(\texttt{IO}\ (\texttt{letrec}\ Env\ \texttt{in}\ s''))]\ (p_1 \to (t_1\ t_0))\ldots)] & (= t) \\
\xrightarrow{n,IOlet} & R[(\texttt{case}\ R_A[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{IO}\ s''))]\ (p_1 \to (t_1\ t_0))\ldots)] & (= r) \\
\hline
 & R[((\texttt{case}\ R_A[(\texttt{IO}\ (\texttt{letrec}\ Env\ \texttt{in}\ s''))]\ (p_1 \to t_1)\ldots)\ t_0)] & (= s) \\
\xrightarrow{n,IOlet} & R[((\texttt{case}\ R_A[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{IO}\ s''))]\ (p_1 \to t_1)\ldots)\ t_0)] & (= r') \\
\xrightarrow{iR,capp} & R[(\texttt{case}\ R_A[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{IO}\ s''))]\ (p_1 \to (t_1\ t_0))\ldots)] & (= r)
\end{array}$$

II. Auf t folgt eine (n, a, P)-Reduktion mit $a \in \{\text{IOr-in}, \text{cp-in}\}$.

Die zugehörigen Fälle 3.b) und 4.c) aus Definition 3.14 zeigen, dass t die Form $(\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_j = x_{j-1}, Env\ \texttt{in}\ R_A^-[x_j])$ hat, wobei der Normalordnungsredex t, t_A eine Konstante oder eine Abstraktion und R_A^- ein schwacher Reduktionskontext ist.

Da $t = R[t']$ gilt, hat t folgende Form, die auch das Aussehen von s spezifiziert:

$$\begin{array}{rl}
t = (\texttt{letrec}\ & x_1 = t_A, x_2 = x_1, \ldots, x_j = x_{j-1}, Env \\
& \texttt{in}\ R_B^-[(\texttt{case}\ R_C^-[x_j]\ (p_1 \to (t_1\ t_0))\ldots)]) \\
s = (\texttt{letrec}\ & x_1 = t_A, x_2 = x_1, \ldots, x_j = x_{j-1}, Env \\
& \texttt{in}\ R_B^-[((\texttt{case}\ R_C^-[x_j]\ (p_1 \to t_1)\ldots)\ t_0)])
\end{array}$$

Sei $s \xrightarrow{iR,capp} t \xrightarrow{n,a,P} r$, dann unterscheidet sich r nun darin von t, dass der Unterausdruck $R_C^-[x_j]$ durch einen Ausdruck t_c ersetzt wurde.

Wie die Form von s zeigt, ist auf s dieselbe (n, a, P)-Reduktion anwendbar und $R_C^-[x_j]$ wird durch den selben Ausdruck t_c ersetzt.

Insgesamt bedeutet dies, dass die Reduktionsfolge $s \xrightarrow{iR,capp} t \xrightarrow{n,a,P} r$ durch die Reduktionsfolge $s \xrightarrow{n,a,P} r' \xrightarrow{iR,capp} r$ ersetzt werden kann.

III. Auf t folgt eine (n, a, P)-Reduktion mit $a \in \{\text{IOr-e}, \text{cp-e}\}$.

Betrachten wir die zugehörigen Fälle 3.c), 4.d) und 4.e) aus Definition 3.14, so zeigt sich, dass t die Form $(\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1}, y_1 = R_A^-[R_B^-[x_j]], Env\ \texttt{in}\ t'')$ hat, wobei y_1 in einem Reduktionskontext ist.

Die Bedingung für y_1 können wir auch ausdrücken, indem wir die Form von t weiter spezifizieren: $t = (\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1}, y_1 = R_A^-[R_B^-[x_j]], y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}], Env\ \texttt{in}\ R_C^-[y_j])$. Zusätzlich ist t der Normalordnungsredex, t_A eine Konstante oder eine Abstraktion und R_B^- ist von der Form $(\texttt{IO}\ [\cdot])$, $([\cdot]\ s_A)$ oder $(\texttt{case}\ [\cdot]\ Alts)$.

Da $t = R[t']$, gibt es folgende Möglichkeiten für t':

i) $t = (\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1},$
$\qquad y_1 = R_D^-[(\texttt{case}\ R_E^-[R_A^-[x_m]]\ (p_1 \to (t_1\ t_0))\ldots)], \ldots$
$\quad \texttt{in}\ R_C^-[y_j])$

ii) $t = (\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1},$
$\qquad y_1 = R_B^-[(\texttt{case}\ x_m\ (p_1 \to (t_1\ t_0))\ldots)], \ldots$
$\quad \texttt{in}\ R_C^-[y_j])$

iii) $t = (\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1},$
$\qquad y_1 = R_B^-[R_A^-[x_m]], y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]$
$\quad \texttt{in}\ R_{j+1}^-[y_j])$

mit $y_i = R_{i'}^-[t'] = R_{i'}^-[(\texttt{case}\ R_E^-[y_{i-1}]\ (p_1 \to (t_1\ t_0))\ldots)]$ für $2 \le i \le j+1$

Wird nun die (n, a, P)-Reduktion auf t angewendet, so erhalten wir einen Ausdruck r, der sich nur dadurch von t unterscheidet, dass der Term $R_A^-[x_j]$, bzw. für Fall ii) nur die Variable x_m, durch einen Term t_c ersetzt wurde. Wiederum zeigt sich, dass dasselbe für s gilt und die Reduktionsfolge $s \xrightarrow{iR,capp} t \xrightarrow{n,a,P} r$ durch die Folge $s \xrightarrow{n,a,P} r' \xrightarrow{iR,capp} r$ ersetzt werden kann.

Wir veranschaulichen dies anhand einer (n,IOr-e)-Reduktion in Fall i).

$$
\begin{aligned}
& (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, && (= s)\\
&\qquad y_1 = R_D^-[((\texttt{case}\ R_E^-[(\texttt{IO}\ x_m)]\ (p_1 \to t_1)\ldots)\ t_0)],\\
&\qquad y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]\\
&\quad \texttt{in}\ R_C^-[y_j])\\
\xrightarrow{iR,capp}\ & (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, && (= t)\\
&\qquad y_1 = R_D^-[(\texttt{case}\ R_E^-[(\texttt{IO}\ x_m)]\ (p_1 \to (t_1\ t_0))\ldots)],\\
&\qquad y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]\\
&\quad \texttt{in}\ R_C^-[y_j])\\
\xrightarrow{n,IOr-e,(c,d)}\ & (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, && (= r)\\
&\qquad y_1 = R_D^-[(\texttt{case}\ R_E^-[d]\ (p_1 \to (t_1\ t_0))\ldots)],\\
&\qquad y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]\\
&\quad \texttt{in}\ R_C^-[y_j])\\
\hline
& (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, && (= s)\\
&\qquad y_1 = R_D^-[((\texttt{case}\ R_E^-[(\texttt{IO}\ x_m)]\ (p_1 \to t_1)\ldots)\ t_0)],\\
&\qquad y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]\\
&\quad \texttt{in}\ R_C^-[y_j])
\end{aligned}
$$

$$\xrightarrow{n,IOr-e,(c,d)} (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= r')$$
$$y_1 = R_D^-[((\texttt{case}\ R_E^-[d]\ (p_1 \to t_1)\ldots)\ t_0)],$$
$$y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]$$
$$\texttt{in}\ R_C^-[y_j])$$
$$\xrightarrow{iR,capp} (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= r)$$
$$y_1 = R_D^-[(\texttt{case}\ R_E^-[d]\ (p_1 \to t_1\ (t_0))\ldots)],$$
$$y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]$$
$$\texttt{in}\ R_C^-[y_j])$$

IV. Auf t folgt eine (n,llet)-Reduktion.

Die Reduktion kann keine (llet-in)-Reduktion sein, denn in diesem Fall, d.h. Fall 1.d) aus Definition 3.14, hätte t die Form ($\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t'')$). Dies ist jedoch nicht möglich, denn der `case`-Ausdruck t' kann dann nicht in einem Reduktionskontext stehen, d.h. $t \neq R[t']$.

Betrachten wir nun den Fall einer (n,llet-e)-Reduktion, d.h. Fall 1.e) aus Definition 3.14. Dann hat t die Form $t = (\texttt{letrec}\ x = (\texttt{letrec}\ Env'\ \texttt{in}\ t_A), x_2 = R_2^-[x], \ldots, x_j = R_j^-[x_{j-1}], Env\ \texttt{in}\ R_0^-[x_j])$. Die ($n$,llet-e)-Reduktion führt dann zu einem Ausdruck $r = (\texttt{letrec}\ x = t_A, x_2 = R_2^-[x], \ldots, x_j = R_j^-[x_{j-1}], Env, Env'\ \texttt{in}\ R_0^-[x_j]))$. Die Reduktionskontexte, in denen der Ausdruck t' in t stehen kann, sind die gleichen an denen s' im Ausdruck s stehen kann.

Wiederum ist die Reduktionsfolge $s \xrightarrow{iR,capp} t \xrightarrow{n,llet} r$ durch die Folge $s \xrightarrow{n,llet} r' \xrightarrow{iR,capp} r$ austauschbar.

V. Auf t folgt eine (n,lcase)-Reduktion.

Dann hat t die Form $R_0[t_A]$ mit $t_A = (\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ Alts)$. Nun ist zu unterscheiden, an welcher Position t' in t steht, wobei jedoch $t = R[t']$ gelten muss. Die Fälle mit $t' \neq t_A$ sind den Fällen aus I. sehr ähnlich und es zeigt sich, dass stets das erste Diagramm anwendbar ist. Es bleibt also der Fall $t' = t_A$. Hierbei kann nun nach der Struktur von R_0 folgendermaßen unterschieden werden:

i) R_0 ist ein schwacher Reduktionskontext R_A^-, d.h. $t = R_A^-[t']$

ii) $R_0 = (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[\cdot])$.

iii) $R_0 = (\texttt{letrec}\ x_1 = R_1^-[\cdot], x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env\ \texttt{in}\ R_A^-[x_j])$.

In allen drei Fällen ist das zweite Diagramm anwendbar, wobei Lemma 3.39 zeigt, dass die Normalordnungsreduktion dazu führt, dass der `letrec`-Ausdruck innerhalb von t' bzw. s' eliminiert wird und damit die (capp)-Transformation auch dann in einem Reduktionskontext anwendbar ist, wenn auf s zunächst die (n,lcase)-Reduktion angewendet wird. Wir illustrieren dies

an einem Beispiel für den dritten Fall:

$$
\begin{array}{rl}
& (\texttt{letrec}\ x_1 = R_1^-[((\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ (p_1 \to t_1)\ldots)\ t_0)], \qquad (= s)\\
& \qquad\qquad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env\\
& \quad \texttt{in}\ R_A^-[x_j])\\
\xrightarrow{iR,capp} & (\texttt{letrec}\ x_1 = R_1^-[(\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ (p_1 \to (t_1\ t_0))\ldots)], \qquad (= t)\\
& \qquad\qquad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env\\
& \quad \texttt{in}\ R_A^-[x_j])\\
\xrightarrow{n,lcase} & (\texttt{letrec}\ x_1 = R_1^-[(\texttt{letrec}\ Env'\ \texttt{in}\ (\texttt{case}\ t_B\ (p_1 \to (t_1\ t_0))\ldots))],\\
& \qquad\qquad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env\\
& \quad \texttt{in}\ R_A^-[x_j])\\
\xrightarrow{n,(lll)^*} & (\texttt{letrec}\ x_1 = R_1^-[(\texttt{case}\ t_B\ (p_1 \to (t_1\ t_0))\ldots)],\\
& \qquad\qquad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env, Env'\\
& \quad \texttt{in}\ R_A^-[x_j])\\
\hline
& (\texttt{letrec}\ x_1 = R_1^-[((\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ (p_1 \to t_1)\ldots)\ t_0)], \qquad (= s)\\
& \qquad\qquad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env\\
& \quad \texttt{in}\ R_A^-[x_j])\\
\xrightarrow{n,lcase} & (\texttt{letrec}\ x_1 = R_1^-[((\texttt{letrec}\ Env'\ \texttt{in}\ (\texttt{case}\ t_B\ (p_1 \to t_1)\ldots))\ t_0)],\\
& \qquad\qquad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env\\
& \quad \texttt{in}\ R_A^-[x_j])\\
\xrightarrow{n,(lll)^*} & (\texttt{letrec}\ x_1 = R_1^-[((\texttt{case}\ t_B\ (p_1 \to t_1)\ldots)\ t_0)],\\
& \qquad\qquad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env, Env'\\
& \quad \texttt{in}\ R_A^-[x_j])\\
\xrightarrow{iR,capp} & (\texttt{letrec}\ x_1 = R_1^-[(\texttt{case}\ t_B\ (p_1 \to (t_1\ t_0))\ldots)],\\
& \quad x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env, Env'\\
& \quad \texttt{in}\ R_A^-[x_j])
\end{array}
$$

VI. Auf t folgt eine (n,case)-Reduktion.

Betrachten wir die Fälle 2.a), 2.b), 2.c) und 4.1) aus Definition 3.14, so kann t je nach Art der case-Reduktion folgende Formen haben:

- Im Falle einer (n,case-c)- oder (n,case-lam)-Reduktion:

$$t = R_0[(\texttt{case}\ t_A\ Alts)],$$

  wobei t_A eine Konstruktoranwendung oder eine Abstraktion ist.

- Im Falle einer (n,case-in)-Reduktion:

$$t = (\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[(\texttt{case}\ x_m\ Alts)]),$$

  wobei t_A eine Konstruktoranwendung ist.

- Im Falle einer (n,case-e)-Reduktion:

$$\begin{aligned} t = (&\texttt{letrec } x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1}, \\ &y_1 = R_0^-[(\texttt{case } x_m \ Alts)], y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env \\ &\texttt{in } R_A^-[y_j]), \end{aligned}$$

  wobei t_A eine Konstruktoranwendung ist.

Für alle drei Fälle kann t' so positioniert sein, dass das erste Diagramm angewendet werden kann. Dies ist immer dann der Fall, wenn $t' \neq (\texttt{case } t_A \ Alts)$ bzw. $t' \neq (\texttt{case } x_m \ Alts)$. Diese Fälle sind analog zu den Fällen aus den Punkten I.,II., und III. und werden deshalb nicht weiter betrachtet.

Sei also $t' = (\texttt{case } t_A \ Alts)$. Dann wird durch die Anwendung der (n,case)-Reduktion der Ausdruck t' durch einen Ausdruck t_c ersetzt, und bei der (case-in)- und (case-e)- Reduktion zusätzlich die `letrec`-Umgebung verändert. t_c hat in allen Fällen die Form $(\texttt{letrec } Env' \texttt{ in } (t_i \ t_0))$. Wendet man die ($n$,case)-Reduktion auf s an, so erhält man anstelle von t_c einen Ausdruck der Form $((\texttt{letrec } Env' \texttt{ in } t_i) \ t_0)$, der mittels einer weiteren (n,lapp)-Reduktion in t_c überführt werden kann.

Wir zeigen dies anhand zweier Beispiele:

Eine (n,case-lam)-Reduktion:

$$\begin{aligned} &\texttt{letrec } Env \texttt{ in } R_A^-[((\texttt{case } (\lambda v.w) \ (p_1 \to t_1) \ldots (\texttt{lambda} \to t_N)) \ t_0)] && (= s) \\ \xrightarrow{iR,capp} \ &\texttt{letrec } Env \texttt{ in } R_A^-[(\texttt{case } (\lambda v.w) \ (p_1 \to (t_1 \ t_0)) \ldots (\texttt{lambda} \to (t_N \ t_0)))] && (= t) \\ \xrightarrow{n,case-lam} \ &\texttt{letrec } Env \texttt{ in } R_A^-[(\texttt{letrec } \{\} \texttt{ in } (t_N \ t_0))] \end{aligned}$$

$$\begin{aligned} &\texttt{letrec } Env \texttt{ in } R_A^-[((\texttt{case } (\lambda v.w) \ (p_1 \to t_1) \ldots (\texttt{lambda} \to t_N)) \ t_0)] && (= s) \\ \xrightarrow{n,case-lam} \ &\texttt{letrec } Env \texttt{ in } R_A^-[((\texttt{letrec } \{\} \texttt{ in } t_N) \ t_0)] \\ \xrightarrow{n,lapp} \ &\texttt{letrec } Env \texttt{ in } R_A^-[(\texttt{letrec } \{\} \texttt{ in } (t_N \ t_0))] \end{aligned}$$

Eine (n,case-in)-Reduktion:

$$\begin{aligned} &(\texttt{letrec } x_1 = (c_i \ a_1 \ldots \ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ &\quad \texttt{in } R_0^-[((\texttt{case } x_m \ \ldots ((c_i \ z_1 \ldots z_n) \to t_i) \ldots) \ t_0)]) && (= s) \\ \xrightarrow{iR,capp} \ &(\texttt{letrec } x_1 = (c_i \ a_1 \ldots \ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ &\quad \texttt{in } R_0^-[(\texttt{case } x_m \ \ldots ((c_i \ z_1 \ldots z_n) \to (t_i \ t_0)) \ldots)]) && (= t) \\ \xrightarrow{n,case} \ &(\texttt{letrec } x_1 = (c_i \ y_1 \ldots \ y_n), y_1 = a_1, \ldots, y_n = a_n, x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ &\quad \texttt{in } R_0^-[(\texttt{letrec } z_1 = y_1, \ldots, z_n = y_n \texttt{ in } (t_i \ t_0))]) \end{aligned}$$

$$\begin{aligned} &(\texttt{letrec } x_1 = (c_i \ a_1 \ldots \ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \\ &\quad \texttt{in } R_0^-[((\texttt{case } x_m \ \ldots ((c_i \ z_1 \ldots z_n) \to t_i) \ldots) \ t_0)]) && (= s) \end{aligned}$$

$$\xrightarrow{n,case} (\texttt{letrec}\ x_1 = (c_i\ y_1 \dots\ y_n), y_1 = a_1, \dots, y_n = a_n, x_2 = x_1, \dots, x_m = x_{m-1}, Env \\ \texttt{in}\ R_0^-[((\texttt{letrec}\ z_1 = y_1, \dots, z_n = y_n\ \texttt{in}\ t_i)\ t_0)])$$
$$\xrightarrow{n,lapp} (\texttt{letrec}\ x_1 = (c_i\ y_1 \dots\ y_n), y_1 = a_1, \dots, y_n = a_n, x_2 = x_1, \dots, x_m = x_{m-1}, Env \\ \texttt{in}\ R_0^-[(\texttt{letrec}\ z_1 = y_1, \dots, z_n = y_n\ \texttt{in}\ (t_i\ t_0))])$$

□

Lemma 3.43. *Ein vollständiger Satz von Gabeldiagrammen für* $\xrightarrow{iR,capp}$ *ist:*

$$\xleftarrow{n,a,P} \cdot \xrightarrow{iR,capp} \rightsquigarrow \xrightarrow{iR,capp} \cdot \xleftarrow{n,a,P}$$
$$\xleftarrow{n,(lll)^+} \cdot \xrightarrow{iR,capp} \rightsquigarrow \xrightarrow{iR,capp} \cdot \xleftarrow{n,(lll)^+}$$
$$\xleftarrow{n,lll} \cdot \xleftarrow{n,case} \cdot \xrightarrow{iR,capp} \rightsquigarrow \xleftarrow{n,case}$$

Beweis. Die Gabeldiagramme können mithilfe der Vertauschungsdiagramme gefunden werden. Überprüft man die gleichen Fälle wie im Beweis zu Lemma 3.42, so zeigt sich die Vollständigkeit des Satzes. □

Lemma 3.44. *Die Reduktion* (capp) *kann nur endlich oft angewendet werden.*

Beweis. Sei $\mathcal{M}_{\mathbb{N}_0 \times \mathbb{N}_0}$ das System der Multimengen über $(\mathbb{N}_0 \times \mathbb{N}_0)$ wie es in [Ave95, Definition 1.3.7] definiert wird.

Wir betrachten das Termmaß $mo : L_{FUNDIO} \mapsto \mathcal{M}_{\mathbb{N}_0 \times \mathbb{N}_0}$, dass wir wie folgt definieren

$$mo(t) = \{(rch(s), ad(s, C)) \mid t = C[s] \text{ und } s = (\texttt{case}\ s'\ Alts)\}$$

wobei $rch : L_{FUNDIO} \mapsto \mathbb{N}_0$ und $ad : (L_{FUNDIO} \times C) \mapsto \mathbb{N}_0$ wie folgt definiert sind:

$$\begin{aligned} rch(\texttt{case}\ s\ (p_1 \to t_1) \dots (p_n \to t_N)) &= 1 + \max\{rch(t_i) \mid 1 \le i \le N\} \\ rch(\lambda x.s) &= rch(s) \\ rch((s\ t)) &= rch(s) \\ rch(c_i\ a_1\ \dots\ a_n) &= \max\{rch(a_i) \mid 1 \le i \le n\} \\ rch(\texttt{letrec}\ Env\ \texttt{in}\ s) &= rch(s) \\ rch(c_i) &= 0, \text{ falls } c_i \text{ eine Konstante ist.} \\ rch(x) &= 0, \text{ falls } x \text{ eine Variable ist.} \end{aligned}$$

$$ad(s, C) = \begin{cases} 1 + ad((s\ t), C'), & \text{falls } C[\cdot] = C'[([\cdot]\ t)] \\ 0, & \text{sonst} \end{cases}$$

Informell zählt rch die „rechts-`case`-Höhe“ und ad die Anzahl an Applikationen die direkt auf einen `case`-Ausdruck folgen.

Sei $s \xrightarrow{capp} t$. Die auf der lexikographischen Ordnung von Paaren aus $(\mathbb{N}_0 \times \mathbb{N}_0)$ basierende Multimengenordnung auf dem System $\mathcal{M}_{\mathbb{N}_0 \times \mathbb{N}_0}$ ist nach [Ave95, Satz 1.3.11] wohlfundiert. Deshalb genügt es zu zeigen, dass $mo(s)$ echt größer als $mo(t)$ bzgl. dieser Ordnung ist.

Sei $S = mo(s) \setminus (mo(s) \cap mo(t))$ und $T = mo(t) \setminus (mo(s) \cap mo(t))$. Es genügt zu zeigen (siehe [Ave95, Seite 28]):

$$\begin{aligned} &mo(s) \neq mo(t) \text{ und} \\ &\forall (r,a) \in T\ \exists (r',a') \in S : (r',a') > (r,a) \end{aligned}$$

$$\begin{aligned} &\text{Sei } s = C[(s_{case}\ t_0)] = C[((\texttt{case}\ s_0\ (p_1 \to t_1) \ldots (p_N \to t_N))\ t_0)] \\ &\text{dann ist } t = C[(\texttt{case}\ s_0\ (p_1 \to (t_1\ t_0)) \ldots (p_N \to (t_N\ t_0)))] \end{aligned}$$

S und T enthalten höchstens Paare für den Ausdurck s_{case} und für $i = 1, \ldots, N$ für t_i, falls $t_i = (\ldots (\texttt{case}\ t_i'\ Alts_i)\ a_1) \ldots\ a_n)$, denn andere case-Ausdrücke sind in $mo(s) \cap mo(t)$ enthalten.

Somit folgt: S enthält ein Paar $(rch(s_{case}), ad(s_{case}, C[[\cdot]\ t_0])) := (x, y)$, $x \geq 1$, sowie für $1 \leq i \leq N$ jeweils ein Paar $(x-1, z_i)$, falls $t_i = (\ldots (\texttt{case}\ t_i'\ Alts_i)\ a_1) \ldots a_{z_i})$.

Sei also $S = \{(x,y), (x-1,z_1), (x-1,z_2), \ldots (x-1,z_m)\}$ für $0 \leq m \leq N$.

Dann ist $T = \{(x, y-1), (x-1, z_1+1), (x-1, z_2+1), \ldots (x-1, z_m+1)\}$ für $0 \leq m \leq N$.

Für jedes Paar $(u,v) \in T$ gilt, dass $(x,y) > (u,v)$ und somit folgt, dass eine (capp)-Reduktion das wohlfundierte Maß mo echt verkleinert und damit folgt die Behauptung.

□

Satz 3.45. *Die Regel* (capp) *ist eine korrekte Programmtransformation.*

Beweis. Seien s', t' beliebige Ausdrücke, aufgrund von Lemma 3.29 genügt es zu zeigen:

$$\begin{aligned} &\text{Wenn } s' \xrightarrow{capp} t', \text{ dann} \\ &\qquad \forall R, \overrightarrow{P} : R[s']{\downarrow}(\overrightarrow{P}) \Leftrightarrow R[t']{\downarrow}(\overrightarrow{P}) \text{ und} \\ &\qquad \forall R, \overrightarrow{P} : \overrightarrow{P} \text{ ist gültig für } R[s'] \Leftrightarrow \overrightarrow{P} \text{ ist gültig für } R[t'] \end{aligned}$$

Mit $s = R[s']$ und $t = R[t']$ für einen beliebigen Reduktionskontext R bedeutet dies,

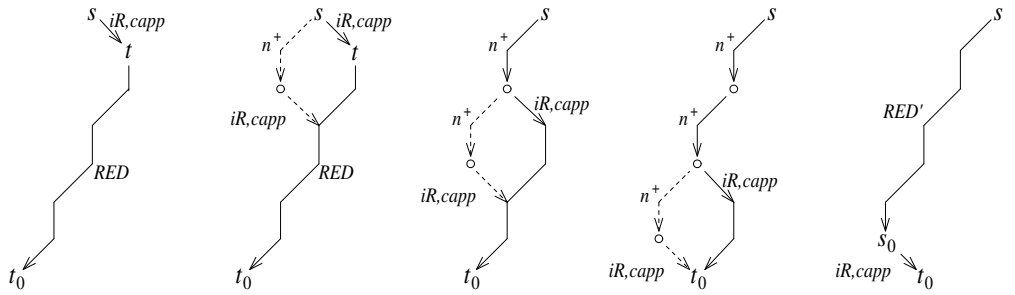

Abbildung 3.7: Konstruktion von RED', wenn $s \xrightarrow{iR,capp} t \xrightarrow{RED} t_0$ mit t_0 WHNF

es genügt zu zeigen:

Wenn $s \xrightarrow{iR,capp} t$, dann

(1) $\forall \overrightarrow{P} : s{\downarrow}(\overrightarrow{P}) \Leftrightarrow t{\downarrow}(\overrightarrow{P})$

(2) $\forall \overrightarrow{P} : \overrightarrow{P}$ ist gültig für $s \Leftrightarrow \overrightarrow{P}$ ist gültig für r

Wir zeigen zuerst (1):

Betrachten wir zunächst den Fall, dass $t{\downarrow}(\overrightarrow{P})$. Sei $\xrightarrow{RED}$ eine Folge von Normalordnungsreduktionen, so dass $t \xrightarrow{RED} t_0$, wobei t_0 eine WHNF ist, und $IOS(\xrightarrow{RED})=\overrightarrow{P'}$, wobei $\overrightarrow{P'}$ ein Prefix von $\overrightarrow{P}$ ist. Transformiere nun die Reduktionsfolge $s \xrightarrow{iR,capp} t \xrightarrow{RED} t_0$ mithilfe des vollständigen Satzes an Vertauschungsdiagrammen in eine Reduktionsfolge $s \xrightarrow{RED'} s_0 \xrightarrow{(iR,capp)^{0\vee 1}} t_0$, so dass $IOS(\xrightarrow{RED'}) = \overrightarrow{P'}$ und $\xrightarrow{RED'}$ nur aus Normalordnungsreduktionen besteht.
Abbildung 3.7 illustriert den Transformationsprozess. Die IO-Sequenz $\overrightarrow{P'}$ bleibt dabei erhalten, denn keines der Vertauschungsdiagramme entfernt oder fügt IOr-Reduktionen hinzu und die Reihenfolge von IOr-Reduktionen wird nicht vertauscht. Für die Transformation sind nur endlich viele Schritte notwendig, da pro Reduktion in $\xrightarrow{RED}$ höchstens ein Transformationsschritt notwendig ist. Mithilfe von Lemma 3.41 folgt, dass s_0 eine WHNF ist.

Betrachten wir nun den Fall, dass $s{\downarrow}(\overrightarrow{P})$. Sei $\xrightarrow{RED}$ nun eine Folge von Normalordnungsreduktionen, so dass $s \xrightarrow{RED} s_0$, wobei s_0 eine WHNF ist, und $IOS(\xrightarrow{RED}) = \overrightarrow{P'}$, wobei $\overrightarrow{P'}$ ein Prefix von $\overrightarrow{P}$ ist. Transformiere nun die Reduktionsfolge $s_0 \xleftarrow{RED} s \xrightarrow{iR,capp} t$ mithilfe des vollständigen Satzes an Gabeldiagrammen in eine Reduktionsfolge $s_0 \xrightarrow{(iR,capp)^{0\vee 1}} t_0 \xleftarrow{RED'} t$, wobei $\xrightarrow{RED'}$ nur aus Normalordnungsreduktionen besteht und $IOS(\xrightarrow{RED'}) = \overrightarrow{P'}$.
Abbildung 3.8 illustriert diese Transformation. Wiederum sind nur endlich viele Transformationsschritte notwendig und mit Lemma 3.41 folgt, dass t_0 eine WHNF

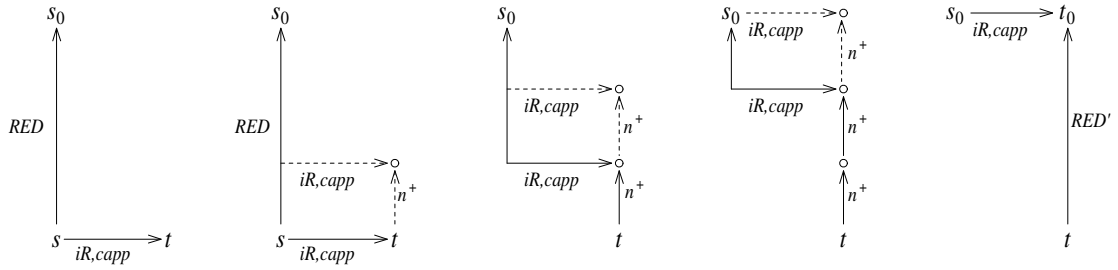

Abbildung 3.8: Konstruktion von RED', wenn $t \xleftarrow{iR,capp} s \xrightarrow{RED} s_0$ mit s_0 WHNF

ist.

Nun zeigen wir (2): Nehmen wir zunächst an $\overrightarrow{P}$ sei gültig für s. Sei RED die no-Reduktionsfolge von s entlang $\overrightarrow{P}$ Dann können zwei Fälle auftreten:

1. RED endet mit einer WHNF und benutzt dabei die IO-Sequenz $\overrightarrow{P'}$, die ein Prefix von $\overrightarrow{P}$ ist.

   In diesem Fall konstruieren wir, wie oben beschrieben, die Reduktionssequenz RED' für t, die ebenfalls mit einer WHNF endet und ebenfalls die IO-Sequenz $\overrightarrow{P'}$ benutzt. Somit folgt, dass $\overrightarrow{P}$ gültig für t ist.

2. RED hat $\overrightarrow{P}$ als IO-Sequenz, aber endet nicht mit einer WHNF.

   Sei nun RED_P der kürzeste Prefix von RED mit $IOS(RED_P) = \overrightarrow{P}$.

   Aufgrund von Definition 3.21 ist $\overrightarrow{P}$ endlich und somit ist auch RED_P endlich. Konstruiere nun mit dem vollständigen Satz an Gabeldiagrammen beginnend mit der Reduktionsfolge $\xleftarrow{RED_P} s \xrightarrow{iR,capp} t$ die Reduktionsfolge $\xrightarrow{iR,capp} . \xleftarrow{RED'_P} t$. Da die Diagramme keine IOr-Reduktionen vertauschen, hinzufügen oder eliminieren, gilt $IOS(RED'_P) = \overrightarrow{P}$ und somit ist $\overrightarrow{P}$ auch gültig für t.

Die andere Richtung, d.h. $\overrightarrow{P}$ ist gültig für $t \implies \overrightarrow{P}$ ist gültig für s, kann analog mithilfe der Vertauschungsdiagramme gezeigt werden. □

3.8.2 Korrektheit von (ccpcx)

Lemma 3.46. *Wenn $s \xrightarrow{iR,ccpcx} t$, dann gilt: s ist in WHNF gdw. t ist in WHNF.*

Beweis. Die Aussage gilt, da eine WHNF keinen `case`-Ausdruck in einem Reduktionskontext besitzt. □

Lemma 3.47. *Ein vollständiger Satz von Vertauschungsdiagrammen für* $\xrightarrow{iR,ccpcx}$ *ist:*

$$\xrightarrow{iR,ccpcx} \cdot \xrightarrow{n,a,P} \quad \rightsquigarrow \quad \xrightarrow{n,a,P} \cdot \xrightarrow{iR,ccpcx}$$

$$\xrightarrow{iR,ccpcx} \cdot \xrightarrow{n,case} \quad \rightsquigarrow \quad \xrightarrow{n,case}$$

$$\xrightarrow{iR,ccpcx} \cdot \xrightarrow{n,cp} \cdot \xrightarrow{n,case} \quad \rightsquigarrow \quad \xrightarrow{n,cp} \cdot \xrightarrow{n,case}$$

$$\xrightarrow{iR,ccpcx} \cdot \xrightarrow{n,case} \cdot \xrightarrow{n,(lll)^*} \quad \rightsquigarrow \quad \xrightarrow{n,case} \cdot \xrightarrow{n,(lll)^*} \cdot \xrightarrow{iR,(cpx)^*} \cdot \xrightarrow{iR,cpcx} \cdot \xrightarrow{iR,(cpx)^*} \cdot \xrightarrow{iR,(xch)^*} \cdot \xrightarrow{iR,(cpx)^*} \cdot \xrightarrow{iR,(xch)^*} \cdot \xrightarrow{iR,(gc)^*}$$

Beweis. Jede Anwendung von (ccpcx) in einem Reduktionskontext ist intern.

Sei x eine Variable, $s = R[s'] = R[(\texttt{case}\ x\ \ldots(p_i \to C[x])\ldots)]$ und $t = R[t'] = R[(\texttt{case}\ x\ \ldots(p_i \to C[p_i]\ldots))]$, d.h. $s \xrightarrow{iR,ccpcx} t$. Da es nicht möglich ist, dass s keine WHNF und t eine WHNF ist, wie Lemma 3.46 zeigt, genügt es zu zeigen, dass für jede Reduktionsfolge $s \xrightarrow{iR,ccpcx} t \xrightarrow{n,a,P} r$ ein anwendbares Diagramm im Satz enthalten ist, oder die Reduktionsfolge mittels Normalordnungsreduktionen so verlängert werden kann, dass ein Diagramm anwendbar ist.

Wir unterscheiden danach, welche Normalordnungsreduktion auf t folgt.

I. Auf t kann keine (n,llet-in)- oder (n,IOr-in,P)-Reduktion folgen.

   Dies zeigt sich, wenn man die zugehörigen Fälle 1.d) und 3.b) aus Definition 3.14 betrachtet. t müsste die Form ($\texttt{letrec}\ Env_A\ \texttt{in}\ (\texttt{letrec}\ Env_B\ \texttt{in}\ t_B)$) oder ($\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ R_0^-[(\texttt{IO}\ x_m)]$) haben. Dann kann jedoch $t = R[t']$ nicht gelten.

II. Auf t folgt eine (n, a, P)-Reduktion mit $a \in \{$IOlet, lapp, lcase, case-c, IOr-c, case-lam, lbeta$\}$.

   Die zugehörigen Fälle 1.a), 1.b), 1.c), 2.a), 3.a), 4.a) und 4.b) aus Definition 3.14 zeigen, dass t die Form $R_0[R_A^-[t_A]]$ hat, wobei der Normalordnungsredex $R_A^-[t_A]$, R_A^- von der Form ($\texttt{IO}\ [\cdot]$), ($[\cdot]\ t_B$) oder ($\texttt{case}\ [\cdot]\ Alts$) und t_A ein `letrec`-Ausdruck, eine Konstruktoranwendung, eine Konstante oder eine Abstraktion ist.

   Da $t = R[t']$, gibt es folgende Möglichkeiten für t':

   i) t' kann aufgrund der spezifischen Formen von R_A^- und t_A nicht innerhalb von $R_A^-[t_A]$ liegen.

   ii) $t = (\texttt{letrec}\ x_1 = R_C^-[R_A^-[t_A]], x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env$
   $\texttt{in}\ R_{j+1}^-[x_j])$
   und $R_i^-[x_{i-1}] = R_{i'}^-[(\texttt{case}\ [x_{i-1}]\ Alts)]$ für $2 \le i \le j+1$.

Sei $t \xrightarrow{n,a,P} r$, dann unterscheidet sich r dadurch von t, dass der Unterausdruck $R_A^-[t_A]$ durch einen Ausdruck t_c ersetzt wurde. Dabei wird t' nicht verändert. t definiert auch das Aussehen von s und es zeigt sich, dass die Folge $s \xrightarrow{iR,ccpcx} t \xrightarrow{n,a,P} r$ durch die Folge $s \xrightarrow{n,a,P} r' \xrightarrow{iR,ccpcx} r$ ersetzt werden kann.

Wir veranschaulichen dies an einer (n,case-c)-Reduktion:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x_1 = R_C^-[(\texttt{case}\ (c_l\ a_1 \ldots a_n) \ldots ((c_l\ y_1 \ldots y_n) \to t_D))], \qquad (= s)\\
 & \qquad x_2 = R_2^-[x_1], \ldots, x_i = R_{i'}^-[(\texttt{case}\ [x_{i-1}]\ \ldots (p_k \to C[x_{i-1}]) \ldots)], \ldots,\\
 & \qquad x_j = R_j^-[x_{j-1}], Env\\
 & \texttt{in}\ R_D^-[x_j])\\
\xrightarrow{iR,ccpcx} & (\texttt{letrec}\ x_1 = R_C^-[(\texttt{case}\ (c_l\ a_1 \ldots a_n) \ldots ((c_l\ y_1 \ldots y_n) \to t_D))], \qquad (= t)\\
 & \qquad x_2 = R_2^-[x_1], \ldots, x_i = R_{i'}^-[(\texttt{case}\ [x_{i-1}]\ \ldots (p_k \to C[p_k]) \ldots)], \ldots,\\
 & \qquad x_j = R_j^-[x_{j-1}], Env\\
 & \texttt{in}\ R_D^-[x_j])\\
\xrightarrow{n,case-c} & (\texttt{letrec}\ x_1 = R_C^-[(\texttt{letrec}\ y_1 = a_1, \ldots, y_n = a_n\ \texttt{in}\ t_D)], x_2 = R_2^-[x_1], \ldots, \quad (= r)\\
 & \qquad x_i = R_{i'}^-[(\texttt{case}\ [x_{i-1}]\ \ldots (p_k \to C[p_k]) \ldots)], \ldots, x_j = R_j^-[x_{j-1}], Env\\
 & \texttt{in}\ R_D^-[x_j])
\end{array}
$$

$$
\begin{array}{ll}
 & (\texttt{letrec}\ x_1 = R_C^-[(\texttt{case}\ (c_l\ a_1 \ldots a_n) \ldots ((c_l\ y_1 \ldots y_n) \to t_D))], \qquad (= s)\\
 & \qquad x_2 = R_2^-[x_1], \ldots, x_i = R_{i'}^-[(\texttt{case}\ [x_{i-1}]\ \ldots (p_k \to C[x_{i-1}]) \ldots)], \ldots,\\
 & \qquad x_j = R_j^-[x_{j-1}], Env\\
 & \texttt{in}\ R_D^-[x_j])\\
\xrightarrow{n,case-c} & (\texttt{letrec}\ x_1 = R_C^-[(\texttt{letrec}\ y_1 = a_1, \ldots, y_n = a_n\ \texttt{in}\ t_D)], x_2 = R_2^-[x_1], \ldots, \quad (= r')\\
 & \qquad x_i = R_{i'}^-[(\texttt{case}\ [x_{i-1}]\ \ldots (p_k \to C[x_{i-1}]) \ldots)], \ldots, x_j = R_j^-[x_{j-1}], Env\\
 & \texttt{in}\ R_D^-[x_j])\\
\xrightarrow{iR,ccpcx} & (\texttt{letrec}\ x_1 = R_C^-[(\texttt{letrec}\ y_1 = a_1, \ldots, y_n = a_n\ \texttt{in}\ t_D)], x_2 = R_2^-[x_1], \ldots, \quad (= r)\\
 & \qquad x_i = R_{i'}^-[(\texttt{case}\ [x_{i-1}]\ \ldots (p_k \to C[p_k]) \ldots)], \ldots, x_j = R_j^-[x_{j-1}], Env\\
 & \texttt{in}\ R_D^-[x_j])
\end{array}
$$

III. Auf t folgt eine (n,llet-e)-Reduktion.

Fall 1.e) aus Definition 3.14 zeigt, dass t die Form $(\texttt{letrec}\ x_1 = (\texttt{letrec}\ Env_A\ \texttt{in}\ t_A), x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env\ \texttt{in}\ R_{j+1}^-[x_j])$ hat. Da $t = R[t']$, muss gelten $R_i^-[x_i] = R_{i'}^-[t'] = R_{i'}^-[(\texttt{case}\ x_i\ Alts)]$ für $2 \leq i \leq j+1$.

Die zugehörige Form von s zeigt, dass die Reduktionsfolge $s \xrightarrow{iR,ccpcx} t \xrightarrow{n,llet-e} r$ durch die Folge $s \xrightarrow{n,llet-e} r' \xrightarrow{iR,ccpcx} r$ ausgetauscht werden kann.

IV. Auf t folgt eine (n,IOr-e, P)- oder eine (n,cp-e)-Reduktion, die eine Abstraktion in eine Applikation kopiert.

Die zugehörigen Fälle 3.c) und 4.d) aus Definition 3.14 zeigen, dass t die

Form ($\texttt{letrec}\ x_1 = t_A, x_2 = x_1, \ldots, x_m = x_{m-1}, y_1 = R_1^-[R_A^-[x_m]], y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}], Env\ \texttt{in}\ R_{j+1}^-[y_j]$) hat, wobei R_A^- von der Form ($\texttt{IO}\ [\cdot]$) oder ($[\cdot]\ t_B$) und t_A eine Konstante oder eine Abstraktion ist.

Um $R[t'] = t$ zu erfüllen, muss $R_i^-[y_{i-1}] = R_{i'}^-[t'] = R_{i'}^-[(\texttt{case}\ y_{i-1}\ Alts)]$ für $2 \leq i \leq j+1$ gelten.

Wiederum werden die Ausdrücke s' in s und t' in t durch die Normalordnungsreduktion nicht verändert und bleiben in einem Reduktionskontext. Somit kann das erste Diagramm angewendet werden.

V. Auf t folgt eine (allgemeine) (n,cp-in)- oder eine (n,cp-e)-Reduktion, die eine Abstraktion in einen `case`-Ausdruck kopiert

Betrachten wir die zugehörigen Fälle 4.c) und 4.e) aus Definition 3.14: Es gibt Fälle in denen die Variable, die durch die Abstraktion ersetzt wird, nicht das erste Argument des `case`-Ausdrucks t' ist. Diese Fälle sind analog zu I. und IV. und werden nicht weiter betrachtet.

Sei nun $t' = (\texttt{case}\ x_m\ Alts)$ und die Variable x_m werde mittels der (n,cp)-Reduktion durch eine Abstraktion ersetzt.

Dann hat t die Form

i) $(\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$\texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots (p_i \to C[p_i]) \ldots (p_N \to t_N))])$
falls eine (n,cp-in)-Reduktion stattfindet.

ii) $(\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1},$
$y_1 = R_1^-[(\texttt{case}\ x_m\ \ldots (p_i \to C[p_i]) \ldots (p_N \to t_N))],$
$y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}]$
$\texttt{in}\ R_0^-[y_j])$
falls eine (n,cp-e)-Reduktion stattfindet.

In beiden Fällen ist das dritte Diagramm anwendbar, da nach der (n,cp)-Reduktion eine (n,case-lam)-Reduktion folgen muss, die dazu führt, dass sowohl die Reduktionsfolge $s \xrightarrow{iR,ccpcx} t \xrightarrow{n,cp} r_1 \xrightarrow{n,case-lam} r$ als auch die Folge $s \xrightarrow{n,cp} r_2 \xrightarrow{n,case-lam} r$ mit dem selben Ausdruck r enden.

Wir zeigen dies beispielhaft an einer (n,cp-in)-Reduktion:

$$
\begin{aligned}
& (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env && (= s)\\
& \quad \texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots (p_i \to C[x_m]) \ldots (p_N \to t_N))]) \\
\xrightarrow{iR,ccpcx}\ & (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env && (= t)\\
& \quad \texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots (p_i \to C[p_i]) \ldots (p_N \to t_N))]) \\
\xrightarrow{n,cp-in}\ & (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in} && (= r_1)\\
& \quad R_0^-[(\texttt{case}\ (\lambda v.w)\ \ldots (p_i \to C[p_i]) \ldots (p_N \to t_N))])
\end{aligned}
$$

$$\xrightarrow{n,case-lam} (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \quad (= r)$$
$$\texttt{in}\ R_0^-[(\texttt{letrec}\ \{\}\ \texttt{in}\ t_n)])$$

$$(\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \quad (= s)$$
$$\texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots(p_i \to C[x_m])\ldots(p_N \to t_N))])$$
$$\xrightarrow{n,cp-in} (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \quad (= r_2)$$
$$\texttt{in}\ R_0^-[(\texttt{case}\ (\lambda v.w)\ \ldots(p_i \to C[x_m])\ldots(p_N \to t_N))])$$
$$\xrightarrow{n,case-lam} (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \quad (= r)$$
$$\texttt{in}\ R_0^-[(\texttt{letrec}\ \{\}\ \texttt{in}\ t_n)])$$

VI. Auf t folgt eine (n,case-in)- oder eine (n,case-e)-Reduktion.

Falls der durch die case-Reduktion eliminierte `case`-Ausdruck nicht der Ausdruck t' ist, ist das erste Diagramm anwendbar. Die Aufspaltung dieser Fälle erfolgt ähnlich wie in I. und IV.

Wir verwenden nun folgende Notation um Ausdrücke verkürzt darzustellen. Mit $c\ \overrightarrow{y}$ bezeichnen wir die Konstruktoranwendung $c\ y_1 \ldots y_{ar(c)}$, mit $\overrightarrow{y} = \overrightarrow{z}$ kürzen wir die Menge von Bindungen $y_1 = z_1, \ldots, y_n = z_n$ in einer `letrec`-Umgebung ab.

Sei t' nun der `case`-Ausdruck, der durch die case-Reduktion entfernt wird.

Mit den Fällen 2.b) und 2.c) aus Definition 3.14 folgt, dass t eine der folgenden Formen haben muss:

i) $(\texttt{letrec}\ x_1 = (c_k\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$\texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots(p_i \to C[p_i])\ldots(p_N \to t_N))]),$
falls eine (n,case-in)-Reduktion stattfindet.

ii) $(\texttt{letrec}\ x_1 = (c_k\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1},$
$y_1 = R_1^-[(\texttt{case}\ x_m\ \ldots(p_i \to C[p_i])\ldots(p_N \to t_N))],$
$y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_j]$
$\texttt{in}\ R_0^-[y_j]),$
falls eine (n,case-e)-Reduktion stattfindet.

Je nachdem, ob die Ersetzung durch die (iR,ccpcx)-Reduktion in der durch die (n,case)-Reduktion gewählten Alternative stattfindet ($i = k$), oder in einer anderen `case`-Alternative ($i \neq k$), ist das vierte, bzw. das zweite Diagramm, anwendbar. Wir zeigen dies exemplarisch.
Zunächst betrachten wir den Fall $i \neq k$ anhand einer (n,case-in)-Reduktion:

$$(\texttt{letrec}\ x_1 = (c_k\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \quad (= s)$$
$$\texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots(p_i \to C[x_m])\ldots((c_k\ \overrightarrow{z}) \to t_k)\ldots)])$$
$$\xrightarrow{iR,ccpcx} (\texttt{letrec}\ x_1 = (c_k\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \quad (= t)$$
$$\texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots(p_i \to C[p_i])\ldots((c_k\ \overrightarrow{z}) \to t_k)\ldots)])$$

$$\begin{array}{rl}
\xrightarrow{n,case-in} & (\texttt{letrec}\ x_1 = (c_k\ \overrightarrow{y}), \overrightarrow{y} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1}, Env \qquad (= r)\\
& \texttt{in}\ R_0^-[(\texttt{letrec}\ \overrightarrow{z} = \overrightarrow{y}\ \texttt{in}\ t_k)])\\
\hline
& (\texttt{letrec}\ x_1 = (c_k\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1}, Env \qquad (= s)\\
& \texttt{in}\ R_0^-[(\texttt{case}\ x_m\ \ldots (p_i \to C[x_m]) \ldots ((c_k\ \overrightarrow{z}) \to t_k) \ldots)])\\
\xrightarrow{n,case-in} & (\texttt{letrec}\ x_1 = (c_k\ \overrightarrow{y}), \overrightarrow{y} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1}, Env \qquad (= r)\\
& \texttt{in}\ R_0^-[(\texttt{letrec}\ \overrightarrow{z} = \overrightarrow{y}\ \texttt{in}\ t_k)])
\end{array}$$

Im Anhang in Abschnitt A.1.1 ist ein Beispiel für eine (n,case-e)-Reduktion für den Fall $i = k$ angegeben.

□

Lemma 3.48. *Ein vollständiger Satz von Gabeldiagrammen für* $\xrightarrow{iR,ccpcx}$ *ist:*

$$\xleftarrow{n,a,P} \cdot \xrightarrow{iR,ccpcx} \quad \rightsquigarrow \quad \xrightarrow{iR,ccpcx} \cdot \xleftarrow{n,a,P}$$

$$\xleftarrow{n,case} \cdot \xrightarrow{iR,ccpcx} \quad \rightsquigarrow \quad \xleftarrow{n,case}$$

$$\xleftarrow{n,case} \cdot \xleftarrow{n,cp} \cdot \xrightarrow{iR,ccpcx} \quad \rightsquigarrow \quad \xleftarrow{n,case} \cdot \xleftarrow{n,cp}$$

$$\xleftarrow{n,(lll)^*} \cdot \xleftarrow{n,case} \cdot \xrightarrow{iR,ccpcx} \quad \rightsquigarrow \quad \xrightarrow{iR,(cpx)^*} \cdot \xrightarrow{iR,cpcx} \cdot \xrightarrow{iR,(cpx)^*} \cdot \xrightarrow{iR,(xch)^*} \cdot \xrightarrow{iR,(cpx)^*} \cdot$$
$$\xrightarrow{iR,(xch)^*} \cdot \xrightarrow{iR,(gc)^*} \cdot \xleftarrow{n,(lll)^*} \cdot \xleftarrow{n,case}$$

Beweis. Die Gabeldiagramme können mithilfe der Vertauschungsdiagramme gefunden werden. Eine Fallunterscheidung wie im Beweis zu Lemma 3.47 zeigt dann die Vollständigkeit der Diagramme □

Lemma 3.49. *Die Reduktion* (ccpcx) *kann nur endlich oft angewendet werden.*

Beweis. Sei $av((\texttt{case}\ x\ Alts))$ die Anzahl an Vorkommen der Variablen x in $Alts$.

Betrachtet man nun die Summe der av aller Ausdrücke der Form $(\texttt{case}\ x\ Alts)$, wobei x eine Variable ist, so wird diese Summe durch Anwendung einer (ccpcx)-Reduktion stets um 1 verringert.

Da diese Summe nicht negativ werden kann, gibt es nur eine endliche Anzahl von (ccpcx)-Reduktionen. □

Satz 3.50. *Die Regel* (ccpcx) *ist eine korrekte Programmtransformation.*

Beweis. Der Beweis verläuft analog zum Beweis von Satz 3.45 in Verbindung mit Lemma 3.46, wobei jedoch noch zu zeigen ist, dass die jeweilige Transformation der Reduktionsfolge terminiert.

Dies ist jedoch offensichtlich, denn alle Diagramme, bis auf das jeweils erste, entfernen die (iR,ccpcx)-Reduktion und ersetzen diese durch eine endliche Anzahl von korrekten Programmtransformationen.

Die jeweils ersten Diagramme der vollständigen Sätze verringern bei Anwendung die Länge der noch zu transformierenden Reduktionsfolge um eins. □

3.8.3 Korrektheit von (lcshift)

Wir benötigen die Transformation (lcshift), um später die Korrektheit der (ccase-in)-Transformation zu zeigen.

Lemma 3.51. *Die Transformation (lcshift) ist eine korrekte Programmtransformation.*

Beweis. Wir überführen die beiden Ausdrücke der (lcshift)-Regel ineinander, indem wir nur korrekte Programmtransformationen benutzen. z sei eine neue Variable.

$$
\begin{aligned}
& (\texttt{letrec}\ y = s, Env\ \texttt{in}\ R^-[(\texttt{case}\ y\ (p_1 \to t_1) \ldots (p_N \to t_N))]) \\
\xleftarrow{(gc)^*}\ & (\texttt{letrec}\ y = s, Env\ \texttt{in}\ R^-[(\texttt{case}\ y\ (p_1 \to (\texttt{letrec}\ z = y\ \texttt{in}\ t_1)) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ z = y\ \texttt{in}\ t_N)))]) \\
\xleftarrow{(cpx)^*}\ & (\texttt{letrec}\ y = s, Env\ \texttt{in}\ R^-[(\texttt{case}\ y\ (p_1 \to (\texttt{letrec}\ z = y\ \texttt{in}\ t_1[z/y])) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ z = y\ \texttt{in}\ t_N[z/y])))]) \\
\xrightarrow{(ccpcx)^*}\ & (\texttt{letrec}\ y = s, Env\ \texttt{in}\ R^-[(\texttt{case}\ y\ (p_1 \to (\texttt{letrec}\ z = p_1\ \texttt{in}\ t_1[z/y])) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ z = p_N\ \texttt{in}\ t_N[z/y])))]) \\
\xrightarrow{ucp}\ & (\texttt{letrec}\ y = s, Env\ \texttt{in}\ R^-[(\texttt{case}\ s\ (p_1 \to (\texttt{letrec}\ z = p_1\ \texttt{in}\ t_1[z/y])) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ z = p_N\ \texttt{in}\ t_N[z/y])))]) \\
\xrightarrow{gc}\ & (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ s\ (p_1 \to (\texttt{letrec}\ z = p_1\ \texttt{in}\ t_1[z/y])) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ z = p_N\ \texttt{in}\ t_N[z/y])))]) \\
\xleftarrow{(gc)^*}\ & (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ s\ (p_1 \to (\texttt{letrec}\ y = z, z = p_1\ \texttt{in}\ t_1[z/y])) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ y = z, z = p_N\ \texttt{in}\ t_N[z/y])))]) \\
\xrightarrow{(xch)^*}\ & (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ s\ (p_1 \to (\texttt{letrec}\ y = p_1, z = y\ \texttt{in}\ t_1[z/y])) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ y = p_N, z = y\ \texttt{in}\ t_N[z/y])))]) \\
\xrightarrow{(cpx)^*}\ & (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ s\ (p_1 \to (\texttt{letrec}\ y = p_1, z = y\ \texttt{in}\ t_1)) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ y = p_N, z = y\ \texttt{in}\ t_N)))]) \\
\xrightarrow{(gc)^*}\ & (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ s\ (p_1 \to (\texttt{letrec}\ y = p_1\ \texttt{in}\ t_1)) \ldots \\
& \qquad (p_N \to (\texttt{letrec}\ y = p_N\ \texttt{in}\ t_N)))])
\end{aligned}
$$

Die (ucp)-Reduktion darf in der Umformung verwendet werden, da y nicht frei in s, Env und R^- vorkommt. □

3.8.4 Korrektheit von (ccase)

Lemma 3.52. *Wenn $s \xrightarrow{iR,ccase} t$, dann gilt: s ist in WHNF gdw. t ist in WHNF.*

Beweis. Die Aussage gilt, da eine WHNF keinen case-Ausdruck in einem Reduktionskontext besitzt. □

Lemma 3.53. *Ein vollständiger Satz von Vertauschungsdiagrammen für* $\xrightarrow{iR,ccase}$ *ist:*

$$\xrightarrow{iR,ccase} \cdot \xrightarrow{n,a,P} \quad \rightsquigarrow \quad \xrightarrow{n,a,P} \cdot \xrightarrow{iR,ccase}$$

$$\xrightarrow{iR,ccase} \cdot \xrightarrow{n,(lll)^+} \quad \rightsquigarrow \quad \xrightarrow{n,(lll)^+} \cdot \xrightarrow{iR,ccase}$$

$$\xrightarrow{iR,ccase} \cdot \xrightarrow{n,case} \quad \rightsquigarrow \quad \xrightarrow{n,case} \cdot \xrightarrow{n,lll}$$

Beweis. Der Beweis verläuft mit einer identischen Fallunterscheidung wie der Beweis der Vollständigkeit des Satzes an Vertauschungsdiagrammen für die Transformation (capp) (Lemma 3.42). Aufgrund dessen geben wir nur die wesentlichen Resultate des Beweises und einige Beispiele an.

Jede Anwendung von (ccase) in einem Reduktionskontext ist intern. Sei $s = R[s'] = R[(\texttt{case}\ (\texttt{case}\ s_0\ (p_1 \to t_1) \ldots (p_N \to t_N))\ Alts)]$ und $t = R[t'] = R[(\texttt{case}\ s_0\ (p_1 \to (\texttt{case}\ t_1\ Alts)) \ldots (p_N \to (\texttt{case}\ t_N\ Alts)))]$, d.h. $s \xrightarrow{iR,ccase} t$. Aufgrund von Lemma 3.52 genügt es zu zeigen, dass für jede Reduktionsfolge $s \xrightarrow{iR,ccase} t \xrightarrow{n,a,P} r$ ein anwendbares Diagramm im vollständigen Satz enthalten ist, oder dass die Reduktionsfolge mithilfe von Normalordnungsreduktionen derart verlängert werden kann, dass ein Diagramm angewendet werden kann.

I. Folgt auf t eine (n, a, P)-Reduktion mit $a \in \{$IOlet, lapp, IOr-c, lbeta, IOr-in, cp-in IOr-e, cp-e, llet-e $\}$, so kann die Reduktionsfolge $s \xrightarrow{iR,ccase} t \xrightarrow{n,a,P} r$ durch die Folge $s \xrightarrow{n,a,P} r' \xrightarrow{iR,ccase} r$ ersetzt werden, d.h. das erste Diagramm ist anwendbar. Der Beweis verläuft analog zu I., II., III., und IV. im Beweis zu Lemma 3.42.

   Ein Beispiel einer (n,cp-e)-Reduktion ist im Anhang im Abschnitt A.2.1 angegeben.

II. Eine (n,llet-in)-Reduktion kann nicht auf t folgen.

III. Folgt eine (n,lcase)-Reduktion auf t, so kann die Reduktionsfolge $s \xrightarrow{iR,ccase} t \xrightarrow{n,lcase} r$ mit (n,lll)-Reduktionen verlängert werden, so dass das zweite Diagramm angewendet werden kann (analog wie V. in Beweis zu Lemma 3.42).

   Beispiel:

$$\begin{array}{rl}
s & = (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ (p_1 \to t_1) \ldots)\ Alts)]) \\
\xrightarrow{iR,ccase} & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ (p_1 \to (\texttt{case}\ t_1\ Alts)) \ldots)]) \\
\xrightarrow{n,lcase} & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{letrec}\ Env'\ \texttt{in}\ (\texttt{case}\ t_B\ (p_1 \to (\texttt{case}\ t_1\ Alts)) \ldots))]) \\
\xrightarrow{n,(lll)^*} & (\texttt{letrec}\ Env, Env'\ \texttt{in}\ R_A^-[(\texttt{case}\ t_B\ (p_1 \to (\texttt{case}\ t_1\ Alts)) \ldots)]) \\
\hline
s & = (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ (p_1 \to t_1) \ldots)\ Alts)]) \\
\xrightarrow{n,lcase} & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ (\texttt{case}\ t_B\ (p_1 \to t_1) \ldots))\ Alts)])
\end{array}$$

$$\xrightarrow{n,lcase} (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{letrec}\ Env'\ \texttt{in}\ (\texttt{case}\ (\texttt{case}\ t_B\ (p_1 \to t_1)\ldots)\ Alts))])$$
$$\xrightarrow{n,(lll)^*} (\texttt{letrec}\ Env\ , Env'\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{case}\ t_B\ (p_1 \to t_1)\ldots)\ Alts)])$$
$$\xrightarrow{iR,ccase} (\texttt{letrec}\ Env\ , Env'\ \texttt{in}\ R_A^-[(\texttt{case}\ t_B\ (p_1 \to (\texttt{case}\ t_1\ Alts))\ldots)])$$

IV. Folgt auf t eine (n,case)-Reduktion, die t' eliminiert, so kann das dritte Diagramm angewendet werden.

Beispiel:

$$\begin{array}{ll}
& (\texttt{letrec}\ x_1 = (c_i\ a_1 \ldots\ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= s)\\
& \qquad y_1 = R_0^-[(\texttt{case}\ (\texttt{case}\ x_m\ \ldots((c_i\ z_1 \ldots z_n) \to t_i)\ldots)\ Alts)],\\
& \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
& \texttt{in}\ R_A^-[y_j])\\
\xrightarrow{iR,ccase} & (\texttt{letrec}\ x_1 = (c_i\ a_1 \ldots\ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= t)\\
& \qquad y_1 = R_0^-[(\texttt{case}\ x_m\ \ldots((c_i\ z_1 \ldots z_n) \to (\texttt{case}\ t_i\ Alts))\ldots)],\\
& \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
& \texttt{in}\ R_A^-[y_j])\\
\xrightarrow{n,case-e} & (\texttt{letrec}\ x_1 = (c_i\ w_1 \ldots\ w_n), w_1 = a_1, \ldots, w_n = a_n x_2 = x_1, \ldots, x_m = x_{m-1},\\
& \qquad y_1 = R_0^-[(\texttt{letrec}\ z_1 = w_1, \ldots, z_n = w_n\ \texttt{in}(\texttt{case}\ t_i\ Alts))],\\
& \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
& \texttt{in}\ R_A^-[y_j])
\end{array}$$

$$\begin{array}{ll}
& (\texttt{letrec}\ x_1 = (c_i\ a_1 \ldots\ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= s)\\
& \qquad y_1 = R_0^-[(\texttt{case}\ (\texttt{case}\ x_m\ \ldots((c_i\ z_1 \ldots z_n) \to t_i)\ldots)\ Alts)],\\
& \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
& \texttt{in}\ R_A^-[y_j])\\
\xrightarrow{n,case-e} & (\texttt{letrec}\ x_1 = (c_i\ w_1 \ldots\ w_n), w_1 = a_1, \ldots, w_n = a_n x_2 = x_1, \ldots, x_m = x_{m-1},\\
& \qquad y_1 = R_0^-[(\texttt{case}\ (\texttt{letrec}\ z_1 = w_1, \ldots, z_n = w_n\ \texttt{in}\ t_i)\ Alts)],\\
& \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
& \texttt{in}\ R_A^-[y_j])\\
\xrightarrow{n,lcase} & (\texttt{letrec}\ x_1 = (c_i\ w_1 \ldots\ w_n), w_1 = a_1, \ldots, w_n = a_n x_2 = x_1, \ldots, x_m = x_{m-1},\\
& \qquad y_1 = R_0^-[(\texttt{letrec}\ z_1 = w_1, \ldots, z_n = w_n\ \texttt{in}\ (\texttt{case}\ t_i\ Alts))],\\
& \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
& \texttt{in}\ R_A^-[y_j])
\end{array}$$

Bei allen anderen case-Reduktionen ist das erste Diagramm anwendbar.

□

Lemma 3.54. *Ein vollständiger Satz von Gabeldiagrammen für* $\xrightarrow{iR,ccase}$ *ist:*

$$\xleftarrow{n,a,P} \cdot \xrightarrow{iR,ccase} \quad \rightsquigarrow \quad \xrightarrow{iR,ccase} \cdot \xleftarrow{n,a,P}$$

$$\xleftarrow{n,(lll)^+} \cdot \xrightarrow{iR,ccase} \rightsquigarrow \xrightarrow{iR,ccase} \cdot \xleftarrow{n,(lll)^+}$$

$$\xleftarrow{n,lll} \cdot \xleftarrow{n,case} \cdot \xrightarrow{iR,ccase} \rightsquigarrow \xleftarrow{n,case}$$

Beweis. Die Gabeldiagramme können mithilfe der Vertauschungsdiagramme gefunden werden. Die Vollständigkeit kann mit einer Fallunterscheidung wie im Beweis zu Lemma 3.53 bewiesen werden. □

Lemma 3.55. *Die Reduktion* (ccase) *kann nur endlich oft angewendet werden.*

Beweis. Durch eine ccase-Reduktion wird ein äußerer `case`-Ausdruck nach innen verschoben. Da Ausdrücke nur endliche Tiefe haben, können alle `case`-Ausdrücke nur endlich oft nach innen geschoben werden. □

Satz 3.56. *Die Regel* (ccase) *ist eine korrekte Programmtransformation.*

Beweis. Der Beweis verläuft analog zum Beweis von Satz 3.45 in Verbindung mit Lemma 3.52 und den vollständigen Sätzen an Vertauschungs- und Gabeldiagrammen. □

3.8.5 Korrektheit von (ccase-in)

Im Folgenden zeigen wir, dass die Transformation (ccase-in) eine korrekte Programmtransformation ist, indem wir den Ausdruck auf der linken Seite in den Ausdruck auf der rechten Seite der Regel transformieren und dabei nur korrekte Programmtransformationen benutzen. Wir verwenden insbesondere die zuvor als korrekt bewiesenen (lcshift)- und (ccase)- Transformationen.

Lemma 3.57. *(ccase-in) ist eine korrekte Programmtransformation*

Beweis.

$$\begin{array}{l}
\quad (\texttt{letrec}\ y = (\texttt{case}\ s\ (p_1 \to t_1) \ldots (p_N \to t_N)) \\
\quad\ \texttt{in}\ (\texttt{case}\ y\ (q_1 \to r_1) \ldots (q_N \to r_N))) \\
\xrightarrow{lcshift} (\texttt{letrec}\ \{\}\ \texttt{in}\ (\texttt{case}\ (\texttt{case}\ s\ (p_1 \to t_1) \ldots (p_N \to t_N)) \\
\qquad\qquad (q_1 \to (\texttt{letrec}\ y = q_1\ \texttt{in}\ r_1)) \\
\qquad\qquad \ldots \\
\qquad\qquad (q_N \to (\texttt{letrec}\ y = q_n\ \texttt{in}\ r_N))))) \\
\xrightarrow{gc} (\texttt{case}\ (\texttt{case}\ s\ (p_1 \to t_1) \ldots (p_N \to t_N))\ (q_1 \to (\texttt{letrec}\ y = q_1\ \texttt{in}\ r_1)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (q_N \to (\texttt{letrec}\ y = q_n\ \texttt{in}\ r_N))))
\end{array}$$

$$
\begin{array}{ll}
\xrightarrow{ccase} & (\texttt{case}\ s \\
& \quad (p_1 \to (\texttt{case}\ t_1\ (q_1 \to\ y = q_1\ \texttt{in}\ r_1) \\
& \qquad\qquad \ldots \\
& \qquad\qquad (q_N \to\ y = q_N\ \texttt{in}\ r_N))) \\
& \quad \ldots \\
& \quad (p_N \to (\texttt{case}\ t_N\ (q_1 \to\ y = q_1\ \texttt{in}\ r_1) \\
& \qquad\qquad \ldots \\
& \qquad\qquad (q_N \to\ y = q_N\ \texttt{in}\ r_N)))) \\
\xrightarrow{(lcshift)^*} & (\texttt{case}\ s \\
& \quad (p_1 \to (\texttt{letrec}\ y = t_1\ \texttt{in}\ (\texttt{case}\ y\ (q_1 \to r_1) \ldots (q_N \to r_N)))) \\
& \quad \ldots \\
& \quad (p_N \to (\texttt{letrec}\ y = t_N\ \texttt{in}\ (\texttt{case}\ y\ (q_1 \to r_1) \ldots (q_N \to r_N)))))
\end{array}
$$

□

3.8.6 Korrektheit von (crpl)

Zunächst zeigen wir an einem Beispiel, warum die Bedingung an den Kontext, in dem die (crpl)-Transformation stattfindet, notwendig ist.

Beispiel 3.58. *Sei*

$$
\begin{array}{rl}
s & = (\texttt{letrec}\ v = (\lambda x.x), w = (\texttt{IO}\ c)\ \texttt{in}\ (\texttt{case}\ (v\ w)\ (\ldots(d \to d)\ldots))) \text{ und} \\
t & = (\texttt{letrec}\ v = (\lambda x.x), w = (\texttt{IO}\ c)\ \texttt{in}\ (\texttt{case}\ (v\ w)\ (\ldots(d \to c)\ldots)))
\end{array}
$$

wobei c und d Konstanten sind.

Die Ausdrücke s und t sind nicht kontextuell äquivalent, denn der Kontext

$$C = (\texttt{case}\ [\cdot]\ (c \to \bot)\ (d \to d)\ \ldots)$$

unterscheidet die Ausdrücke, da $C[s]$ für die IO-Multimenge $P = \{(c, d)\}$ terminiert, $C[t]$ für gleiches P jedoch nicht.

Lemma 3.59. *Wenn $s \xrightarrow{iR,crpl} t$, dann gilt: s ist in WHNF gdw. t ist in WHNF.*

Beweis. Die Aussage gilt, da eine WHNF keinen `case`-Ausdruck in einem Reduktionskontext besitzt. □

Lemma 3.60. *Ein vollständiger Satz von Vertauschungsdiagrammen für $\xrightarrow{iR,crpl}$ ist:*

$$
\begin{array}{ccccc}
\xrightarrow{iR,crpl} & . & \xrightarrow{n,a,P} & \rightsquigarrow & \xrightarrow{n,a,P} \ . \ \xrightarrow{iR,crpl} \\
\xrightarrow{iR,crpl} & . & \xrightarrow{n,(lll)^+} & \rightsquigarrow & \xrightarrow{n,(lll)^+} \ . \ \xrightarrow{iR,crpl} \\
\xrightarrow{iR,crpl} & . & \xrightarrow{n,case} & \rightsquigarrow & \xrightarrow{n,case}
\end{array}
$$

Beweis. Der Beweis verläuft wiederum mit der gleichen Fallunterscheidung wie der Beweis zu Lemma 3.42.

Jede Anwendung von (crpl) in einem Reduktionskontext ist intern. Sei $s = R[s'] = R[(\texttt{case}\ s_0\ \ldots(p_i \to t_i)\ldots(p_N \to t_N))]$ und ein R ein Reduktionskontext, der die freien Variablen von s nicht auf eine Weise bindet, so dass s bezüglich R zu einer Konstruktoranwendung $(c_i\ a_1 \ldots a_n)$ auswerten kann.

Sei $t = R[t'] = R[(\texttt{case}\ s_0\ \ldots(p_i \to q)\ldots(p_N \to t_N))]$, d.h. $s \xrightarrow{iR,crpl} t$.

Lemma 3.59 zeigt, dass es nicht möglich ist, dass s keine WHNF aber t eine WHNF ist. Aufgrund dessen genügt es zu zeigen, dass für jede Reduktionsfolge $s \xrightarrow{iR,crpl} t \xrightarrow{n,a,P} r$ ein anwendbares Diagramm im vollständigen Satz enthalten ist, oder dass die Reduktionsfolge mittels Normalordnungsreduktionen verlängert werden kann, so dass ein Diagramm angewendet werden kann.

I. Folgt auf t eine (n, a, P)-Reduktion mit $a \in \{$IOlet, lapp, IOr-c, lbeta, IOr-in, cp-in IOr-e, cp-e, llet-e $\}$, so kann die Reduktionsfolge $s \xrightarrow{iR,crpl} t \xrightarrow{n,a,P} r$ durch die Folge $s \xrightarrow{n,a,P} r' \xrightarrow{iR,crpl} r$ ersetzt werden, d.h. das erste Diagramm ist anwendbar. Der Beweis verläuft analog zu I., II., III., und IV. im Beweis zu Lemma 3.42.

   Allerdings ist zu beachten, dass die (n, a, P)-Reduktion in der neuen Reduktionsfolge die Bedingung an s_0 und den Kontext R nicht verändert, denn sonst könnte die (iR,crpl)-Reduktion nicht auf r' angewendet werden. Dies ist jedoch unproblematisch, da die Bedingung durch Normalordnungsreduktionen nicht verändert werden kann.

   Wir zeigen die Anwendbarkeit des ersten Diagrammes an einer (n,IOr-in)-Reduktion, wobei sichergestellt ist, dass $(\texttt{IO}\ x_m)$ nicht zu einer Konstruktoranwendung $(c_i\ b_1 \ldots b_n)$ auswerten kann, da $(\texttt{IO}\ x_m)$ höchstens durch einen nullstelligen Konstruktor ersetzt wird.

$$
\begin{array}{rll}
 & (\texttt{letrec}\ x_1 = c_k, x_2 = x_1, \ldots, x_m = x_{m-1}, & (= s) \\
 & \quad \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ x_m)\ \ldots((c_i\ a_1 \ldots a_n) \to t_i)\ldots) & \\
\xrightarrow{iR,crpl} & (\texttt{letrec}\ x_1 = c_k, x_2 = x_1, \ldots, x_m = x_{m-1}, & (= t) \\
 & \quad \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ x_m)\ \ldots((c_i\ a_1 \ldots a_n) \to q)\ldots) & \\
\xrightarrow{n,IOr-in,(c_k,c_j))} & (\texttt{letrec}\ x_1 = c_k, x_2 = x_1, \ldots, x_m = x_{m-1}, & (= r) \\
 & \quad \texttt{in}\ (\texttt{case}\ c_j\ \ldots((c_i\ a_1 \ldots a_n) \to q)\ldots) &
\end{array}
$$

$$
\begin{array}{rll}
 & (\texttt{letrec}\ x_1 = c_k, x_2 = x_1, \ldots, x_m = x_{m-1}, & (= s) \\
 & \quad \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ x_m)\ \ldots((c_i\ a_1 \ldots a_n) \to t_i)\ldots) & \\
\xrightarrow{n,IOr-in,(c_k,c_j))} & (\texttt{letrec}\ x_1 = c_k, x_2 = x_1, \ldots, x_m = x_{m-1}, & (= r') \\
 & \quad \texttt{in}\ (\texttt{case}\ c_j\ \ldots((c_i\ a_1 \ldots a_n) \to t_i)\ldots) & \\
\xrightarrow{iR,crpl} & (\texttt{letrec}\ x_1 = c_k, x_2 = x_1, \ldots, x_m = x_{m-1}, & (= t) \\
 & \quad \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ x_m)\ \ldots((c_i\ a_1 \ldots a_n) \to q)\ldots) &
\end{array}
$$

II. Eine (n,llet-in)-Reduktion kann nicht auf t folgen.

III. Folgt eine (n,lcase)-Reduktion auf t, so kann die Reduktionsfolge $s \xrightarrow{iR,crpl} t \xrightarrow{n,lcase} r$ mit (n,lll)-Reduktionen verlängert werden, so dass das zweite Diagramm angewendet werden kann (analog zu V. in Beweis zu Lemma 3.42).

In folgendem Beispiel sei wiederum sichergestellt, dass die freien Variablen im Ausdruck (`letrec` Env' `in` t_B) nicht derart durch den Kontext (`letrec` Env `in` $[\cdot]$) gebunden werden, dass der Ausdruck bezüglich dieses Kontextes zu einer Konstruktoranwendung $(c_i\ a_1 \ldots a_n)$ auswerten kann.

$$
\begin{array}{rll}
 & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ \ldots(p_i \to t_i)\ldots)]) & (=s)\\
\xrightarrow{iR,crpl} & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ \ldots(p_i \to q)\ldots)]) & (=t)\\
\xrightarrow{n,lcase} & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{letrec}\ Env'\ \texttt{in}\ (\texttt{case}\ t_B\ \ldots(p_i \to q)))]) & \\
\xrightarrow{n,(lll)^*} & (\texttt{letrec}\ Env, Env'\ \texttt{in}\ R_A^-[(\texttt{case}\ t_B\ \ldots(p_i \to q))]) & (=r)\\
\hline
 & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{case}\ (\texttt{letrec}\ Env'\ \texttt{in}\ t_B)\ \ldots(p_i \to t_i)\ldots)]) & (=s)\\
\xrightarrow{n,lcase} & (\texttt{letrec}\ Env\ \texttt{in}\ R_A^-[(\texttt{letrec}\ Env'\ \texttt{in}\ (\texttt{case}\ t_B\ \ldots(p_i \to t_i)\ldots))]) & \\
\xrightarrow{n,(lll)^*} & (\texttt{letrec}\ Env\ , Env'\ \texttt{in}\ R_A^-[(\texttt{case}\ t_B\ \ldots(p_i \to t_i)\ldots)]) & \\
\xrightarrow{iR,crpl} & (\texttt{letrec}\ Env\ , Env'\ \texttt{in}\ R_A^-[(\texttt{case}\ t_B\ \ldots(p_i \to q)\ldots)]) & (=r)
\end{array}
$$

IV. Folgt auf t eine (n,case)-Reduktion, die t' eliminiert, so kann das dritte Diagramm angewendet werden. Die Bedingung an den Kontext R führt dazu, dass die Alternative, die durch die $(iR, crpl)$-Reduktion verändert wird, nicht durch die case-Reduktion gewählt wird.

Beispiel:

$$
\begin{array}{rll}
 & (\texttt{letrec}\ x_1 = (c_j\ a_1 \ldots\ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, & (=s)\\
 & \qquad y_1 = R_0^-[(\texttt{case}\ x_m\ \ldots(p_i \to t_i)\ldots((c_j\ z_1 \ldots z_n) \to t_j)\ldots)], & \\
 & \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env & \\
 & \texttt{in}\ R_A^-[y_j]) & \\
\xrightarrow{iR,crpl} & (\texttt{letrec}\ x_1 = (c_j\ a_1 \ldots\ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, & (=t)\\
 & \qquad y_1 = R_0^-[(\texttt{case}\ x_m\ \ldots(p_i \to q)\ldots((c_j\ z_1 \ldots z_n) \to t_j)\ldots)], & \\
 & \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env & \\
 & \texttt{in}\ R_A^-[y_j]) & \\
\xrightarrow{n,case-e} & (\texttt{letrec}\ x_1 = (c_j\ w_1 \ldots\ w_n), w_1 = a_1, \ldots, w_n = a_n, & (=r)\\
 & \qquad x_2 = x_1, \ldots, x_m = x_{m-1}, & \\
 & \qquad y_1 = R_0^-[(\texttt{letrec}\ z_1 = w_1, \ldots, z_n = w_n\ \texttt{in}\ t_j)], & \\
 & \qquad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env & \\
 & \texttt{in}\ R_A^-[y_j]) &
\end{array}
$$

$$
\begin{array}{rl}
 & (\texttt{letrec}\ x_1 = (c_j\ a_1 \ldots\ a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= s)\\
 & \qquad\quad y_1 = R_0^-[(\texttt{case}\ x_m\ \ldots (p_i \to t_i) \ldots ((c_j\ z_1 \ldots z_n) \to t_j) \ldots)],\\
 & \qquad\quad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
 & \texttt{in}\ R_A^-[y_j])\\
\xrightarrow{n,case-e} & (\texttt{letrec}\ x_1 = (c_j\ w_1 \ldots\ w_n), w_1 = a_1, \ldots, w_n = a_n, \qquad (= r)\\
 & \qquad\quad x_2 = x_1, \ldots, x_m = x_{m-1},\\
 & \qquad\quad y_1 = R_0^-[(\texttt{letrec}\ z_1 = w_1, \ldots, z_n = w_n\ \texttt{in}\ t_j)],\\
 & \qquad\quad y_2 = R_2^-[y_1], \ldots, x_j = R_j[y_{j-1}], Env\\
 & \texttt{in}\ R_A^-[y_j])
\end{array}
$$

Bei allen anderen case-Reduktionen ist das erste Diagramm anwendbar.

□

Lemma 3.61. *Ein vollständiger Satz von Gabeldiagrammen für* $\xrightarrow{iR,crpl}$ *ist:*

$$
\begin{array}{ccccc}
\xleftarrow{n,a,P} & \cdot & \xrightarrow{iR,crpl} & \rightsquigarrow & \xrightarrow{iR,crpl} \cdot \xleftarrow{n,a,P}\\
\xleftarrow{n,(lll)^+} & \cdot & \xrightarrow{iR,crpl} & \rightsquigarrow & \xrightarrow{iR,crpl} \cdot \xleftarrow{n,(lll)^+}\\
\xleftarrow{n,case} & \cdot & \xrightarrow{iR,crpl} & \rightsquigarrow & \xleftarrow{n,case}
\end{array}
$$

Beweis. Die Gabeldiagramme können mithilfe der Vertauschungsdiagramme gefunden werden. Eine Fallunterscheidung wie im Beweis von Lemma 3.60 zeigt die Vollständigkeit der Diagramme. □

Man beachte, dass die Reduktion (iR,crpl) beliebig oft angewendet werden kann, aber keines der Diagramme konstruiert eine unendliche Folge von (iR,crpl)-Reduktionen.

Satz 3.62. *Die Regel* (crpl) *ist eine korrekte Programmtransformation.*

Beweis. Der Beweis verläuft analog zum Beweis von Satz 3.45 in Verbindung mit Lemma 3.59 und den vollständigen Sätzen an Vertauschungs- und Gabeldiagrammen, wobei jedoch wichtig ist zu beobachten, dass bei den einzelnen Transformationen der Reduktionsfolge die Eigenschaft erhalten bleibt, dass der Reduktionskontext, indem die (crpl)-Reduktion angewendet wird, die freien Variablen des ersten Argumentes des `case`-Ausdrucks nicht derart bindet, dass dieses Argument zu einer Konstruktoranwendung $(c_i\ a_1 \ldots a_n)$ auswerten kann. □

3.9 Transformationen zum Kopieren von Ausdrücken

In [Sch03a] wurden bereits einige korrekte Transformationen definiert, die das Kopieren von bestimmten Ausdrücken in bestimmte Kontexte erlauben. So dürfen Variablen (cpx-Reduktion), Konstanten (cpcx-Reduktion) und Abstraktionen (cp-Reduktion) in beliebige Kontexte kopiert werden. Zudem wurde in [Sch03a] die (ucp)-Reduktion definiert und als korrekt bewiesen, die es erlaubt Ausdrücke zu kopieren, wenn sie nur einmal und nicht im Rumpf einer Abstraktion vorkommen.

Im Folgenden werden wir weitere Transformationen definieren, die ein eingeschränktes Kopieren erlauben.

Definition 3.63. *Die Sprache $L_{cheap} \subset L_{FUNDIO}$ sei durch folgende Grammatik definiert:*

$$
\begin{array}{lcll}
\mathbf{E}_c & ::= & V & \text{mit } V \text{ Variable} \\
& | & (\lambda V.s) & \text{mit } s \in L_{FUNDIO} \\
& | & (c_i\ \mathbf{E}_{c,1}\ \ldots \mathbf{E}_{c,n}) & \text{mit } ar(c_i) = n \\
& | & (\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ \ \mathbf{E}_{c,1}\ \ \ldots\ \ \mathbf{E}_{c,m} & \text{mit } ar(c_i) = n > m
\end{array}
$$

Definition 3.64. *In Abbildung 3.9 sind die Transformationen* (cpcheap-in)*,* (cpcheap-e)*,* (brcp-in)*,* (brcp-e)*,* (ucpb-in) *und* (ucpb-e) *definiert.*

Die Vereinigung von (cpcheap-in) *und* (cpcheap-e) *bezeichnen wir als* (cpcheap)*.*

Die Vereinigung von (brcp-in) *und* (brcp-e) *bezeichnen wir als* (brcp)*.*

Die Vereinigung von (ucpb-in) *und* (ucpb-e) *bezeichnen wir als* (ucpb)*.*

Die (cpcheap)-Regel fasst mehrere (cp)-, (cpx)- und (cpcx)-Reduktionen zusammen, so dass Ausdrücke, die aus Variablen, Abstraktionen oder Konstruktoranwendungen (deren Argumente aus L_{cheap} sind) aufgebaut sind, in einem Schritt kopiert werden können. Der letzte Ausdruck in der Defintion von L_{cheap} ist notwendig, um ungesättigte Konstruktoranwendungen (die es in L_{FUNDIO} nicht gibt) zu simulieren.

Die (brcp)-Transformation ermöglicht es, `letrec`-Bindungen über einen `case`-Ausdruck hinweg zu schieben und sie dient als Grundlage für den Beweis der Korrektheit der (ucpb)-Regel, die eine Erweiterung der (ucp)-Regel darstellt: Ausdrücke dürfen auch dann in eine `case`-Alternative kopiert werden, selbst wenn die Variable, an die der Ausdruck gebunden ist, in anderen Alternativen mehrfache freie Vorkommen hat.

In den folgenden Abschnitten beweisen wir die Korrektheit der Transformationen.

3.9.1 Korrektheit von (cpcheap)

Lemma 3.65. *(cpcheap) ist eine korrekte Programmtransformation.*

(cpcheap-in) $(\texttt{letrec}\ x = t,\ Env\ \texttt{in}\ C[x]) \longrightarrow (\texttt{letrec}\ x = t,\ Env\ \texttt{in}\ C[t])$
wobei $t \in L_{cheap}$

(cpcheap-e) $(\texttt{letrec}\ x = t,\ y = C[x],\ Env\ \texttt{in}\ s)$
$\longrightarrow (\texttt{letrec}\ x = t,\ y = C[t],\ Env\ \texttt{in}\ s)$
wobei $t \in L_{cheap}$

(brcp-in) $(\texttt{letrec}\ y = s,\ Env\ \texttt{in}\ R^-[(\texttt{case}\ t\ (pat_1 \to t_1) \ldots (pat_N \to t_N))])$
$\longrightarrow (\texttt{letrec}\ Env\ \texttt{in}$
$R^-[(\texttt{case}\ t\ (pat_1 \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_1))$
$\ldots$
$(pat_N \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_N)))])$
wenn y nicht frei in R^-, Env, s und t vorkommt

(brcp-e) $(\texttt{letrec}$
$y = s, x = R^-[(\texttt{case}\ t\ (pat_1 \to t_1) \ldots (pat_N \to t_N))], Env$
$\texttt{in}\ t')$
$\longrightarrow (\texttt{letrec}$
$x = R^-[(\texttt{case}\ t\ (pat_1 \to (\texttt{letrec}\ x = s\ \texttt{in}\ t_1))$
$\ldots$
$(pat_N \to (\texttt{letrec}\ x = s\ \texttt{in}\ t_N)))],$
Env
$\texttt{in}\ t')$
wenn y nicht frei in R^-, Env, s, t' und t vorkommt.

(ucpb-in) $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ \ S_1[(\texttt{case}\ t\ \ldots (pat_i \to S_2[x]) \ldots)])$
$\longrightarrow (\texttt{letrec}\ x = s, Env\ \texttt{in}\ \ S_1[(\texttt{case}\ t\ \ldots (pat_i \to S_2[s]) \ldots)])$
wenn x nicht in Env, S_1, S_2, t und s vorkommt.

(ucpb-e) $(\texttt{letrec}\ x = s, Env, y = S_1[(\texttt{case}\ t \ldots (pat_i \to S_2[x]) \ldots)]\ \ \texttt{in}\ \ t_1)$
$\longrightarrow (\texttt{letrec}\ x = s, Env, y = S_1[(\texttt{case}\ t \ldots (pat_i \to S_2[s]) \ldots)]\ \ \texttt{in}\ \ t_1)$
wenn x nicht in Env, S_1, S_2, t, t_1 und s vorkommt.

Abbildung 3.9: Transformationen zum Kopieren von Ausdrücken

Beweis. Sei kt die „Konstruktortiefe“ eines Ausdrucks aus L_{cheap}, die wir wie folgt definieren:

$$\begin{aligned} kt(c_i\ a_1 \ldots a_n) &= 1 + max\{kt(a_i) \mid i = 1, \ldots n\} \\ kt((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ a_1 \ldots a_m) &= 1 + max\{kt(a_i) \mid i = 1, \ldots, m\} \\ kt(v) &= 0, \text{ wenn } v \text{ eine Variable ist} \\ kt(c) &= 0, \text{ wenn } c \text{ eine Konstante ist} \\ kt(\lambda x.s) &= 0 \end{aligned}$$

Nun beweisen wir die Aussage mit vollständiger Induktion über die Konstruktortiefe von t:

Induktionsanfang:

Die Basisfälle, in denen t eine Abstraktion, eine Variable oder eine Konstante ist, sind aufgrund der (cp)-, (cpx)- und (cpcx)-Reduktion korrekt.

Induktionsannahme:

$\forall j : j \leq k$: Für t mit $kt(t) = j$ ist die cpcheap-Transformation korrekt.

Reduktionspfeile, die diese Induktionsannahme verwenden, schreiben wir als $\xrightarrow{IA}$.

Induktionsschritt:

Sei $kt(t) = k + 1$. Wir unterscheiden zwei Fälle, je nachdem welche Form t hat.

I. $t = (c_i\ e_1 \ldots\ e_n)$, wobei die $e_i \in L_{cheap}$.

Mit der Definition von kt folgt, dass $kt(e_i) \leq k$ für $i = 1, \ldots, n$
Wir betrachten zunächst den Fall einer (cpcheap-in)-Reduktion:

$$\begin{aligned} & (\texttt{letrec}\ x = (c_i\ e_1\ \ldots\ e_n),\ Env\ \texttt{in}\ C[x]) \\ \xrightarrow{cpcx}\ & (\texttt{letrec}\ x = (c_i\ y_1\ \ldots\ y_n), y_1 = e_1,\ \ldots\ y_n = e_n,\ Env\ \texttt{in}\ C[(c_i\ y_1\ \ldots\ y_n)]) \\ \xrightarrow{(IA)^*}\ & (\texttt{letrec}\ x = (c_i\ y_1\ \ldots\ y_n), y_1 = e_1,\ \ldots\ y_n = e_n,\ Env\ \texttt{in}\ C[(c_i\ e_1\ \ldots\ e_n)]) \\ \xrightarrow{(IA)^*}\ & (\texttt{letrec}\ x = (c_i\ e_1\ \ldots\ e_n), y_1 = e_1,\ \ldots\ y_n = e_n,\ Env\ \texttt{in}\ C[(c_i\ e_1\ \ldots\ e_n)]) \\ \xrightarrow{(gc)^*}\ & (\texttt{letrec}\ x = (c_i\ e_1\ \ldots\ e_n),\ Env\ \texttt{in}\ C[(c_i\ e_1\ \ldots\ e_n)]) \end{aligned}$$

Schließlich müssen wir noch den Fall einer (cpcheap-e)-Reduktion überprüfen:

$$\begin{aligned} & (\texttt{letrec}\ x = (c_i\ e_1\ \ldots\ e_n),\ y = C[x],\ Env\ \texttt{in}\ s) \\ \xrightarrow{cpcx}\ & (\texttt{letrec}\ x = (c_i\ z_1\ \ldots\ z_n), z_1 = e_1,\ \ldots\ z_n = e_n,\ y = C[(c_i\ z_1\ \ldots\ z_n)],\ Env\ \texttt{in}\ s) \\ \xrightarrow{(IA)^*}\ & (\texttt{letrec}\ x = (c_i\ z_1\ \ldots\ z_n), z_1 = e_1,\ \ldots\ z_n = e_n,\ y = C[(c_i\ e_1\ \ldots\ e_n)],\ Env\ \texttt{in}\ s) \\ \xrightarrow{(IA)^*}\ & (\texttt{letrec}\ x = (c_i\ e_1\ \ldots\ e_n), z_1 = e_1,\ \ldots\ z_n = e_n,\ y = C[(c_i\ e_1\ \ldots\ e_n)],\ Env\ \texttt{in}\ s) \\ \xrightarrow{(gc)^*}\ & (\texttt{letrec}\ x = (c_i\ e_1\ \ldots\ e_n),\ y = C[(c_i\ e_1\ \ldots\ e_n)],\ Env\ \texttt{in}\ s) \end{aligned}$$

II. $t = ((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_m)$, $e_i \in L_{cheap}$ und $ar(c_i) = n > m$.

Mit der Definition von kt folgt, dass $kt(e_i) \leq k$ für $i = 1, \ldots, m$.

Wir betrachten zunächst den Fall einer (cpcheap-in)-Reduktion:

$$
\begin{aligned}
& (\texttt{letrec}\ x = ((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_m),\ Env\ \texttt{in}\ C[x]) \\
\xrightarrow{(lll)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1; \ldots x_m = e_m,\ Env\ \texttt{in}\ C[x]) \\
\xrightarrow{cp}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1; \ldots x_m = e_m,\ Env \\
& \quad \texttt{in}\ C[(\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n))]) \\
\xrightarrow{(IA)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1; \ldots x_m = e_m,\ Env \\
& \quad \texttt{in}\ C[(\lambda x_{m+1} \ldots x_n.(c_i\ e_1 \ldots e_m\ x_{m+1} \ldots x_n))]) \\
\xleftarrow{(gc)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1; \ldots x_m = e_m,\ Env \\
& \quad \texttt{in}\ C[(\texttt{letrec}\ x_1 = e_1, \ldots x_n = e_n\ \texttt{in}\ (\lambda x_{m+1} \ldots x_n.(c_i\ e_1 \ldots e_m\ x_{m+1} \ldots x_n)))]) \\
\xleftarrow{(IA)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1; \ldots x_m = e_m,\ Env \\
& \quad \texttt{in}\ C[(\texttt{letrec}\ x_1 = e_1, \ldots x_n = e_n\ \texttt{in}\ (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)))]) \\
\xleftarrow{(lll)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1; \ldots x_m = e_m,\ Env \\
& \quad \texttt{in}\ C[((\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_n)]) \\
\xleftarrow{(lll)^*}\ & (\texttt{letrec}\ x = ((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_n),\ Env \\
& \quad \texttt{in}\ C[((\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_n)])
\end{aligned}
$$

Abschließend betrachten wir den Fall einer (cpcheap-e)-Reduktion:

$$
\begin{aligned}
& (\texttt{letrec}\ x = ((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_m),\ y = C[x],\ Env\ \texttt{in}\ s) \\
\xrightarrow{(lll)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1, \ldots, x_m = e_m, y = C[x], Env\ \texttt{in}\ s) \\
\xrightarrow{cp}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1, \ldots, x_m = e_m, \\
& \qquad y = C[(\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n))],\ Env \\
& \quad \texttt{in}\ s) \\
\xrightarrow{(IA)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1, \ldots, x_m = e_m, \\
& \qquad y = C[(\lambda x_{m+1} \ldots x_n.(c_i\ e_1 \ldots e_m\ x_{m+1} \ldots x_n))],\ Env \\
& \quad \texttt{in}\ s) \\
\xleftarrow{(gc)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1, \ldots, x_m = e_m, Env, \\
& \qquad y = C[(\texttt{letrec}\ x_1 = e_1, \ldots x_m = e_m \\
& \qquad\qquad \texttt{in}\ (\lambda x_{m+1} \ldots x_n.(c_i\ e_1 \ldots e_m\ x_{m+1} \ldots x_n)))] \\
& \quad \texttt{in}\ s) \\
\xleftarrow{(IA)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1, \ldots, x_m = e_m, Env, \\
& \qquad y = C[(\texttt{letrec}\ x_1 = e_1, \ldots x_m = e_m\ \texttt{in}\ (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)))] \\
& \quad \texttt{in}\ s) \\
\xleftarrow{(lll)^*}\ & (\texttt{letrec}\ x = (\lambda x_{m+1} \ldots x_n.(c_i\ x_1 \ldots x_n)), x_1 = e_1, \ldots, x_m = e_m, \\
& \qquad y = C[((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_m)],\ Env \\
& \quad \texttt{in}\ s)
\end{aligned}
$$

$$\xleftarrow{(lll)^*} (\texttt{letrec}\ x = ((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_m),$$
$$y = C[((\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ e_1 \ldots e_m)],\ Env$$
$$\texttt{in}\ s)$$

□

3.9.2 Korrektheit von (brcp)

Lemma 3.66. *Wenn $s \xrightarrow{iR,brcp} t$, dann gilt: s ist in WHNF, gdw. t ist in WHNF.*

Beweis. Für (iR,brcp-in) ist dies offensichtlich, da beide Ausdrücke aus der Definition der (brcp-in)-Reduktion nie in WHNF sind und eine WHNF keinen `letrec`-Ausdruck in einem nicht trivialen Reduktionskontext besitzt.

Das letzte Argument gilt auch für die (iR,brcp-e)-Reduktion. Betrachten wir also diese Regel nur noch im leeren Kontext:

Sei

$$s = (\texttt{letrec}\ x = s', y = S_1[(\texttt{case}\ t' \ldots (pat_i \to S_2[x]) \ldots)], Env\ \texttt{in}\ t'')\ \texttt{und}$$
$$t = (\texttt{letrec}\ y = S_1[(\texttt{case}\ t' \ldots (pat_i \to S_2[s']) \ldots)], Env\ \texttt{in}\ t'')$$

Wenn s eine WHNF ist, dann ist t'' entweder ein Wert (dann ist t auch eine WHNF) oder eine Variable, die über weitere Bindungen an eine Konstruktoranwendung gebunden ist. Aufgrund der Bedingungen der (brcp-e)-Reduktion ist $x = s'$ nicht beteiligt an diesen Bindungen und der an y gebundene Ausdruck kann durch Anwendung der Regel nicht derart verändert werden, dass er vorher eine Konstruktoranwendung, aber danach keine solche ist. Somit ist t ebenso eine WHNF.

Wenn s keine WHNF ist, dann kann auch t keine WHNF sein, da t'' durch die Reduktion nicht verändert wird, die Bindung $x = s'$ wieder nicht relevant ist, da x nicht frei in t'' vorkommen darf, und die Regel die Bindung für y nicht derart verändern kann, dass y zunächst keine Konstruktoranwendung, aber danach eine solche ist. □

Lemma 3.67. *Ein vollständiger Satz an Vertauschungsdiagrammen für (iR,brcp) ist*

$$\xrightarrow{iR,brcp} \cdot \xrightarrow{n,a,P} \quad \rightsquigarrow \quad \xrightarrow{n,a,P} \cdot \xrightarrow{iR,brcp}$$

$$\xrightarrow{iR,brcp} \cdot \xrightarrow{n,lcase} \cdot \xrightarrow{n,(lll)^+} \quad \rightsquigarrow \quad \xrightarrow{n,lcase} \cdot \xrightarrow{n,(lll)^+} \cdot \xrightarrow{iR,brcp}$$

$$\xrightarrow{iR,brcp} \cdot \xrightarrow{n,case} \cdot \xrightarrow{n,(lll)^+} \quad \rightsquigarrow \quad \xrightarrow{n,case} \cdot \xrightarrow{n,(lll)^+}$$

Beweis. Wir führen den Beweis zweiteilig. Wir zeigen zunächst die Vollständigkeit für die (iR,brcp-in)-Reduktion und im Anschluss daran die Vollständigkeit für die (iR,brcp-e)-Reduktion.

Beweis für (iR,brcp-in):

Sei $s = R[s'] = (\texttt{letrec}\ y = s_A,\ Env\ \texttt{in}\ R^-[(\texttt{case}\ t_A\ \ldots(pat_i \to t_i)\ldots)])$ und $t = R[t'] = (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ t_A\ \ldots(pat_i \to \texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)\ldots)])$, d.h. $s \xrightarrow{iR,brcp-in} t$. Wir zeigen die Vollständigkeit des Satzes für (brcp-in), indem wir zeigen, dass für jede Reduktionsfolge $s \xrightarrow{iR,brcp-in} t \xrightarrow{n,a,P} r$ ein anwendbares Diagramm im Satz enthalten ist, oder dass die Folge mithilfe von Normalordnungsreduktionen derart verlängert werden kann, dass ein Diagramm anwendbar ist. Dies ist aufgrund von Lemma 3.66 ausreichend.

Wir unterscheiden danach, welche Normalordnungsreduktion auf t folgt, indem wir die Fälle entsprechend der Definition 3.14 durchgehen, wobei wir diese sinnvoll zusammenfassen.

I. Auf t folgt eine (n,a,P)-Reduktion mit $a \in \{\text{IOlet}, \text{lapp}\}$, d.h. wir betrachten die Fälle 1.a) und 1.b) aus Definition 3.14.

Dann hat t die Form $R_0[R_B^-[(\texttt{letrec}\ Env_B\ \texttt{in}\ t_B)]]$, wobei R_B^- ein schwacher Reduktionskontext der Form $(\texttt{IO}\ [\cdot])$ oder $([\cdot]\ s_B)$ ist.

Da t' in einem Reduktionskontext steht, hat t eine der Formen:

i) $t = R_0[R_B^-[(\texttt{letrec}\ Env_B\ \texttt{in}\ (\texttt{case}\ t_A\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots))]]$

ii) $t = (\texttt{letrec}\ y_1 = R_C^-[R_B^-[(\texttt{letrec}\ Env_B\ \texttt{in}\ t_B)]], y_2 = R_2^-[y_1], \ldots, y_m = R_m^-[y_{m-1}], Env$
$\texttt{in}\ R^-[(\texttt{case}\ R_D^-[y_m]\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)))])$

iii) $t = (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ R_C^-[R_B^-[(\texttt{letrec}\ Env_B\ \texttt{in}\ t_B)]]$
$\ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)\ldots)]$

Die Form für t spezifiziert auch das Aussehen von s. Wie man leicht nachrechnet, ist in allen drei Fällen das erste Diagramm anwendbar, d.h. die Reduktionsfolge $s \xrightarrow{iR,brcp-in} t \xrightarrow{n,a,P} r$ kann durch die Folge $s \xrightarrow{n,a,P} r' \xrightarrow{iR,brcp-in} r$ ersetzt werden.

II. Auf t folgt eine $(n$,lcase)-Reduktion.

Neben Fällen, die analog zu I. sind, kann hier ein weiterer Fall auftreten, wenn der durch die (lcase)-Reduktion betroffene `case`-Ausdruck identisch zum `case`-Ausdruck ist, auf den sich die (iR,brcp-in)-Reduktion bezieht. In diesem Fall ist das zweite Diagramm anwendbar, was wir im Folgenden zeigen:

$$
\begin{array}{rl}
 & (\texttt{letrec}\ y = t_A, Env\ \texttt{in}\ R^-[(\texttt{case}\ (\texttt{letrec}\ Env_B\ \texttt{in}\ t_B)\ldots(pat_i \to t_i)\ldots)]) \\
\xrightarrow{iR,brcp-in} & (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ (\texttt{letrec}\ Env_B\ \texttt{in}\ t_B) \\
 & \qquad \ldots(pat_i \to (\texttt{letrec}\ y = t_A\ \texttt{in}\ t_i))\ldots)]) \\
\xrightarrow{n,lcase} & (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{letrec}\ Env_B \\
 & \qquad \texttt{in}\ (\texttt{case}\ t_B\ \ldots(pat_i \to (\texttt{letrec}\ y = t_A\ \texttt{in}\ t_i))\ldots))]) \\
\xrightarrow{n,(lll)^+} & (\texttt{letrec}\ Env, Env_B\ \texttt{in}\ R^-[(\texttt{case}\ t_B\ \ldots(pat_i \to (\texttt{letrec}\ y = t_A\ \texttt{in}\ t_i))\ldots)]) \\
\hline
 & (\texttt{letrec}\ y = t_A, Env\ \texttt{in}\ R^-[(\texttt{case}\ (\texttt{letrec}\ Env_B\ \texttt{in}\ t_B)\ldots(pat_i \to t_i)\ldots)])
\end{array}
$$

$$\xrightarrow{n,lcase} (\texttt{letrec}\ y = t_A, Env\ \texttt{in}\ R^-[(\texttt{letrec}\ Env_B\ \texttt{in}\ (\texttt{case}\ t_B\ \ldots(pat_i \to t_i)\ldots))])$$
$$\xrightarrow{n,(lll)^+} (\texttt{letrec}\ y = t_A, Env, Env_B\ \texttt{in}\ R^-[(\texttt{case}\ t_B\ \ldots(pat_i \to t_i)\ldots)])$$
$$\xrightarrow{iR,brcp-in} (\texttt{letrec}\ Env, Env_B\ \texttt{in}\ R^-[(\texttt{case}\ t_B\ \ldots(pat_i \to (\texttt{letrec}\ y = t_A\ \texttt{in}\ t_i))\ldots)])$$

III. Auf t folgt eine (n,llet)-Reduktion. Die zugehörigen Fälle 1.d) und 1.e) aus Definition 3.14 und die Bedingung, dass $t = R[t']$, führen zu folgenden Möglichkeiten für t:

i) $t = (\texttt{letrec}\ Env_B\ \texttt{in}$
$\quad(\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ t_A\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)]))$

ii) $t = (\texttt{letrec}\ x_1 = (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ t_A\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)]),$
$\qquad x_2 = R_2^-[x_1], \ldots, R_j^-[x_{j-1}], Env_B$
$\quad\texttt{in}\ R_{j+1}[x_j])$

iii) $t = (\texttt{letrec}\ x_1 = (\texttt{letrec}\ Env_C\ \texttt{in}\ t_C), x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env$
$\quad\texttt{in}\ R^-[(\texttt{case}\ R_B^-[x_j]\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_A))\ldots)])$

Die Berechnung der einzelnen Fälle ergibt, dass stets das erste Diagramm anwendbar ist, wobei in Fall ii) die Reduktionsfolge $s \xrightarrow{iR,brcp-in} t \xrightarrow{n,llet} r$ durch die Folge $s \xrightarrow{n,llet} r' \xrightarrow{iR,brcp-e} r$ ersetzt wird.

IV. Auf t folgt eine (n,case)-Reduktion. Da $t = R[t']$ gelten muss, führt die Betrachtung der zugehörigen Fälle 2.a), 2.b), 2.c) und 4.a) aus der Definition der Normalordnungsreduktion dazu, dass t eine der folgenden Formen haben muss:

i) $t = (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ t_A\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)])$
wobei $t_A = (c_j\ a_1 \ldots a_n)$ oder $t_A = (\lambda v.w)$

ii) $t = (\texttt{letrec}\ Env\ \texttt{in}$
$\qquad R^-[(\texttt{case}\ R_B^-[(\texttt{case}\ t_B\ Alts)]\ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)])$
wobei $t_B = (c_j\ a_1 \ldots a_n)$ oder $t_B = (\lambda v.w)$

iii) $t = (\texttt{letrec}\ x = (\texttt{case}\ t_A\ Alts), x_1 = R^-[x], \ldots, x_j = R_j^-[x_{j-1}]$
$\quad\texttt{in}\ R^-[(\texttt{case}\ R_B^-[x_j]\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)])$
wobei $t_A = (c_j\ a_1 \ldots a_n)$ oder $t_A = (\lambda v.w)$

iv) $t = (\texttt{letrec}\ x_1 = (c_j\ a_1 \ldots a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$\quad\texttt{in}\ R^-[(\texttt{case}\ x_m\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)])$

v) $t = (\texttt{letrec}\ x_1 = (c_j\ a_1 \ldots a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$\quad\texttt{in}\ R^-[(\texttt{case}\ R_B^-[(\texttt{case}\ x_m\ Alts)]\ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)])$

vi) $t = (\texttt{letrec}\ x_1 = (c_j\ a_1 \ldots a_n), x_2 = x_1, \ldots, x_m = x_{m-1},$
$\qquad y_1 = R_0^-[(\texttt{case}\ x_m\ Alts)], y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}], Env$
$\quad\texttt{in}\ R^-[(\texttt{case}\ R_B^-[y_j]\ \ldots(pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)])$

In den Fällen ii), iii), v) und vi) ist das erste Diagramm anwendbar, da der durch die (n,case)-Reduktion betroffene (case)-Ausdruck ein anderer ist, als

derjenige auf den sich die (iR,brcp-in)-Reduktion bezieht.

Die Fälle i) und iv) werden durch das dritte Diagramm abgedeckt, was wir anhand eines Beispiels einer (case-c)-Reduktion in Fall i) illustrieren:

$$
\begin{array}{l}
\quad (\texttt{letrec}\ y = s_A, Env\ \texttt{in}\ R^-[(\texttt{case}\ (c_i\ a_1 \ldots a_n) \ldots ((c_i\ y_1 \ldots y_n) \to t_i) \ldots)]) \\
\xrightarrow{iR,brcp-in} (\texttt{letrec}\ Env\ \texttt{in} \\
\qquad R^-[(\texttt{case}\ (c_i\ a_1 \ldots a_n) \ldots ((c_i\ y_1 \ldots y_n) \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)]) \\
\xrightarrow{n,case} (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{letrec}\ y_1 = a_1, \ldots, y_n = a_n\ \texttt{in}\ (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))]) \\
\xrightarrow{n,(lll)^+} (\texttt{letrec}\ Env, y_1 = a_1, \ldots, y_n = a_n, y = s_A\ \texttt{in}\ R^-[t_i]) \\
\hline
\quad (\texttt{letrec}\ y = s_A, Env\ \texttt{in}\ R^-[(\texttt{case}\ (c_i\ a_1 \ldots a_n) \ldots ((c_i\ y_1 \ldots y_n) \to t_i) \ldots)]) \\
\xrightarrow{n,case} (\texttt{letrec}\ y = s_A, Env\ \texttt{in}\ R^-[(\texttt{letrec}\ y_1 = a_1, \ldots, y_n = a_n\ \texttt{in}\ t_i)]) \\
\xrightarrow{n,(lll)^+} (\texttt{letrec}\ y = s_A, Env, y_1 = a_1, \ldots, y_n = a_n\ \texttt{in}\ R^-[t_i])
\end{array}
$$

V. Auf t folgt eine (n,IOr,P)- oder eine (n, lbeta)- Reduktion, d.h. wir betrachten die Fälle 3.a), 3.b), 3.c) und 4.b) aus Definition 3.14. Da $t = R[t']$ gilt, kann t folgende Formen haben:

i) $t = (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ R_B^-[t_B]\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)])$
wobei $t_B = (\texttt{IO}\ c)$ oder $t_B = ((\lambda v.w)\ s_B)$

ii) $t = (\texttt{letrec}\ x_1 = R_0^-[t_B], x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env$
$\texttt{in}\ R^-[(\texttt{case}\ R_B^-[x_m]\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)])$
wobei $t_B = (\texttt{IO}\ c)$ oder $t_B = ((\lambda v.w)\ s_B)$

iii) $t = (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$\texttt{in}\ R^-[(\texttt{case}\ R_B^-[\texttt{IO}\ x_m]\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)])$

iv) $t = (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1},$
$y_1 = R_1^-[\texttt{IO}\ x_m], y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}], Env$
$\texttt{in}\ R^-[(\texttt{case}\ R_B^-[y_j]\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)])$

Die Berechnung der Fälle ergibt, dass stets das erste Diagramm anwendbar ist. Dies ist auch einsichtig, da die (n,IOr,P)-Reduktion nur lokal den Teilausdruck $(\texttt{IO}\ x_m)$ bzw. $(\texttt{IO}\ c)$ ersetzt und die (n,lbeta)-Reduktion den Teilausdruck $((\lambda v.w)\ s_B)$ ersetzt, wobei diese Unterausdrücke jeweils keinen Einfluss auf die Anwendbarkeit der (iR,brcp-in)-Reduktion haben.

VI. Auf t folgt eine (n,cp)-Reduktion. Aufgrund der Eigenschaft, dass $t = R[t']$ gelten muss, hat t eine der Formen

i) $t = (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$\texttt{in}\ R^-[(\texttt{case}\ R_B^-[x_m]\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)])$

ii) $t = (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1},$
$y_1 = R_1^-[t_B], y_2 = R_2^-[y_1], \ldots, y_j = R_2^-[y_{j-1}], Env$
$\texttt{in}\ R^-[(\texttt{case}\ R_B^-[y_j]\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)])$
wobei $t_B = (x_m\ s)$ oder $t_B = (\texttt{case}\ x_m\ Alts)$

Wiederum ist das erste Diagramm anwendbar, da das Kopieren innerhalb des ersten Argumentes des `case`-Ausdrucks oder in einer Bindung der Umgebung erfolgt, und somit die äußere Struktur des `letrec`-Ausdrucks, wie ihn die (iR,brcp-in)-Reduktion benötigt, nicht verändert wird.

Beweis für (iR,brcp-e):

Sei $s = R[s'] = (\texttt{letrec}\ y = s_A, x = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to t_i)\ldots)], Env\ \texttt{in}\ t_B)$ und $t = R[t'] = (\texttt{letrec}\ x = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = t_A\ \texttt{in}\ t_i))\ldots)], Env\ \texttt{in}\ t_B)$, d.h. $s \xrightarrow{iR,brcp-e} t$. Wiederum genügt es zu zeigen, dass für jede Reduktionsfolge $s \xrightarrow{iR,brcp-e} t \xrightarrow{n,a,P}$ ein anwendbares Diagramm im Satz enthalten ist, oder die Folge verlängert werden kann, so dass ein Diagramm anwendbar ist. Die Fallunterscheidung verläuft identisch zur der im Beweis für (iR,brcp-in):

I. Auf t folgt eine (n, a, P)-Reduktion mit $a \in \{\text{IOlet}, \text{lapp}\}$. Mit $t = R[t']$ und den Regeln 1.a) und 1.b) aus Definition 3.14 folgt, dass t eine der folgenden Formen haben muss, wobei $R_B^- = [(\texttt{IO}\ [\cdot])]$ oder $R_B^- = [([\cdot]\ s_B)]$:

i) $t = R_0[R_B^-[(\texttt{letrec}\ x = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)]\ \texttt{in}\ t_B)]]$

ii) $t = (\texttt{letrec}\ x = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)], Env$
$\texttt{in}\ R_C^-[R_B^-[(\texttt{letrec}\ Env_C\ \texttt{in}\ t_c)]])$

iii) $t = (\texttt{letrec}\ x_1 = R^-[(\texttt{case}\ R_C^-[R_B^-[(\texttt{letrec}\ Env_C\ \texttt{in}\ t_C)]]$
$\ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)],$
$x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}], Env$
$\texttt{in}\ R_{j+1}^-[x_j])$

iv) $t = (\texttt{letrec}\ x_1 = R_1^-[R_B^-[(\texttt{letrec}\ Env_C\ \texttt{in}\ t_C)], x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}],$
$x_k = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i))\ldots)], Env$
$\texttt{in}\ R_{j+1}^-[x_j])$
wobei entweder $2 \le k \le j$ und $t_A = R_D^-[x_{k-1}]$ oder $k > j$

Berechnet man die einzelnen Fälle, so zeigt sich, dass stets das erste Diagramm anwendbar ist.

II. Auf t folgt eine (n,lcase)-Reduktion.

Neben analogen Fällen wie in I. (wobei $R_B^- = (\texttt{case}\ [\cdot]\ Alts)$) gibt es einen

weiteren Fall, in dem das zweite Diagramm anwendbar ist:

$$
\begin{array}{ll}
& (\texttt{letrec } y = s, x_1 = R^-[(\texttt{case } (\texttt{letrec } Env_C \texttt{ in } t_C) \ \ldots (pat_i \to t_i) \ldots)] \qquad (=\text{s}) \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j]) \\
\xrightarrow{iR,brcp} & (\texttt{letrec } x_1 = R^-[(\texttt{case } (\texttt{letrec } Env_C \texttt{ in } t_C) \qquad\qquad (=\text{t}) \\
& \qquad\qquad\qquad \ldots (pat_i \to (\texttt{letrec } y = s_A \texttt{ in } t_i)) \ldots)] \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j]) \\
\xrightarrow{n,lcase} & (\texttt{letrec } x_1 = R^-[(\texttt{letrec } Env_C \texttt{ in } (\texttt{case } t_C \ \ldots (pat_i \to (\texttt{letrec } y = s_A \texttt{ in } t_i)) \ldots))] \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j]) \\
\xrightarrow{n,(lll)^+} & (\texttt{letrec } x_1 = R^-[(\texttt{case } t_C \ \ldots (pat_i \to (\texttt{letrec } y = s_A \texttt{ in } t_i)) \ldots)], Env_C \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j]) \\
\hline
& (\texttt{letrec } y = s, x_1 = R^-[(\texttt{case } (\texttt{letrec } Env_C \texttt{ in } t_C) \ \ldots (pat_i \to t_i) \ldots)] \qquad (=\text{s}) \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j]) \\
\xrightarrow{n,lcase} & (\texttt{letrec } y = s, x_1 = R^-[(\texttt{letrec } Env_C \texttt{ in } (\texttt{case } t_C \ \ldots (pat_i \to t_i) \ldots))] \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j]) \\
\xrightarrow{n,(lll)^+} & (\texttt{letrec } y = s, x_1 = R^-[(\texttt{case } t_C \ \ldots (pat_i \to t_i) \ldots)], Env_C \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j]) \\
\xrightarrow{iR,brcp} & (\texttt{letrec } x_1 = R^-[(\texttt{case } t_C \ \ldots (pat_i \to (\texttt{letrec } y = s_A \texttt{ in } t_i)) \ldots)], Env_C \\
& \qquad\qquad x_2 = R_2^-[x_2], \ldots, x_j = R_j^-[x_{j-1}], Env \\
& \texttt{in } R_{j+1}^-[x_j])
\end{array}
$$

III. Auf t folgt eine (n,llet)-Reduktion.

t hat eine der Formen:

i) $t = (\texttt{letrec } x = R^-[(\texttt{case } t_A \ \ldots (pat_i \to (\texttt{letrec } y = s_A \texttt{ in } t_i)) \ldots)], Env$
$\texttt{in } (\texttt{letrec } Env_C \texttt{ in } t_C))$

ii) $t = (\texttt{letrec } Env$
$\texttt{in } (\texttt{letrec } x = R^-[(\texttt{case } t_A \ldots (pat_i \to (\texttt{letrec } y = s_A \texttt{ in } t_i)) \ldots)], Env_C \texttt{ in } t_C))$

iii) $t = (\texttt{letrec } x_1 = (\texttt{letrec } Env_C \texttt{ in } t_C), x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_j], Env,$
$x_k = R^-[(\texttt{case } t_A \ \ldots (pat_i \to (\texttt{letrec } y = s_A \texttt{ in } t_i)) \ldots)],$
$\texttt{in } R_{j+1}^-[x_j])$

wobei entweder $2 \leq k \leq j$ und $t_A = R_D^-[x_{k-1}]$ oder $k > j$

iv) $t = (\texttt{letrec}$
$x_1 = (\texttt{letrec}\ x = R^-[(\texttt{case}\ t_A \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)], Env_C\ \texttt{in}\ t_C),$
$x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_j], Env$
$\texttt{in}\ R_{j+1}^-[x_j])$

In sämtlichen Fällen ist die Reduktionsfolge $s \xrightarrow{iR,brcp-e} t \xrightarrow{n,llet} r$ durch die Folge $s \xrightarrow{n,llet} r' \xrightarrow{iR,brcp-e} r$ austauschbar, d.h. das erste Diagramm ist anwendbar.

IV. Auf t folgt eine (n,case)-Reduktion.
Die Fälle 2.a), 2.b), 2.c) und 4.a) aus Definition 3.14 sowie die Bedingung, dass $t = R[t']$ führen dazu, dass t eine der folgenden Formen haben muss:

i) $t = (\texttt{letrec}\ Env, x = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)]$
$\texttt{in}\ R_A^-[(\texttt{case}\ t_B\ Alts)]$
wobei $t_B = (c_i\ a_1 \ldots a_n)$ oder $t_B = (\lambda v.w)$.

ii) $t = (\texttt{letrec}\ x_1 = R^-[(\texttt{case}\ t_B\ Alts)], x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}],$
$x_k = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)], Env$
$\texttt{in}\ R_{j+1}^-[x_j]$
wobei $t_B = (c_i\ a_1 \ldots a_n)$ oder $t_B = (\lambda v.w)$ und eine der folgenden Bedingungen gilt:
(a) $k > j$
(b) $2 \le k \le j$ und $t_A = R_A^-[x_{k-1}]$
(c) $k = 1$ und $t_A = R_A^-[(\texttt{case}\ t_B\ Alts)]$
(d) $k = 1$ und $t_A = (c_i\ a_1 \ldots a_n)$

iii) $t = (\texttt{letrec}\ x_1 = (c_i\ a_1 \ldots a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, Env,$
$u = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)]$
$\texttt{in}\ R_0^-[(\texttt{case}\ x_m\ Alts)]$

iv) $t = (\texttt{letrec}\ x_1 = (c_i\ a_1 \ldots a_n), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$y_1 = R_0^-[(\texttt{case}\ x_m\ Alts)], y_2 = R_2^-[y_1], \ldots y_j = R_j^-[y_{j-1}],$
$y_k = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ y = s_A\ \texttt{in}\ t_i)) \ldots)]$
$\texttt{in}\ R_{j+1}^-[y_j]$
wobei genau eine der folgenden Bedingungen erfüllt ist:
(a) $k > j$
(b) $2 \le k \le j$ und $t_A = R_A^-[y_{k-1}]$
(c) $k = 1$ und $t_A = R_A^-[(\texttt{case}\ (x_m)\ Alts)]$
(d) $k = 1$ und $t_A = x_m$

Berechnet man die einzelnen Fälle, so ergibt sich, dass die Fälle i), ii) (a)-(c), iii) und iv) (a)-(c) durch das erste Diagramm abgedeckt sind. Die übrigen Fälle sind genau diejenigen, in denen der `case`-Ausdruck, auf den sich die case-Reduktion bezieht, identisch zum `case`-Ausdruck ist, auf den sich die (brcp-e)-Reduktion bezieht. In diesen Fällen ist das dritte Diagramm anwendbar.

V. Auf t folgt eine (n,IOr,P)- oder eine (n,lbeta)-Reduktion.

t hat eine der Formen:

i) $t = (\texttt{letrec}\ x = R^-[(\texttt{case}\ t_A \ldots (pat_i \to (\texttt{letrec}\ s_A\ \texttt{in}\ t_i))\ldots)], Env\ \texttt{in}\ R_0^-[t_B]$
wobei $t_B = (\texttt{IO}\ c)$ oder $t_B = ((\lambda v.w)\ s_B)$

ii) $t = (\texttt{letrec}\ x_1 = R_1^-[t_B], x_2 = R_2^-[x_1], \ldots, x_j = R_j^-[x_{j-1}],$
$x_k = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ s_A\ \texttt{in}\ t_i))\ldots)], Env$
$\texttt{in}\ R_{j+1}[x_j]$
wobei $t_B = (\texttt{IO}\ c)$ oder $t_B = ((\lambda v.w)\ s_B)$ und eine der folgenden Bedingungen gilt:
(a) $k > j$
(b) $2 \le k \le j$ und $t_A = R_A^-[x_{k-1}]$
(c) $k = 1$ und $t_A = R_A^-[t_B]$

iii) $t = (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env,$
$u = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ s_A\ \texttt{in}\ t_i))\ldots)]$
$\texttt{in}\ R_0^-[(\texttt{IO}\ x_m)]$

iv) $t = (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$y_1 = R_0^-[(\texttt{IO}\ x_m)], y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}],$
$y_k = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ s_A\ \texttt{in}\ t_i))\ldots)]$
$\texttt{in}\ R_{j+1}^-[y_j]$
wobei eine der folgenden Bedingungen erfüllt ist:
(a) $k > j$
(b) $2 \le k \le j$ und $t_A = R_A^-[y_{k-1}]$
(c) $k = 1$ und $t_A = R_A^-[(\texttt{IO}\ x_m)]$

Überprüft man die Fälle, so zeigt sich, dass stets das erste Diagramm anwendbar ist.

VI. Auf t folgt eine (n,cp)-Reduktion. Die Fälle 4.c), 4.d) und 4.e) sowie die Bedingung, dass $t = R[t']$ gilt, ergeben, dass t eine der folgenden Formen haben muss:

i) $t = (\texttt{letrec}\ x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$u = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ s_A\ \texttt{in}\ t_i))\ldots)]$
$\texttt{in}\ R_0^-[x_m]$

ii) $t = (\texttt{letrec}\ x_1^= (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, Env$
$y_1 = R_0^-[t_B], y_2 = R_2^-[y_1], \ldots, y_j = R_j^-[y_{j-1}],$
$y_k = R^-[(\texttt{case}\ t_A\ \ldots (pat_i \to (\texttt{letrec}\ s_A\ \texttt{in}\ t_i))\ldots)]$
$\texttt{in}\ R_{j+1}^-[y_j]$
wobei $t_B = (\texttt{case}\ x_m\ Alts)$ oder $t_B = [(x_m\ s)]$ und eine der folgenden Bedingungen gilt:
(a) $k > j$
(b) $2 \le k \le j$ und $t_A = R_A^-[y_{k-1}]$
(c) $k = 1$ und $t_A = R_A^-[t_B]$
(d) $k = 1$, $t_B = (\texttt{case}\ x_m\ Alts)$ und $t_A = x_m$

Berechnet man die Fälle, so zeigt sich, dass wiederum in allen Fällen das erste Diagramm anwendbar ist.

□

Lemma 3.68. *Ein vollständiger Satz an Gabeldiagrammen für (iR,brcp) ist*

$$\xleftarrow{n,a,P} \cdot \xrightarrow{iR,brcp} \quad \rightsquigarrow \quad \xrightarrow{iR,brcp} \xleftarrow{n,a,P}$$

$$\xleftarrow{n,(lll)^+} \cdot \xleftarrow{n,lcase} \cdot \xrightarrow{iR,brcp} \quad \rightsquigarrow \quad \xrightarrow{iR,brcp} \cdot \xleftarrow{n,(lll)^+} \cdot \xleftarrow{n,lcase}$$

$$\xleftarrow{n,(lll)^+} \cdot \xleftarrow{n,case} \cdot \xrightarrow{iR,brcp} \quad \rightsquigarrow \quad \xleftarrow{n,(lll)^+} \cdot \xleftarrow{n,lcase}$$

Beweis. Die Diagramme können mithilfe der Vertauschungsdiagramme gefunden werden. Eine Fallunterscheidung wie im Beweis zu Lemma 3.67 zeigt die Vollständigkeit des Satzes. □

Lemma 3.69. *Die Regel (brcp) kann nur endlich oft angewendet werden.*

Beweis. Durch Anwendung einer (brcp)-Reduktion wird eine Bindung nach innen in den Ausdruck verschoben. Da die Tiefe eine Ausdrucks beschränkt ist, kann nur endlich oft nach innen verschoben werden. □

Satz 3.70. *Die Regel (brcp) ist eine korrekte Programmtransformation.*

Beweis. Der Beweis verläuft analog wie der Beweis von Lemma 3.45 in Verbindung mit Lemma 3.66 und den vollständigen Sätzen an Vertauschungs- und Gabeldiagrammen. □

3.9.3 Korrektheit von (ucpb)

Wir können nun die Korrektheit von (ucpb) zeigen, indem wir die Ausdrücke aus der Regel nur mithilfe von korrekten Programmtransformationen ineinander überführen.

Satz 3.71. *(ucpb) ist eine korrekte Programmtransformation.*

Beweis. Wir zeigen hier nur den Beweis für (ucpb-in), die Überführung für (ucpb-e) verläuft analog.

$$\begin{aligned}
&(\texttt{letrec}\ x = s, Env\ \texttt{in}\ S_1[(\texttt{case}\ t\ \ldots(pat_j \to t_j)\ldots(pat_i \to S_2[x])\ldots)]) \\
\xleftarrow{(gc)}\ &(\texttt{letrec}\ x = s, Env\ \texttt{in} \\
&\quad S_1[(\texttt{letrec}\ y = x\ \texttt{in}\ (\texttt{case}\ t\ \ldots(pat_j \to t_j)\ldots(pat_i \to S_2[x])\ldots))]) \\
\xleftarrow{(cpx)^*}\ &(\texttt{letrec}\ x = s, Env\ \texttt{in} \\
&\quad S_1[(\texttt{letrec}\ y = x\ \texttt{in}\ (\texttt{case}\ t\ \ldots(pat_j \to t_j[y/x])\ldots(pat_i \to S_2[y])\ldots))])
\end{aligned}$$

$$
\begin{array}{rl}
\xrightarrow{ucp} & (\texttt{letrec}\ Env\ \texttt{in} \\
& \quad S_1[(\texttt{letrec}\ y = s\ \texttt{in}\ (\texttt{case}\ t\ \ldots (pat_j \to t_j[y/x]) \ldots (pat_i \to S_2[y]) \ldots))]) \\
\xrightarrow{brcp} & (\texttt{letrec}\ Env\ \texttt{in} \\
& \quad S_1[(\texttt{case}\ t\ \ldots (pat_j \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_j[y/x])) \\
& \qquad\qquad \ldots (pat_i \to (\texttt{letrec}\ y = s\ \texttt{in}\ S_2[y])) \ldots)]) \\
\xrightarrow{ucp} & (\texttt{letrec}\ Env\ \texttt{in} \\
& \quad S_1[(\texttt{case}\ t\ \ldots (pat_j \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_j[y/x])) \ldots (pat_i \to (\texttt{letrec}\ \{\}\ \texttt{in}\ S_2[s])) \ldots)]) \\
\xleftarrow{gc} & (\texttt{letrec}\ Env\ \texttt{in} \\
& \quad S_1[(\texttt{case}\ t\ \ldots (pat_j \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_j[y/x])) \\
& \qquad\qquad \ldots (pat_i \to (\texttt{letrec}\ y = s\ \texttt{in}\ S_2[s])) \ldots)]) \\
\xleftarrow{brcp} & (\texttt{letrec}\ Env\ \texttt{in}\ S_1[(\texttt{letrec}\ y = s\ \texttt{in}\ (\texttt{case}\ t\ \ldots (pat_j \to t_j[y/x]) \ldots (pat_i \to S_2[s]) \ldots))]) \\
\xleftarrow{gc} & (\texttt{letrec}\ x = s, Env\ \texttt{in} \\
& \quad S_1[(\texttt{letrec}\ y = s\ \texttt{in}\ (\texttt{case}\ t\ \ldots (pat_j \to t_j[y/x]) \ldots (pat_i \to S_2[s]) \ldots))]) \\
\xleftarrow{ucp} & (\texttt{letrec}\ x = s, Env\ \texttt{in} \\
& \quad S_1[(\texttt{letrec}\ y = x\ \texttt{in}\ (\texttt{case}\ t\ \ldots (pat_j \to t_j[y/x]) \ldots (pat_i \to S_2[s]) \ldots))]) \\
\xrightarrow{(cpx)^*} & (\texttt{letrec}\ x = s, Env\ \texttt{in} \\
& \quad S_1[(\texttt{letrec}\ y = x\ \texttt{in}\ (\texttt{case}\ t\ \ldots (pat_j \to t_j[x]) \ldots (pat_i \to S_2[s]) \ldots))]) \\
\xrightarrow{gc} & (\texttt{letrec}\ x = s, Env\ \texttt{in}\ S_1[(\texttt{case}\ t\ \ldots (pat_j \to t_j) \ldots (pat_i \to S_2[s]) \ldots)])
\end{array}
$$

□

3.10 Striktheitsoptimierung

Im Folgenden definieren wir eine Programmtransformation, die Striktheit benötigt. Wir beweisen die Korrektheit der Transformation nicht, sondern begründen diese.

Zunächst definieren wir Striktheit in etwas vereinfachter Weise als in [Sch03a, Definition 22.1] sowie die Transformation (streval) für den FUNDIO-Kalkül.

Anschließend stellen wir eine begründete Vermutung auf.

Definition 3.72. *Eine Abstraktion s heißt* strikt*, wenn gilt $(s\ \bot) \sim_c \bot$, wobei $\bot$ ein nichtterminierender Ausdruck ist.*

Definition 3.73. *Die Regel (streval) sei wie folgt definiert:*

$$
\begin{array}{ll}
\text{(streval)} & ((\lambda y.s)\ t) \\
& \longrightarrow (\texttt{letrec}\ w = t\ \texttt{in} \\
& \qquad (\texttt{case}\ w\ (pat_1 \to ((\lambda y.s)\ w)) \ldots (pat_N \to ((\lambda y.s)\ w)))) \\
& \text{wenn } (\lambda y.s) \text{ eine strikte Abstraktion ist}
\end{array}
$$

Vermutung 3.74. *(streval) ist eine korrekte Programmtransformation.*

Begründung: Wir beziehen uns auf [Sch03a, Abschnitte 20, 21 und 22], wobei wir die Definitionen für eine *Auswertungsstrategie* ([Sch03a, Definition 20.4]), *relative Normalordnung* ([Sch03a, Definition 21.4]), *relative WHNF* ([Sch03a, Definition 21.3]) und *s-strikte Normalordnungsreduktion* ([Sch03a, Definition 22.8]) hier aufgrund des Umfangs nicht angeben, sondern auf die Quelle verweisen.

Aus [Sch03a, Theorem 21.12] folgt, dass folgende Auswertungsstrategie korrekt ist:

Sei $t' = R[((\lambda y.s)\ t)]$ geschlossen, wobei $(\lambda y.s)$ eine geschlossene strikte Abstraktion ist. Dann ist es korrekt, zunächst t zu einer rWHNF t_W auszuwerten und danach $R[((\lambda x.s')\ t_W)]$ weiter in Normalordnung auszuwerten.

Wir nehmen an, dass dies auch gilt, wenn $(\lambda y.s)$ offen ist.

Die Transformation (streval) modelliert diese Strategie, indem sie das in der Strategie explizite Auswerten von t implizit dadurch erreicht, dass die Transformation mittels des `case`-Ausdrucks die Auswertung von t erzwingt.

Im Weiteren vergleichen wir die Auswertungen zweier allgemeiner Ausdrücke, die durch Anwendung einer (streval)-Reduktion ineinander überführt werden können:

Werde (streval) in einem Kontext C angewendet, d.h. $C[u] \xrightarrow{iC,streval} C[v]$, wobei $C[u]$ und $C[v]$ geschlossen sind.

Sei RED_u (bzw. RED_v) die no-Reduktionsfolge von $C[u]$ (bzw. $C[v]$).

Wir unterscheiden in folgende Fälle:

1. RED_u wertet den Teilterm u nicht aus.

   Dann wertet RED_v den Teilterm v ebenso nicht aus.

2. RED_u wertet den Teilterm u aus.

   In diesem Fall enthält RED_u eine (lbeta)-Reduktion, deren Redex $((\lambda y.s)\ t)$ ist, d.h. der auszuwertende Ausdruck zu diesem Zeitpunkt ist $R[u]$, wobei R ein Reduktionskontext ist. In RED_v wird anstelle dieser (lbeta)-Reduktion zunächst (nach möglichen (lll)-Reduktionen) t ausgewertet (genauer: Der auszuwertende Ausdruck ist dann $R'[t]$, für einen weiteren Reduktionskontext R').

   a) $t \sim_c \bot$

      Wenn die Auswertung von t in RED_v nicht terminiert, d.h. $t \sim_c \bot$, dann terminiert aufgrund von Satz 3.33 die Auswertung von $R'[t]$ und somit die Auswertung von $C[v]$ nicht.

      Die Auswertung von $C[u]$ terminiert dann ebenfalls nicht: $(\lambda y.s)$ ist eine strikte Abstraktion und somit gilt $R[u] \sim_c R[(\lambda y.s)\ \bot] \sim_c R[\bot] \sim_c \bot$ aufgrund von Definition 3.72 und Satz 3.33

   b) $t \xrightarrow{n^*} t_W$ mit t_W WHNF.

      D.h. die Auswertung von t in RED_v terminiert und nach der Auswertung existiert eine Bindung $w = t_W$ und RED_v reduziert den Redex $((\lambda y.s)\ w)$,

wobei zusätzlich (lll)-Reduktionen in RED_v enthalten sein können, die jedoch die kontextuelle Gleichheit erhalten.

Der wesentliche Unterschied zwischen [Sch03a, Theorem 22.12] und Vermutung 3.74 besteht darin, dass in [Sch03a, Theorem 22.12] eine Strategie benutzt wird, um Striktheitsoptimierung durchzuführen, d.h. die Auswertungsreihenfolge wird zur Laufzeit verändert, so dass zum Teil nicht in Normalordnung reduziert wird, während (streval) eine Programmtransformation ist, d.h. vor Auswerten des Ausdrucks in Normalordnung wird die Termstruktur verändert (zur Compilezeit); zur Laufzeit kann dann in Normalordnung ausgewertet werden.

So gesehen kann man die Sicht vertreten, dass die Anwendung der (streval)-Transformation zur Compilezeit eine Implementierung der s-strikten Strategie darstellt.

3.11 Ergebnisse

Im folgenden Theorem halten wir fest, dass sämtliche neu eingeführten Transformationen bis auf (streval) korrekt sind.

Theorem 3.75. *Die Programmtransformationen (capp), (ccpcx), (lcshift), (ccase), (ccase-in), (crpl), (cpcheap), (brcp) und (ucpb) sind korrekt.*

Beweis. Folgt aus den Sätzen 3.45, 3.50, 3.51, 3.56, 3.57, 3.62, 3.65, 3.70 und 3.71. □

Nach dieser ausführlichen Betrachtung des FUNDIO-Kalküls beschäftigen wir uns im nächsten Kapitel mit Transformationen im GHC, wobei wir die Resultate dieses Kapitels benutzen werden.

Kapitel 4

FUNDIO als Semantik für die Kernsprache des GHC

In diesem Kapitel prüfen wir die Korrektheit einer Vielzahl der Programmtransformationen, die im GHC durchgeführt werden. Hierfür übersetzen wir die Kernsprache des GHC in L_{FUNDIO} und prüfen anschließend, ob die Transformationen korrekt bezüglich der FUNDIO-Semantik sind.

4.1 Übersetzung der GHC-Kernsprache in FUNDIO

Der FUNDIO-Kalkül ist ungetypt, aufgrund dessen betrachten wir die Kernsprache des GHC ebenfalls ungetypt in der Darstellung wie wir sie in Definition 2.1 definiert haben.

4.1.1 Die Übersetzung $[\![\cdot]\!]$

Wir definieren nun eine Übersetzung $[\![\cdot]\!]$, die (ungetypte) Ausdrücke der GHC-Kernsprache in Ausdrücke des FUNDIO-Kalküls übersetzt.

Definition 4.1. *Sei $e \in L_{GHCCore}$, dann ist $[\![e]\!] \in L_{FUNDIO}$ der übersetzte Ausdruck. In den Abbildungen 4.1, 4.2 und 4.3 sind die Übersetzungsregeln angegeben. Die einzelnen Schritte einer Übersetzung trennen wir mit dem Symbol $\equiv$.*

Die Übersetzung eines Ausdrucks der GHC-Kernsprache erfolgt komponentenweise anhand seiner Termstruktur von oberster Ebene nach unten. Die Übersetzung ist insofern sinnvoll, da die einzelnen Konstrukte, wie `case`, `letrec`, Abstraktion und Applikation in dieselben Konstrukte des FUNDIO-Kalküls übersetzt werden, wann immer dies möglich ist. Im Folgenden betrachten wir einige Spezialfälle genauer.

- In $L_{GHCCore}$ decken die Alternativen eines `case`-Ausdrucks i.A. nicht alle Konstruktoren ab. In FUNDIO hingegen müssen stets Alternativen für

Programm:

$[\![binding_1; \ldots; \text{main} = t; \ldots; binding_n]\!]$
$\equiv$ (`letrec` $[\![binding_1]\!], \ldots, \text{main} = [\![t]\!], \ldots,$ $[\![binding_n]\!]$ `in` main)

$[\![binding_1; \ldots; \texttt{rec}\ \{bind_{i,1}; \ldots; \text{main} = t; \ldots; bind_{i,n_i}\}; \ldots; binding_n]\!]$
$\equiv$ (`letrec` $[\![binding_1]\!]$, ...,
$[\![bind_{i,1}]\!], \ldots, \text{main} = [\![t]\!], \ldots, [\![bind_{i,n_i}]\!],$
$\ldots, [\![binding_n]\!]$
`in` main)

Bindungen:

$[\![x = t]\!] \equiv x = [\![t]\!]$

$[\![\texttt{rec}\ \{x_1 = t_1; \ldots; x_n = t_n\}]\!] \equiv x_1 = [\![t_1]\!], \ldots, x_n = [\![t_n]\!]$

Applikation:

$[\![t\ a]\!] \equiv ([\![t]\!]\ [\![a]\!])$

Abstraktion:

$[\![\lambda \text{var}_1 \ldots \text{var}_n \texttt{->}\ t]\!] \equiv (\lambda \text{var}_1.\ (\ldots (\lambda \text{var}_n.\ [\![t]\!]) \ldots)$

let-Ausdruck:

$[\![\texttt{let}\ v = s\ \texttt{in}\ t]\!] \equiv$ (`letrec` $v = [\![s]\!]$ `in` $[\![t]\!]$)

letrec-Ausdruck:

$[\![\texttt{letrec}\ v_1 = s_1; \ldots; v_n = s_n\ \texttt{in}\ t]\!]$
$\equiv$ (`letrec` $v_1 = [\![s_1]\!], \ldots, v_n = [\![s_n]\!]$ `in` $[\![t]\!]$)

Konstruktor:

$[\![c]\!] \equiv (\lambda x_1.(\lambda x_2. \ldots (\lambda x_{ar(c)}.(c\ x_1 \ldots x_{ar(c)})) \ldots))$

Variable:

$[\![x]\!] \equiv x$
wenn x eine Variable ist.

Literal:

$[\![unboxed\ value]\!] \equiv c_i$
wobei es für jeden ausgepackten Wert eine Konstante c_i gibt.

Abbildung 4.1: Übersetzung von $L_{GHCCore}$ nach L_{FUNDIO}

Pattern:

$[\![c\ a_1 \ldots a_{ar(c)}]\!] \equiv (c\ a_1 \ldots a_{ar(c)})$
wenn $c\ a_1 \ldots a_{ar(c)}$ ein Pattern ist.

case ohne default-Alternative:

$[\![$`case` t `of` pat_1 `->` t_1; $\ldots$ pat_n `->` $t_n;]\!]$
$\equiv$ (`case` $[\![t]\!]$
($[\![pat_1]\!]$ `->` $[\![t_1]\!]) \ldots ([\![pat_n]\!]$ `->` $[\![t_n]\!])$ (pat_{n+1} `->` $\bot) \ldots (pat_N$ `->` $\bot))$
$\bot$ sei ein nichtterminierender Ausdruck in L_{FUNDIO}, $pat_{n+1}, \ldots pat_N$ seien die fehlenden Pattern für die Konstruktoren aus $\mathcal{C}$ und sind somit von folgender Form: wenn pat_i den Konstruktor c_i abdeckt so ist $pat_i = c_i\ a_1\ \ldots\ a_{ar(c_i)}$, für $i = n+1, \ldots, N-1$ und $pat_N =$ `lambda`.

case mit Alternativen und default-Alternative:

$[\![$`case` t `of` pat_1 `->` $t_1; \ldots; pat_n$ `->` $t_n; x$ `->` $s]\!]$
$\equiv$ (`letrec` $y = [\![t]\!]$ `in`
(`case` y
($[\![pat_1]\!]$ `->` $[\![t_1]\!]) \ldots ([\![pat_n]\!]$ `->` $[\![t_n]\!])$
($[\![pat_{n+1}]\!]$ `->` $[\![s[y/x]]\!]) \ldots$
($[\![pat_m]\!]$ `->` $[\![s[y/x]]\!]))$
(pat_{m+1} `->` $\bot)\ \ldots$
(pat_N `->` $\bot))$
wenn die pat_i, $i = 1, \ldots, n$ von einem Typ mit $m \geq n$ Konstruktoren sind. $pat_{n+1}, \ldots pat_m$ seien die fehlenden Pattern für die Konstruktoren von diesem Typ. $pat_{m+1}, \ldots pat_N$ decken die restlichen Konstruktoren in L_{FUNDIO} ab. y sei eine neue Variable.

case einzig mit default-Alternative:

$[\![$`case` t `of` x `->` $s]\!]$
$\equiv$ (`letrec` $y = [\![t]\!]$ `in`
(`case` y
(pat_1 `->` $[\![s[y/x]]\!]$)
...
(pat_N `->` $[\![s[y/x]]\!])))$
y sei eine neue Variable.

Abbildung 4.2: Übersetzung von $L_{GHCCore}$ nach L_{FUNDIO} (Forts.)

Primitive Operatoren:
Handelt es sich um Operatoren, die keine Seiteneffekte beinhalten, so übersetzen wir diese als Funktionen, die alle Kombinationen von möglichen Eingaben (das sind endliche viele) testen und jeweils die entsprechende Konstante zurück geben. Für die primitive Addition (`+#`) von `Int#`-Werten sieht die entsprechende Übersetzung beispielsweise folgendermaßen aus:

$$\begin{aligned}
[\![\texttt{+\#}]\!] \equiv\ & (\lambda a_1.(\lambda a_2.(\texttt{case}\ a_1\ ([\![\texttt{-2147483648\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots) \\
& ([\![\texttt{-2147483647\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots) \\
& \ldots \\
& ([\![\texttt{1\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots ([\![\texttt{1\#}]\!] \rightarrow [\![\texttt{2\#}]\!])([\![\texttt{2\#}]\!] \rightarrow [\![\texttt{3\#}]\!]) \ldots) \\
& \ldots \\
& ([\![\texttt{2147483647\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots) \\
& (p_n \rightarrow \bot) \ldots (p_N \rightarrow \bot)\ )))
\end{aligned}$$

Seiteneffekt-behaftete Operatoren werden mithilfe des `IO`-Konstruktes übersetzt. Wir demonstrieren dies beispielhaft an den Funktionen `getChar` und `putChar`:

$$\begin{aligned}
[\![\texttt{getChar}]\!] \equiv\ & ([\![IO]\!]\ (\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \rightarrow (w, p_1)) \ldots (p_n \rightarrow (w, p_n)) \\
& \qquad (p_{n+1} \rightarrow \bot) \ldots (p_N \rightarrow \bot))))
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{putChar}]\!] \equiv\ & (\lambda x.([\![IO]\!]\ (\lambda w.(\texttt{case}\ x \\
& \qquad (p_1 \rightarrow (\texttt{case}\ (\texttt{IO}\ x)\ (p_1 \rightarrow (w, [\![\texttt{()}]\!])) \ldots (p_N \rightarrow (w, [\![\texttt{()}]\!])))) \\
& \qquad \ldots \\
& \qquad (p_n \rightarrow (\texttt{case}\ (\texttt{IO}\ x)\ (p_1 \rightarrow (w, [\![\texttt{()}]\!])) \ldots (p_N \rightarrow (w, [\![\texttt{()}]\!])))) \\
& \qquad (p_{n+1} \rightarrow \bot) \ldots (p_N \rightarrow \bot)))))
\end{aligned}$$

wobei:

- $p_1; \ldots, p_n$, sind Pattern für die Konstruktoren aus der Menge der Zeichen, die eine Teilmenge von $\mathcal{C}$ ist, und $p_{n+1}, \ldots p_N$ sind Pattern für die restlichen Konstruktoren aus $\mathcal{C}$:
- $\mathcal{B}$ ist ein besonderes „Blank-Symbol“ sei, dass wir den Konstruktoren von FUNDIO hinzufügen.
- $[\![IO]\!]$ ist die Übersetzung des Konstruktors IO aus der GHC-Kernsprache.
- $[\![\texttt{()}]\!]$ ist die Übersetzung des Konstruktors `()` aus der GHC-Kernsprache.

Abbildung 4.3: Übersetzung von $L_{GHCCore}$ nach L_{FUNDIO} (Forts.)

sämtliche Konstruktoren vorhanden sein. Aufgrund dessen fügen wir während der Übersetzung genügend Alternativen hinzu, die alle zu einem nichtterminierenden Ausdruck führen.

- `case`-Ausdrücke, die eine default-Alternative enthalten, können nicht direkt übersetzt werden, da im FUNDIO-Kalkül ein solches Konstrukt nicht gegeben ist. Deshalb übersetzen wir Ausdrücke dieser Form als einen `case`-Ausdruck, der die durch die default-Alternative abgedeckten Konstruktoren durch einzelne zusätzliche Alternativen abdeckt. Da die default-Alternative das erste Argument des `case`-Ausdrucks in ausgewerteter Form durch eine Variable bindet, fügen wir dem übersetzten Ausdruck ein geeignetes `letrec`-Konstrukt hinzu, um den Ausdruck ebenfalls zu binden, damit er weiter verwendet werden kann.

- Unboxed values enthält der FUNDIO-Kalkül nicht. Da diese aber nur eine endliche Menge von primitiven Werten formen, übersetzen wir sie als Konstanten, die wir der Menge $\mathcal{C}$ der Konstruktoren hinzufügen.

  Unboxed tuples sind zwar im GHC spezielle Konstruktoren, die Effizienzvorteile bringen, in FUNDIO werden sie jedoch durch normale Konstruktoren simuliert.

- Primitive Funktionen ohne Seiteneffekte werden in Funktionen übersetzt, die strikt in allen Argumenten sind. Die Striktheit wird erzeugt, indem die Auswertung der Argumente mithilfe von `case`-Ausdrücken erzwungen wird.

- Seiteneffekt-behaftete Funktionen werden derart übersetzt, dass die Seiteneffekte durch das `IO`-Konstrukt des FUNDIO-Kalküls simuliert werden. So ergibt sich beispielsweise die Übersetzung von `getChar`, wie sie in Abbildung 4.3 angegeben ist, wie folgt: Da `getChar` eine IO-Aktion ist, müssen wir eine durch den Konstruktor IO verpackte Funktion zurück geben, die als Argument einen Zustand der Welt bekommt und ein Paar bestehend aus dem neuen Zustand und einem Zeichen zurück liefert. Das `case`-Konstrukt sorgt dafür, dass der `IO`-Ausdruck ausgewertet ist, bevor auf den neuen Zustand zugegriffen werden kann und dass nur Zeichen als Resultat akzeptiert werden.

  Bei der Übersetzung von `putChar` wird (mithilfe eines `case`-Ausdrucks) geprüft, ob das Argument zu einem Zeichen ausgewertet werden kann.

4.1.2 Beispiele

Im Folgenden zeigen wir, wie die Funktion `unsafePerformIO` in FUNDIO übersetzt wird, und welcher Zusammenhang zwischen `unsafePerformIO` und dem nichtdeterministischen `IO` des FUNDIO-Kalküls besteht.

Eine leicht vereinfachte Definition des `unsafePerformIO` in Haskell ist:

```
unsafePerformIO (IO m) = case m realWorld# of (s, r) -> r
```

Dieser Ausdruck stellt sich dann in $L_{GHCCore}$ wie folgt dar:

$$\begin{aligned}\texttt{unsafePerformIO} = \lambda i \texttt{ -> case } i \texttt{ of}\\ (IO\ m) \texttt{ -> case } m \text{ realWorld\# } \texttt{of}\\ (s, r) \texttt{ -> } r\end{aligned}$$

Beispiel 4.2. *Die Berechnung im Anhang in Abschnitt A.3 zeigt:*

$$\begin{aligned}&[\![\texttt{unsafePerformIO getChar}]\!]\\ \sim_c\ &(\texttt{case } (\texttt{IO } \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))\end{aligned}$$

Hierbei sind $p_1, \ldots, p_n$ Pattern für die Elemente des Zeichensatzes, der eine Teilmenge der Menge $\mathcal{C}$ der Konstruktoren des FUNDIO-Kalküls ist.

Die Übersetzung des Ausdrucks gleicht dem nichtdeterministischen `IO`*-Konstrukt des FUNDIO-Kalküls, wobei der zusätzliche* `case`*-Ausdruck aufgrund dessen entsteht, dass* `getChar` *nur Zeichen und keine anderen Konstanten liefern darf.*

Im Weiteren übersetzen wir manchmal den Ausdruck `unsafePerformIO getChar` direkt in den `case`-Ausdruck des obigen Beispiels, da der Ausdruck kontextuell äquivalent zur eigentlichen Übersetzung ist.

Beispiel 4.3. *Im Anhang in Abschnitt A.4 wird gezeigt, dass folgendes gilt:*

$$\begin{aligned}&[\![\lambda c \texttt{ -> unsafePerformIO } (\texttt{putChar } c)]\!]\\ \sim_c\ &(\lambda c.(\texttt{case } c\ (p_1 \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to ([\![\texttt{()}]\!])) \ldots\ (p_N \to ([\![\texttt{()}]\!])))))\\ &\qquad \ldots\\ &\qquad (p_n \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to ([\![\texttt{()}]\!])) \ldots\ (p_N \to ([\![\texttt{()}]\!]))))\\ &\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))\end{aligned}$$

Der Ausdruck ist damit sehr ähnlich zum Audruck $(\lambda c.(\texttt{IO } c))$, wobei jedoch die zusätzlichen `case`*-Ausdrücke sowohl dafür sorgen, dass nur Zeichen ausgegeben werden, als auch dafür, dass ein erhaltener Eingabewert verworfen und der (übersetzte) Konstruktor* `()` *als Ergebnis zurück gegeben wird.*

Die Übersetzung $[\![\cdot]\!]$ transformiert Konstruktoren mit positiver Stelligkeit in Abstraktionen. Dementsprechend werden Konstruktoranwendungen als Anwendungen auf Abstraktionen übersetzt. Im Folgenden zeigen wir, dass gesättigte Konstruktoranwendungen auch direkt als solche nach L_{FUNDIO} übersetzt werden können.

Beispiel 4.4. *Sei $c\ a_1 \ldots a_n \in L_{GHCCore}$ eine gesättigte Konstruktoranwendung, dann ist der übersetzte Ausdruck in L_{FUNDIO} kontextuell äquivalent zu einer Kon-*

struktoranwendung:

$$
\begin{aligned}
& [\![c\ a_1 \ldots a_n]\!] \\
\equiv\ & (\ldots((\lambda x_1.(\ldots(\lambda x_n.([\![c]\!]\ x_1 \ldots x_n))\ldots))\ [\![a_1]\!])\ldots[\![a_n]\!]) \\
\xrightarrow{lbeta}\ & (\texttt{letrec}\ x_1 = [\![a_1]\!]\ \texttt{in}\ (\ldots((\lambda x_2.(\ldots(\lambda x_n.([\![c]\!]\ x_1 \ldots x_n))\ldots))\ [\![a_2]\!])\ldots[\![a_n]\!])) \\
\xrightarrow{(lll)^*}\ & (\texttt{letrec}\ x_1 = [\![a_1]\!], \ldots x_n = [\![a_n]\!]\ \texttt{in}\ ([\![c]\!]\ x_1 \ldots x_n)) \\
\xrightarrow{(ucp)^*}\ & (\texttt{letrec}\ \{\}\ \texttt{in}\ ([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!])) \\
\xrightarrow{gc}\ & ([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!])
\end{aligned}
$$

Im Folgenden werden wir aufgrund dessen gesättigte Konstruktoranwendungen auch direkt in solche übersetzen.

4.1.3 Korrektheit von Programmtransformationen in $L_{GHCCore}$

Wir definieren nun die Korrektheit einer Programmtransformation in $L_{GHCCore}$, indem wir die Transformation zunächst nach L_{FUNDIO} übersetzen und dann die kontextuelle Äquivalenz des FUNDIO-Kalküls benutzen.

Definition 4.5. ($[\![\cdot]\!]$-Korrektheit)

Sei P eine Programmtransformation auf Ausdrücken $s, t \in L_{GHCCore}$. Wir nennen P $[\![\cdot]\!]$-korrekt*, wenn gilt: $s\ P\ t \implies [\![s]\!] \sim_c [\![t]\!]$*

Hinsichtlich dieser Korrektheit werden wir nun die im GHC angewendeten Transformationen untersuchen.

4.2 Klassifizierung der Transformationen auf der Kernsprache

Die Transformationen auf der GHC-Kernsprache werden nach [PS94, San95, PS98] in zwei Klassen geteilt.

Zum einen gibt es *lokale Transformationen*, die kleine Teilausdrücke transformieren. Die Mächtigkeit dieser Transformationen liegt darin, dass sie zusammen und mehrmals angewendet werden. Die lokalen Transformationen werden im so bezeichneten „Simplifier“ zusammengefasst.

Zum anderen gibt es *globale Transformationen*, wie z.B. Striktheitsanalyse und „common subexpression elimination“. Jede dieser Transformationen ist als einzelner Durchlauf implementiert und kann separat an- und abgeschaltet werden.

Nach vielen der globalen Transformation findet ein Aufruf des Simplifiers statt, um den Code „zu bereinigen“[1]. Aufgrund dessen ist es besonders wichtig, sicherzustellen,

[1] Vgl. [PS98, Seite 3].

dass nur korrekte lokale Transformationen ausgeführt werden, die wir im Folgenden ausführlich untersuchen werden.

Im Anschluss daran gehen wir auf die globalen Transformationen ein, wobei wir diese jedoch nicht eingehend untersuchen werden, da dies den Umfang dieser Arbeit sprengen würde.

4.3 Lokale Transformationen

Wir geben nun eine Übersicht über die lokalen Transformationen, die im GHC angewendet werden. Die hier dargestellten Transformationen werden detailliert in [PS94] und [San95] beschrieben, wobei sich die dort verwendete Kernsprache von der aktuellen Kernsprache und von $L_{GHCCore}$ unterscheidet. Deshalb haben wir die Transformationen entsprechend der aktuellen Implementierung angepasst.

Im Folgenden bezeichnen wir einen Ausdruck als *atomar*, wenn er ein Literal oder eine Variable ist.

Wir benutzen für die Transformationen teilweise die englischen Begriffe, damit sie leicht in der angegebenen Literatur wiederzufinden sind und da adäquate Übersetzungen in die deutsche Sprache schwer zu finden sind.

Im Weiteren wird zunächst die jeweilige im GHC durchgeführte Transformation angegeben, wobei wir eine Transformation mit dem Namen *Regel*, die Ausdrücke der Form von l in Ausdrücke der Form von r überführt, als

$$l \overset{(Regel)}{\texttt{===>}} r$$

notieren.

Danach übersetzen wir die linke und rechte Seite mittels der Übersetzung $[\![\cdot]\!]$ in Ausdrücke des FUNDIO-Kalküls und versuchen dann, die $[\![\cdot]\!]$-Korrektheit der Transformation mithilfe der korrekten Programmtransformationen aus den Sätzen 3.31 und 3.32 sowie Theorem 3.75 zu zeigen. Ist die jeweils betrachtete Programmtransformation nicht korrekt, so werden wir dies anhand von Gegenbeispielen zeigen.

Wir benutzen analog zu „Kontexten“ im FUNDIO-Kalkül, auch allgemeine Kontexte in $L_{GHCCore}$, die wir als $C[\cdot]$ darstellen. Auf eine explizite Definition verzichten wir jedoch.

4.3.1 Beta-Reduktion

In [San95, Abschnitt 3.1] und [PS94, Abschnitt 3.1] ist folgende Transformation beschrieben:

$$(\lambda x \texttt{ -> } e)\ arg \overset{(\beta\text{-atom})}{\texttt{===>}} e[arg/x] \qquad \text{wenn } arg \text{ atomar ist.}$$

Lemma 4.6. *(β-atom) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.
1.Fall : arg ist eine Variable v

$$[\![(\lambda x \texttt{ -> } e)\ v]\!] \equiv ([\![(\lambda x \texttt{ -> } e)]\!]\ [\![v]\!]) \equiv ((\lambda x.\ [\![e]\!])\ v) \xrightarrow{betavar} [\![e]\!][v/x] \equiv [\![e[v/x]]\!]$$

2.Fall : arg ist ein Literal, d.h. $[\![arg]\!] = c_i$

$$\begin{aligned}
&[\![(\lambda x \texttt{ -> } e)\ arg]\!] \equiv ([\![(\lambda x \texttt{ -> } e)]\!]\ [\![arg]\!]) \equiv ((\lambda x.\ [\![e]\!])\ c_i) \\
&\xrightarrow{lbeta} (\texttt{letrec } x = c_i \texttt{ in } [\![e]\!]) \\
&\xrightarrow{(cpcx-in)^*} (\texttt{letrec } x = c_i \texttt{ in } [\![e]\!][c_i/x]) \xrightarrow{gc-1} (\texttt{letrec } \{\} \texttt{ in } [\![e]\!][c_i/x]) \\
&\xrightarrow{gc-2} [\![e]\!][c_i/x] \equiv [\![e]\!][[\![arg]\!]/x] \equiv [\![e[arg/x]]\!]
\end{aligned}$$

Also gilt für beide Fälle: $[\![(\lambda x \texttt{ -> } e)\ arg]\!] \sim_c [\![e[arg/x]]\!]$. □

Im GHC wird eine weitere Variante der Beta-Reduktion durchgeführt, für den Fall, dass das Argument nicht atomar ist. Diese Variante wird auch in [San95, Seite 23] angesprochen.

$$(\lambda x \texttt{ -> } e)\ arg \overset{(\beta)}{\texttt{===>}} \texttt{let } x = arg \texttt{ in } e$$

Lemma 4.7. *(β) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.

$$\begin{aligned}
&[\![(\lambda x \texttt{ -> } e)\ arg]\!] \equiv ([\![(\lambda x \texttt{ -> } e)]\!]\ [\![arg]\!]) \equiv ((\lambda x.\ [\![e]\!])\ [\![arg]\!]) \\
&\xrightarrow{lbeta} (\texttt{letrec } x = [\![arg]\!] \texttt{ in } [\![e]\!]) \equiv [\![\texttt{let } x = arg \texttt{ in } e]\!]
\end{aligned}$$

□

4.3.2 Transformationen für `let`-Ausdrücke

4.3.2.1 Die „floating let out of let"-Transformationen

Für die `let`-über-`let`-Verschiebung[2] werden zwei Varianten durchgeführt, je nachdem, ob das `let` rekursiv ist oder nicht.

[2]Vgl. [San95, Abschnitt 3.4.2] und [PS94, Abschnitt 3.3].

Variante mit `let`:

$$\texttt{let } x = (\texttt{let(rec) } \mathit{Bind} \texttt{ in } B_1) \texttt{ in } B_2 \quad \overset{\text{(flool-let)}}{\texttt{===>}} \quad \texttt{let(rec) } \mathit{Bind} \texttt{ in } (\texttt{let } x = B_1 \texttt{ in } B_2)$$

Lemma 4.8. *(flool-let) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.

$$\begin{aligned}
& [\![\texttt{let } x = (\texttt{let(rec) } \mathit{Bind} \texttt{ in } B_1) \texttt{ in } B_2]\!] \\
\equiv\ & (\texttt{letrec } x = [\![(\texttt{let(rec) } \mathit{Bind} \texttt{ in } B_1)]\!] \texttt{ in } [\![B_2]\!]) \\
\equiv\ & (\texttt{letrec } x = (\texttt{letrec } [\![\mathit{Bind}]\!] \texttt{ in } [\![B_1]\!]) \texttt{ in } [\![B_2]\!]) \\
\xrightarrow{llet-e}\ & (\texttt{letrec } x = [\![B_1]\!]; [\![\mathit{Bind}]\!] \texttt{ in } [\![B_2]\!]) \\
\xleftarrow{llet-in}\ & (\texttt{letrec } [\![\mathit{Bind}]\!] \texttt{ in } (\texttt{letrec } x = [\![B_1]\!] \texttt{ in } [\![B_2]\!])) \\
\equiv\ & (\texttt{let(rec) } [\![\mathit{Bind}]\!] \texttt{ in } [\![(\texttt{let } x = B_1 \texttt{ in } B_2)]\!]) \\
\equiv\ & [\![\texttt{let(rec) } \mathit{Bind} \texttt{ in } (\texttt{let } x = B_1 \texttt{ in } B_2)]\!]
\end{aligned}$$

□

Variante mit `letrec`:

$$\texttt{letrec } x = (\texttt{let(rec) } \mathit{Bind} \texttt{ in } B_1) \texttt{ in } B_2 \quad \overset{\text{(flool-letrec)}}{\texttt{===>}} \quad \texttt{letrec } \mathit{Bind}; x = B_1 \texttt{ in } B_2$$

Lemma 4.9. *(flool-letrec) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.

$$\begin{aligned}
& [\![\texttt{letrec } x = (\texttt{let(rec) } \mathit{Bind} \texttt{ in } B_1) \texttt{ in } B_2]\!] \\
\equiv\ & (\texttt{letrec } x = [\![(\texttt{let(rec) } \mathit{Bind} \texttt{ in } B_1)]\!] \texttt{ in } [\![B_2]\!]) \\
\equiv\ & (\texttt{letrec } x = (\texttt{letrec } [\![\mathit{Bind}]\!] \texttt{ in } [\![B_1]\!]) \texttt{ in } [\![B_2]\!]) \\
\xrightarrow{llet-e}\ & (\texttt{letrec } x = [\![B_1]\!]; [\![\mathit{Bind}]\!] \texttt{ in } [\![B_2]\!]) \\
=\ & (\texttt{letrec } [\![\mathit{Bind}]\!]; x = [\![B_1]\!] \texttt{ in } [\![B_2]\!]) \\
\equiv\ & (\texttt{letrec } [\![\mathit{Bind}; x = B_1]\!] \texttt{ in } [\![B_2]\!]) \\
\equiv\ & [\![\texttt{letrec } \mathit{Bind}; x = B_1 \texttt{ in } B_2]\!]
\end{aligned}$$

Somit wurde gezeigt:

$$[\![\texttt{letrec } x = (\texttt{let(rec) } \mathit{Bind} \texttt{ in } B_1) \texttt{ in } B_2]\!] \sim_c [\![\texttt{letrec } \mathit{Bind}; x = B_1 \texttt{ in } B_2]\!]$$

□

4.3.2.2 Die „floating lets out of a case scrutinee“-Transformation

Die in [San95, Abschnitt 3.4.3] und [PS94, Abschnitt 3.10] beschriebene Transformation entspricht der (lcase)-Reduktion des FUNDIO-Kalküls:

$$\texttt{case}\ (\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ E)\ \texttt{of}\ \mathit{Alts} \overset{\text{(flooacs)}}{\texttt{===>}} \begin{array}{l}\texttt{let(rec)}\ \mathit{Bind}\\ \texttt{in case}\ E\ \texttt{of}\ \mathit{Alts}\end{array}$$

Lemma 4.10. *(flooacs) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Wir unterscheiden zwei Fälle:
1.Fall: *Alts* enthält keine default-Alternative. Seien Alts' die fehlenden Alternativen, die entsprechend der Regeln für die Übersetzung $[\![\cdot]\!]$ ermittelt werden.

$$\begin{aligned}
&[\![\texttt{case}\ (\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ E)\ \texttt{of}\ \mathit{Alts}]\!]\\
\equiv\ &(\texttt{case}\ [\![(\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ E)]\!]\ [\![\mathit{Alts}]\!]\ [\![\mathit{Alts}']\!])\\
\equiv\ &(\texttt{case}\ (\texttt{letrec}\ [\![\mathit{Bind}]\!]\ \texttt{in}\ [\![E]\!])\ [\![\mathit{Alts}]\!]\ [\![\mathit{Alts}']\!])\\
\xrightarrow{lcase}\ &(\texttt{letrec}\ [\![\mathit{Bind}]\!]\ \texttt{in}\ (\texttt{case}\ [\![E]\!]\ [\![\mathit{Alts}]\!]\ [\![\mathit{Alts}']\!]))\\
\equiv\ &(\texttt{letrec}\ [\![\mathit{Bind}]\!]\ \texttt{in}\ [\![\texttt{case}\ E\ \texttt{of}\ \mathit{Alts}]\!])\\
\equiv\ &[\![\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in case}\ E\ \texttt{of}\ \mathit{Alts}]\!]
\end{aligned}$$

2. Fall: *Alts* enthält eine default-Alternative. Seien Alts' die durch die Übersetzung entstandenen Alternativen.

$$\begin{aligned}
&[\![\texttt{case}\ (\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ E)\ \texttt{of}\ \mathit{Alts}; x\ \texttt{->}\ s]\!]\\
\equiv\ &(\texttt{letrec}\ y = [\![(\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ E)]\!]\ \texttt{in}\ (\texttt{case}\ y\ [\![\mathit{Alts}]\!]\ \mathit{Alts}'))\\
\equiv\ &(\texttt{letrec}\ y = (\texttt{letrec}\ [\![\mathit{Bind}]\!]\ \texttt{in}\ [\![E]\!])\ \texttt{in}\ (\texttt{case}\ y\ [\![\mathit{Alts}]\!]\ \mathit{Alts}'))\\
\xrightarrow{llet-e}\ &(\texttt{letrec}\ y = [\![E]\!], [\![\mathit{Bind}]\!]\ \texttt{in}\ (\texttt{case}\ y\ [\![\mathit{Alts}]\!]\ \mathit{Alts}'))\\
\xleftarrow{llet-in}\ &(\texttt{letrec}\ [\![\mathit{Bind}]\!]\ \texttt{in}\ (\texttt{letrec}\ y = [\![E]\!]\ \texttt{in}\ (\texttt{case}\ y\ [\![\mathit{Alts}]\!]\ \mathit{Alts}')))\\
\equiv\ &(\texttt{letrec}\ [\![\mathit{Bind}]\!]\ \texttt{in}\ [\![\texttt{case}\ E\ \texttt{of}\ x\ \texttt{->}\ s]\!])\\
\equiv\ &[\![\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in case}\ E\ \texttt{of}\ x\ \texttt{->}\ s]\!]
\end{aligned}$$

Somit wurde gezeigt:

$$[\![\texttt{case}\ (\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ E)\ \texttt{of}\ \mathit{Alts}]\!] \sim_c [\![\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in case}\ E\ \texttt{of}\ \mathit{Alts}]\!]$$

□

4.3.2.3 Die „dead code removal“-Transformationen

Mithilfe der „dead code removal“-Transformation[3] werden unbenutzte Bindungen entfernt:

[3] Vgl. [San95, Abschnitt 3.2.1] und [PS94, Abschnitt 3.18].

$$\texttt{let}\ x = E\ \texttt{in}\ B \overset{\text{(dcr)}}{\texttt{===>}} B$$

wenn x kein freies Vorkommen in B hat.

Lemma 4.11. *(dcr-let) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.

$$[\![\texttt{let}\ x = E\ \texttt{in}\ B]\!] \equiv (\texttt{letrec}\ x = [\![E]\!]\ \texttt{in}\ [\![B]\!]) \xrightarrow{gc-1} (\texttt{letrec}\ \{\}\ \texttt{in}\ [\![B]\!]) \xrightarrow{gc-2} [\![B]\!]$$

□

In [San95, Seite 25] wird eine weitere Variante der „dead code removal"-Transformation beschrieben, mithilfe derer mehrere rekursive Bindungen entfernt werden.

$$\texttt{rec}\ \textit{bindings}\ \texttt{in}\ B \overset{\text{(dcr-letrec)}}{\texttt{===>}} B$$

wenn keine der Bindungen aus *bindings* in B benutzt wird

Lemma 4.12. *(dcr-letrec) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. O.B.d.A. sei *bindings* die Gruppe von Bindungen $x_1 = E_1, \ldots, x_n = E_n$.

$$\begin{aligned} & [\![\texttt{rec}\ \textit{bindings}\ \texttt{in}\ B]\!] \\ \equiv\ & (\texttt{letrec}\ x_1 = [\![E_1]\!], \ldots, x_n = [\![E_n]\!]\ \texttt{in}\ [\![B]\!]) \\ \xrightarrow{(gc-1)^*}\ & (\texttt{letrec}\ \{\}\ \texttt{in}\ [\![B]\!]) \\ \xrightarrow{gc-2}\ & [\![B]\!] \end{aligned}$$

□

4.3.2.4 Die „inlining"-Transformation

„Inlining"[4] wird benutzt, um Ausdrücke zu kopieren.

$$\texttt{let(rec)}\ x = e\ \texttt{in}\ C[x] \overset{\text{(inl)}}{\texttt{===>}} \texttt{let(rec)}\ x = e\ \texttt{in}\ C[e]$$

Lemma 4.13. *(inl) ist keine $[\![\cdot]\!]$-korrekte Programmtransformation.*

[4] Vgl. [San95, Abschnitt 3.2.2], [PS94, Abschnitt 3.17]. In [PM02] und [San95, Kapitel 6] werden die Vorteile und die Implementierung der Transformation ausführlich diskutiert.

Beweis. Gegenbeispiel: Betrachte folgenden Ausdruck s in der Kernsprache des GHC:

$$s = \texttt{let } x = (\texttt{unsafePerformIO getChar}) \texttt{ in case } x \texttt{ of 'd' -> (case } x \texttt{ of 'd' -> 'd')}$$

Wir erhalten den folgenden Ausdruck t, wenn wir die Transformation (inl) einmal anwenden:

$$\begin{aligned} t = \texttt{let } & x = (\texttt{unsafePerformIO getChar}) \texttt{ in} \\ & \texttt{case } (\texttt{unsafePerformIO getChar}) \texttt{ of 'd' -> (case } x \texttt{ of 'd' -> 'd')} \end{aligned}$$

Man betrachte nun die übersetzten Ausdrücke:

$$\begin{aligned} [\![s]\!] = (\texttt{letrec } & x = (\texttt{case (IO } \mathcal{B}) \ (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \texttt{ in} \\ & (\texttt{case } x \ \ldots (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \end{aligned}$$

$$\begin{aligned} [\![t]\!] = (\texttt{letrec } & x = (\texttt{case (IO } \mathcal{B}) \ (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \texttt{ in} \\ & (\texttt{case (case (IO } \mathcal{B}) \ (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \\ & \quad \ldots \ (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \end{aligned}$$

Sei $P = \{(\mathcal{B}, \texttt{'d'})\}$, dann gilt: $[\![s]\!]\Downarrow(P)$, aber $[\![t]\!]\Uparrow(P)$:

$$\begin{aligned} [\![s]\!] = (\texttt{letrec } & x = (\texttt{case (IO } \mathcal{B}) \ (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \texttt{ in} \\ & (\texttt{case } x \ \ldots (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \\ \xrightarrow{n,IOr,(\mathcal{B},\texttt{'d'})} (\texttt{letrec } & x = (\texttt{case 'd' } (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \texttt{ in} \\ & (\texttt{case } x \ \ldots (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \\ \xrightarrow{n,case} (\texttt{letrec } & x = (\texttt{letrec } \{\} \texttt{ in 'd'}) \texttt{ in} \\ & (\texttt{case } x \ \ldots (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \\ \xrightarrow{n,llet} (\texttt{letrec } & x = \texttt{'d' in (case } x \ \ldots (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \\ \xrightarrow{n,case} (\texttt{letrec } & x = \texttt{'d' in (letrec } \{\} \texttt{ in (case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots))) \\ \xrightarrow{n,llet} (\texttt{letrec } & x = \texttt{'d' in (case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \\ \xrightarrow{n,case} (\texttt{letrec } & x = \texttt{'d' in (letrec } \{\} \texttt{in 'd'})) \\ \xrightarrow{n,llet} (\texttt{letrec } & x = \texttt{'d' in 'd'}) \end{aligned}$$

$$\begin{aligned} [\![t]\!] = (\texttt{letrec } & x = (\texttt{case (IO } \mathcal{B}) \ (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \texttt{ in} \\ & (\texttt{case (case (IO } \mathcal{B}) \ (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \\ & \quad \ldots \ (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \\ \xrightarrow{n,IOr,(\mathcal{B},\texttt{'d'})} (\texttt{letrec } & x = (\texttt{case (IO } \mathcal{B}) \ (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \texttt{ in} \\ & (\texttt{case (case 'd' } (p_1 \to p_1) \ldots (p_n \to p_n) \ (p_{n+1} \to \bot) \ldots) \\ & \quad \ldots \ (\texttt{'d'} \to \texttt{case } x \ \ldots \ (\texttt{'d'} \to \texttt{'d'}) \ \ldots) \ \ldots)) \end{aligned}$$

$$\xrightarrow{n,case} (\texttt{letrec}\ x = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots)\ \texttt{in}$$
$$(\texttt{case}\ (\texttt{letrec}\ \{\}\ \texttt{in'd'})\ \ldots\ (\texttt{'d'} \to \texttt{case}\ x\ \ldots\ (\texttt{'d'} \to \texttt{'d'})\ \ldots)\ \ldots))$$
$$\xrightarrow{n,lcase} (\texttt{letrec}\ x = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots)\ \texttt{in}$$
$$(\texttt{letrec}\ \{\}\ \texttt{in}\ (\texttt{case 'd'}\ \ldots\ (\texttt{'d'} \to \texttt{case}\ x\ \ldots\ (\texttt{'d'} \to \texttt{'d'})\ \ldots)\ \ldots)))$$
$$\xrightarrow{n,llet} (\texttt{letrec}\ x = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots)\ \texttt{in}$$
$$(\texttt{case 'd'}\ \ldots\ (\texttt{'d'} \to \texttt{case}\ x\ \ldots\ (\texttt{'d'} \to \texttt{'d'})\ \ldots)\ \ldots))$$
$$\xrightarrow{n,case} (\texttt{letrec}\ x = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots)\ \texttt{in}$$
$$(\texttt{letrec}\ \{\}\ \texttt{in}\ (\texttt{case}\ x\ \ldots\ (\texttt{'d'} \to \texttt{'d'})\ \ldots)\ \ldots)))$$
$$\xrightarrow{n,llet} (\texttt{letrec}\ x = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots)\ \texttt{in}$$
$$(\texttt{case}\ x\ \ldots\ (\texttt{'d'} \to \texttt{'d'})\ \ldots)\ \ldots))$$
$$\xrightarrow{n,IOr,?} \text{Ein zweites IO-Paar wird benötigt!}$$

Somit folgt $[\![s]\!] \not\sim_c [\![t]\!]$. □

Inlining ist jedoch in speziellen Situationen zulässig, die wir nun besprechen, wobei die folgenden Transformationen durch Untersuchung der Situationen, in denen im GHC Inlining angewendet wird, entstanden sind. Auf Details im GHC gehen wir in Kapitel 5 ein.

Sei die Transformation (uinl) wie folgt definiert, die analog zur (ucp-in)-Regel des FUNDIO-Kalküls ist:

$\texttt{let(rec)}\ x = e\ \texttt{in}\ C[x] \overset{\text{(uinl)}}{\texttt{===>}} C[e]$
wenn x genau einmal frei in $C[x]$ nicht im Rumpf einer Abstraktion vorkommt und x nicht in e vorkommt

Lemma 4.14. *(uinl) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation*

Beweis. Die Bedingung an $C[x]$ bedeutet für die Übersetzung nach L_{FUNDIO}, dass $[\![C]\!]$ ein Oberflächenkontext ist.

$$\begin{aligned} & [\![\texttt{let(rec) x=e in C[x]}]\!] \\ \equiv\ & (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][x]) \xrightarrow{ucp} (\texttt{letrec}\ \{\}\ \texttt{in}\ [\![C]\!][e]) \xrightarrow{gc} [\![C]\!][e] \\ \equiv\ & [\![C[e]]\!] \end{aligned}$$

□

Ebenso kann man auch die $[\![\cdot]\!]$-Korrektheit der zur $(ucp{-}e)$-Regel analogen Transformation zeigen, d.h. auch das Kopieren in eine Bindung ist erlaubt, wenn die Variable x nur einmal und nicht im Rumpf einer Abstraktion vorkommt.

Sei die Transformation (bruinl) analog zur (ucpb-in)-Regel des FUNDIO-Kalküls definiert:

$$
\begin{array}{l}
\texttt{let(rec)}\ x = e\ \texttt{in} \\
\quad C[\texttt{case}\ e_1\ \texttt{of} \\
\qquad P_1\ \texttt{->}\ B_1 \\
\qquad \ldots \\
\qquad P_i\ \texttt{->}\ C'[x] \\
\qquad \ldots \\
\qquad P_n\ \texttt{->}\ B_n]
\end{array}
\quad \overset{\text{(bruinl)}}{\texttt{===>}} \quad
\begin{array}{l}
\texttt{let(rec)}\ x = e\ \texttt{in} \\
\quad C[\texttt{case}\ e_1\ \texttt{of} \\
\qquad P_1\ \texttt{->}\ B_1 \\
\qquad \ldots \\
\qquad P_i\ \texttt{->}\ C'[e] \\
\qquad \ldots \\
\qquad P_n\ \texttt{->}\ B_n]
\end{array}
$$

wenn x nur in $B_1, \ldots, B_n$ und genau einmal in $C'[x]$ frei vorkommt, wobei das Vorkommen in $C[C'[x]]$ nicht im Rumpf einer Abstraktion ist

Lemma 4.15. *(bruinl) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Wir zeigen hier den Fall, indem die Alternativen keine default-Alternative enthalten, die restlichen Fälle sind im Anhang in Abschnitt A.5 angegeben.

Die Bedingungen an das Vorkommen von x bedeuten für die Übersetzung, dass $[\![C]\!]$ und $[\![C']\!]$ Oberflächenkontexte sind.

$$
\begin{array}{rl}
& [\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[\texttt{case}\ e_1\ \texttt{of}\ \{P_1\ \texttt{->}\ B_1; \ldots; P_i\ \texttt{->}\ C'[x] \ldots; P_n\ \texttt{->}\ B_n\}]\!] \\
\equiv & (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][(\texttt{case}\ [\![e_1]\!] \\
& \qquad ([\![P_1]\!] \to [\![B_1]\!]) \ldots ([\![P_i]\!] \to [\![C']\!][x]) \ldots ([\![P_n]\!] \to [\![B_n]\!]) \\
& \qquad (pat_{n+1} \to \bot) \ldots (pat_N \to \bot))]) \\
\xrightarrow{ucpb} & (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][(\texttt{case}\ [\![e_1]\!] \\
& \qquad ([\![P_1]\!] \to [\![B_1]\!]) \ldots ([\![P_i]\!] \to [\![C']\!][e]) \ldots ([\![P_n]\!] \to [\![B_n]\!]) \\
& \qquad (pat_{n+1} \to \bot) \ldots (pat_N \to \bot))]) \\
\equiv & [\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[\texttt{case}\ e_1\ \texttt{of}\ \{P_1\ \texttt{->}\ B_1; \ldots; P_i\ \texttt{->}\ C'[e]; \ldots; P_n\ \texttt{->}\ B_n\}]\!]
\end{array}
$$

□

Analog zur (ucpb-e) Regel ist auch das Kopieren erlaubt, wenn $C[\ldots]$ eine rechte Seite einer anderen Bindung ist und x nur in den B_j und einmal in $C'[x]$ frei vorkommt, wobei das Vorkommen in $C'[C[x]]$ nicht im Rumpf einer Abstraktion ist.

Im Folgenden definieren wir eine Transformation, die es erlaubt bestimmte Ausdrücke beliebig zu kopieren, wofür wir zunächst die Menge $CHEAP$ von besonderen Ausdrücken definieren.

Definition 4.16. *Sei $CHEAP$ die wie folgt definierte Menge von Ausdrücken aus*

$L_{GHCCore}$:

$x \in CHEAP$ *gdw.*

- x *ist ein Literal,*
- x *ist eine Variable,*
- x *ist eine Abstraktion,*
- x *ist ein primitiver Operator mit Stelligkeit* > 0 *oder*
- x *ist eine Konstruktoranwendung* $c_i\ a_1 \ldots a_n$, $n \leq ar(c_i)$ *und* $a_j \in CHEAP$ *für* $j = 1, \ldots, n$

Die Transformation (cheapinl) kann nun wie folgt formuliert werden:

$$\texttt{let(rec)}\ x = e\ \texttt{in}\ C[x] \overset{\text{(cheapinl)}}{\texttt{===>}} \texttt{let(rec)}\ x = e\ \texttt{in}\ C[e]$$

wenn e ein Element der Menge $CHEAP$ ist.

Lemma 4.17. *(cheapinl) ist eine* $[\![\cdot]\!]$*-korrekte Programmtransformation.*

Beweis. Da aus $e \in CHEAP$ folgt, dass $[\![e]\!] \in L_{cheap}$, dürfen wir die (cpcheap)-Reduktion verwenden:

$$\begin{aligned} [\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[x]]\!] &\equiv (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][x]) \\ \xrightarrow{cpcheap} (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][e]) &\equiv [\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[e]]\!] \end{aligned}$$

□

Analog zur (cpcheap-e)-Regel ist es auch möglich Ausdrücke in andere Bindungen zu kopieren, wenn die Ausdrücke aus $CHEAP$ sind.

4.3.3 Transformationen für `case`-Ausdrücke

4.3.3.1 Die „case of known constructor"-Transformationen

Die in [San95, Abschnitt 3.3.1] und [PS94, Abschnitt 3.7] beschriebene „case of known constructor" Transformation beachtet kein Sharing, da die dort verwendete Kernsprache nur atomare Argumente erlaubt.

Die aktuelle Implementierung[5] verwendet aufgrund dessen folgende Transformation, die Sharing beachtet:

[5] Im Modul `ghc/compiler/simplCore/Simplify.lhs` befindet sich die zugehörige Funktion `knownCon`.

$$\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of}\ \ldots\ c\ b_1 \ldots b_n\ \texttt{->}\ e\ \ldots \quad \overset{\text{(cokc)}}{\texttt{===>}} \quad \texttt{letrec}\ b_1 = a_1; \ldots; b_n = a_n\ \texttt{in}\ e$$

Lemma 4.18. *(cokc) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Seien $Alts'$ die durch die Übersetzung des `case`-Ausdrucks hinzugefügten Alternativen.

1. Fall: In den `case`-Alternativen existiert keine default-Alternative:

$$\begin{aligned}
&[\![\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of}\ \ldots (c\ b_1 \ldots b_n\ \texttt{->}\ e) \ldots]\!] \\
\equiv\ &(\texttt{case}\ [\![c\ a_1 \ldots a_n]\!]\ \ldots ([\![c\ b_1 \ldots b_n]\!]\ \texttt{->}\ [\![e]\!]) \ldots Alts') \\
\equiv\ &(\texttt{case}\ (c\ [\![a_1]\!] \ldots [\![a_n]\!]) \ldots ((c\ [\![b_1]\!] \ldots [\![b_n]\!])\ \texttt{->}\ [\![e]\!]) \ldots Alts') \\
\equiv\ &(\texttt{case}\ (c\ [\![a_1]\!] \ldots [\![a_n]\!]) \ldots ((c\ b_1 \ldots b_n)\ \texttt{->}\ [\![e]\!]) \ldots Alts') \\
\xrightarrow{case-c}\ &(\texttt{letrec}\ b_1 = [\![a_1]\!], \ldots, b_n = [\![a_n]\!]\ \texttt{in}\ [\![e]\!]) \\
\equiv\ &[\![\texttt{letrec}\ b_1 = a_1; \ldots; b_n = a_n\ \texttt{in}\ e]\!]
\end{aligned}$$

2. Fall: Die `case`-Alternativen enthalten auch eine default-Alternative: Seien wiederum $Alts'$ die durch die Übersetzung hinzugefügten Alternativen.

$$\begin{aligned}
&[\![\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of}\ \ldots; (c\ b_1 \ldots b_n\ \texttt{->}\ e); \ldots; x\ \texttt{->}\ s]\!] \\
\equiv\ &(\texttt{letrec}\ y = [\![(c\ a_1 \ldots a_n)]\!]\ \texttt{in} \\
&\ (\texttt{case}\ y\ [\![\ldots]\!]\ ([\![(c\ b_1 \ldots b_n)]\!]\ \texttt{->}\ [\![e]\!])\ \ [\![\ldots]\!]\ Alts')) \\
\equiv\ &(\texttt{letrec}\ y = (c\ [\![a_1]\!] \ldots [\![a_n]\!])\ \texttt{in}\ (\texttt{case}\ y\ [\![\ldots]\!]\ ((c\ b_1 \ldots b_n)\ \texttt{->}\ [\![e]\!])\ [\![\ldots]\!]\ Alts')) \\
\xrightarrow{case-in}\ &(\texttt{letrec}\ y = (c\ y_1\ \ldots\ y_n); y_1 = [\![a_1]\!]; \ldots; y_n = [\![a_n]\!] \\
&\ \texttt{in}\ (\texttt{letrec}\ b_1 = y_1; \ldots b_n = y_n\ \texttt{in}\ [\![e]\!])) \\
\xrightarrow{llet-in}\ &(\texttt{letrec}\ y = (c\ y_1\ \ldots\ y_n); y_1 = [\![a_1]\!]; \ldots; y_n = [\![a_n]\!]; b_1 = y_1; \ldots b_n = y_n\ \texttt{in}\ [\![e]\!]) \\
\xrightarrow{(xch)^*}\ &(\texttt{letrec}\ y = (c\ y_1\ \ldots\ y_n); b_1 = [\![a_1]\!]; \ldots; b_n = [\![a_n]\!]; y_1 = b_1; \ldots y_n = b_n\ \texttt{in}\ [\![e]\!]) \\
\xrightarrow{(cpx)^*}\ &(\texttt{letrec}\ y = (c\ b_1\ \ldots\ b_n); b_1 = [\![a_1]\!]; \ldots; b_n = [\![a_n]\!]; y_1 = b_1; \ldots y_n = b_n\ \texttt{in}\ [\![e]\!]) \\
\xrightarrow{(gc)^*}\ &(\texttt{letrec}\ b_1 = [\![a_1]\!]; \ldots; b_n = [\![a_n]\!]\ \texttt{in}\ [\![e]\!]) \\
\equiv\ &[\![\texttt{letrec}\ b_1 = a_1; \ldots; b_n = a_n\ \texttt{in}\ e]\!]
\end{aligned}$$

□

Es gibt analoge Varianten der Transformation, in der das erste Argument des `case`-Ausdruck eine Variable ist, die an eine Konstruktoranwendung gebunden ist, wobei die Argumente atomar sind (siehe [San95, Seite 31]). Hierbei sind zwei Fälle zu unterscheiden:

- Die Variable ist durch einen `let(rec)`-Ausdruck gebunden.

$$\texttt{let(rec)}\ x = c\ a_1 \ldots a_n\ \texttt{in}\ \texttt{case}\ x\ \texttt{of}\ c\ b_1 \ldots b_n\ \texttt{->}\ e\ \ldots \quad \overset{\text{(cokc-l)}}{\texttt{===>}} \quad \texttt{let(rec)}\ x = c\ a_1 \ldots a_n\ \texttt{in letrec}\ b_1 = a_1, \ldots, b_n = a_n\ \texttt{in}\ e$$

Die $[\![\cdot]\!]$-Korrektheit der Transformation (cokc-l) kann einfach gezeigt werden, denn die Konstruktoranwendung darf im Fundio-Kalkül aufgrund der (cpcheap)-Regel für die Variable ersetzt werden und danach kann der Beweis der (cokc)-Transformation verwendet werden.

- Die Variable ist durch einen übergeordneten `case`-Ausdruck gebunden:

$$\texttt{case}\ x\ \texttt{of}\ c\ x_1 \ldots x_n\ \texttt{->}\ \texttt{case}\ x\ \texttt{of}\ c\ y_1 \ldots y_n\ \texttt{->}\ e\ \ldots\ \ldots \quad \overset{\text{(cokc-c)}}{\texttt{===>}} \quad \texttt{case}\ x\ \texttt{of}\ c\ x_1 \ldots x_n\ \texttt{->}\ e[x_i/y_i]_{i=1}^n\ \ldots$$

Die Regel (cokc-c) ist $[\![\cdot]\!]$-korrekt, denn mit der (ccpcx)-Transformation kann die Konstruktoranwendung $c\ x_1 \ldots x_n$ in die Alternative kopiert werden und danach der Beweis der (cokc)-Transformation für den inneren `case`-Ausdruck angewendet werden. Der dadurch entstehende `letrec`-Ausdruck kann durch mehrere (cpcheap)- und eine anschließende (dcr-letrec)-Transformation entfernt werden.

Des Weiteren gibt es eine Variante der „case of known constructor“-Transformation, für den Fall, dass kein Pattern einer Alternative matcht, aber eine default-Alternative vorhanden ist:

$$\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of}\ \ldots\ y\ \texttt{->}\ E \quad \overset{\text{(cokc-default)}}{\texttt{===>}} \quad \texttt{let}\ y = (c\ a_1 \ldots a_n)\ \texttt{in}\ E$$

wenn keine Alternative passt.

Lemma 4.19. *(cokc-default) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. 1. Fall: Es existieren weitere Alternativen außer der default-Alternative.

Seien z, b_1, …, b_n, y_1, …, y_n neue Variablen und $pat_{n+1}, \ldots pat_m$ die Pattern des zu c zugehörigen Typs für die keine Alternative angegeben ist. Die Pattern

$pat_{m+1}, \ldots, pat_N$ seien so gewählt, dass sie die übrigen Konstruktoren aus $\mathcal{C}$ abdecken.

$$
\begin{aligned}
& [\![\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of}\ Alts; y\ \texttt{->}\ E]\!] \\
\equiv\ & (\texttt{letrec}\ z = [\![c\ a_1 \ldots a_n]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ z \\
& \qquad [\![Alts]\!] \\
& \qquad ([\![pat_{n+1}]\!]\ \texttt{->}\ [\![E]\!][z/y])\ \ldots\ ([\![pat_{n+i}]\!]\ \texttt{->}\ [\![E]\!][z/y])\ \ldots\ ([\![pat_m]\!]\ \texttt{->}\ [\![E]\!][z/y]) \\
& \qquad (pat_{m+1}\ \texttt{->}\ \bot)\ \ldots\ (pat_N\ \texttt{->}\ \bot))) \\
\equiv\ & (\texttt{letrec}\ z = (c\ [\![a_1]\!] \ldots [\![a_n]\!])\ \texttt{in} \\
& \quad (\texttt{case}\ z \\
& \qquad [\![Alts]\!] \\
& \qquad ([\![pat_{n+1}]\!]\ \texttt{->}\ [\![E]\!][z/y])\ \ldots\ ((c\ b_1\ \ldots\ b_n)\ \texttt{->}\ [\![E]\!][z/y])\ \ldots\ ([\![pat_m]\!]\ \texttt{->}\ [\![E]\!][z/y]) \\
& \qquad (pat_{m+1}\ \texttt{->}\ \bot)\ \ldots\ (pat_N\ \texttt{->}\ \bot))) \\
\xrightarrow{case-in}\ & (\texttt{letrec}\ z = (c\ y_1\ \ldots y_n), y_1 = [\![a_1]\!], \ldots, y_n = [\![a_n]\!])\ \texttt{in} \\
& \quad (\texttt{letrec}\ b_1 = y_1, \ldots, b_n = y_n\ \texttt{in}\ [\![E]\!][z/y])) \\
\xrightarrow{(gc)^*}\ & (\texttt{letrec}\ z = (c\ y_1\ \ldots y_n), y_1 = [\![a_1]\!], \ldots, y_n = [\![a_n]\!])\ \texttt{in}\ [\![E]\!][z/y]) \qquad \text{(da die } b_i \text{ neu)} \\
\xrightarrow{(ucp)^*}\ & (\texttt{letrec}\ z = (c\ [\![a_1]\!] \ldots [\![a_n]\!])\ \texttt{in}\ [\![E]\!][z/y]) \qquad \text{(da die } y_i \text{ neu)} \\
=\ & (\texttt{letrec}\ y = (c\ [\![a_1]\!] \ldots [\![a_n]\!])\ \texttt{in}\ [\![E]\!]) \\
\equiv\ & [\![\texttt{let}\ y = c\ a_1 \ldots a_n\ \texttt{in}\ E]\!]
\end{aligned}
$$

2. Fall: Es existiert nur die default-Alternative:

Seien $x, y_1 \ldots y_n, z_1, \ldots, z_n$ neue Variablen.

$$
\begin{aligned}
& [\![\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of}\ y\ \texttt{->}\ E]\!] \\
\equiv\ & (\texttt{letrec}\ x = [\![(c\ a_1 \ldots a_n)]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ y\ (pat_1\ \texttt{->}\ [\![E[x/y]]\!])\ \ldots\ (pat_i\ \texttt{->}\ [\![E[x/y]]\!])\ \ldots\ (pat_N\ \texttt{->}\ [\![E[x/y]]\!]))) \\
\equiv\ & (\texttt{letrec}\ x = (c\ [\![a_1]\!] \ldots [\![a_n]\!])\ \texttt{in} \\
& \quad (\texttt{case}\ y\ (pat_1\ \texttt{->}\ [\![E[x/y]]\!])\ \ldots\ ((c\ b_1 \ldots b_n)\ \texttt{->}\ [\![E[x/y]]\!])\ \ldots\ (pat_N\ \texttt{->}\ [\![E[x/y]]\!]))) \\
\xrightarrow{case-in}\ & (\texttt{letrec}\ x = (c\ y_1\ \ldots y_n), y_1 = [\![a_1]\!], \ldots, y_n = [\![a_n]\!] \\
& \quad \texttt{in}\ (\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ [\![E[x/y]]\!])) \\
\xrightarrow{(gc)^*}\ & (\texttt{letrec}\ x = (c\ y_1\ \ldots y_n), y_1 = [\![a_1]\!], \ldots, y_n = [\![a_n]\!]\ \texttt{in}\ [\![E[x/y]]\!]) \qquad \text{(da die } z_i \text{ neu)} \\
\xrightarrow{(ucp)^*}\ & (\texttt{letrec}\ x = (c\ [\![a_1]\!]\ \ldots [\![a_n]\!])\ \texttt{in}\ [\![E[x/y]]\!]) \qquad \text{(da die } y_i \text{ neu)} \\
=\ & (\texttt{letrec}\ y = (c\ [\![a_1]\!]\ \ldots [\![a_n]\!])\ \texttt{in}\ [\![E]\!]) \\
\equiv\ & (\texttt{letrec}\ y = [\![c\ a_1 \ldots a_n]\!]\ \texttt{in}\ [\![E]\!]) \\
\equiv\ & [\![\texttt{let}\ y = c\ a_1 \ldots a_n\ \texttt{in}\ E]\!]
\end{aligned}
$$

□

4.3.3.2 Die „default binding elimination“-Transformation

Die „default binding elimination“[6] ist definiert als:

$$\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e \overset{\text{(dbe)}}{\texttt{===>}} \texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e[v_1/v_2]$$

wobei v_1 und v_2 Variablen sind.

Lemma 4.20. *(dbe) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.

$$\begin{aligned}
& [\![\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e]\!] \\
\equiv\ & (\texttt{letrec}\ x = [\![v_1]\!]\ \texttt{in}\ (\texttt{case}\ x\ (pat_1\ \texttt{->}[\![e[x/v_2]]\!]) \ldots (pat_N\ \texttt{->}[\![e[x/v_2]]\!]))) \\
\equiv\ & (\texttt{letrec}\ x = v_1\ \texttt{in}\ (\texttt{case}\ x\ (pat_1\ \texttt{->}[\![e[x/v_2]]\!]) \ldots (pat_N\ \texttt{->}[\![e[x/v_2]]\!]))) \\
\xrightarrow{(cpx-in)^*}\ & (\texttt{letrec}\ x = v_1\ \texttt{in}\ (\texttt{case}\ x\ (pat_1\ \texttt{->}[\![e[v_1/v_2]]\!]) \ldots (pat_N\ \texttt{->}[\![e[v_1/v_2]]\!]))) \\
=\ & (\texttt{letrec}\ x = [\![v_1]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ x\ (pat_1\texttt{->}\ [\![e[v_1/v_2][x/v_2]]\!]) \ldots (pat_N\texttt{->}\ [\![e[v_1/v_2][x/v_2]]\!]))) \\
\equiv\ & [\![\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e[v_1/v_2]]\!]
\end{aligned}$$

Somit wurde gezeigt:

$$[\![\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e]\!] \sim_c [\![\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e[v_1/v_2]]\!]$$

□

4.3.3.3 Die „dead alternative elimination“-Transformation

Diese Transformation[7] dient dazu, unerreichbare `case`-Alternativen zu entfernen:

$$\begin{array}{l}
\texttt{case}\ x\ \texttt{of} \\
\quad (c_1\ a_{1,1} \ldots a_{1,ar(c_1)})\ \texttt{->}\ E_1; \\
\quad \ldots; \\
\quad (c_k\ a_{k,1} \ldots a_{k,ar(c_k)})\ \texttt{->}\ E_k; \\
\quad \ldots; \\
\quad (c_n\ a_{n,1} \ldots a_{n,ar(c_n)})\ \texttt{->}\ E_n;
\end{array}
\overset{\text{(dae)}}{\texttt{===>}}
\begin{array}{l}
\texttt{case}\ x\ \texttt{of} \\
\quad (c_1\ a_{1,1} \ldots a_{1,ar(c_1)})\ \texttt{->}\ E_1; \\
\quad \ldots; \\
\quad (c_{k-1}\ a_{k-1,1} \ldots a_{k-1,ar(c_{k-1})})\ \texttt{->}\ E_{k-1}; \\
\quad (c_{k+1}\ a_{k+1,1} \ldots a_{k+1,ar(c_{k+1})})\ \texttt{->}\ E_{k+1}; \\
\quad \ldots; \\
\quad (c_n\ a_{n,1} \ldots a_{n,ar(c_n)})\ \texttt{->}\ E_n;
\end{array}$$

wenn x nicht vom Konstruktor c_k ist.

[6] Vgl. [PS94, Abschnitt 3.16] und [San95, Abschnitt 3.3.5].
[7] Vgl. [PS94, Abschnitt 3.8] und [San95, Abschnitt 3.3.6].

Lemma 4.21. *(dae) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Die Zusatzbedingung, dass x nicht vom Konstruktor c_k ist, benutzen wir in FUNDIO als Bedingung, dass $[\![x]\!]$ nicht zu einer Konstruktoranwendung $[\![c_k \ \ldots]\!]$ reduziert werden kann, oder an einen Ausdruck gebunden ist, der zu einer solchen Konstruktoranwendung reduziert.

$$
\begin{aligned}
&[\![\texttt{case}\ x\ \texttt{of}\ (c_1\ a_{1,1}\ldots a_{1,ar(c_1)})\ \texttt{->}\ E_1;\\
&\qquad\ldots;\\
&\qquad(c_k\ a_{k,1}\ldots a_{k,ar(c_k)})\ \texttt{->}\ E_k;\\
&\qquad\ldots;\\
&\qquad(c_n\ a_{n,1}\ldots a_{n,ar(c_n)})\ \texttt{->}\ E_n;]\!]\\
\equiv\ &(\texttt{case}\ [\![x]\!]\ (([\![c_1]\!]\ [\![a_{1,1}]\!]\ldots[\![a_{1,ar(c_1)}]\!])\to[\![E_1]\!])\\
&\qquad\ldots\\
&\qquad(([\![c_k]\!]\ [\![a_{k,1}]\!]\ldots[\![a_{k,ar(c_k)}]\!])\to[\![E_k]\!])\\
&\qquad\ldots\\
&\qquad(([\![c_n]\!]\ [\![a_{n,1}]\!]\ldots[\![a_{n,ar(c_n)}]\!])\to[\![E_n]\!])\\
&\qquad(p_{n+1}\to\bot)\ldots(p_N\to\bot))\\
\xrightarrow{crpl}\ &(\texttt{case}\ [\![x]\!]\ (([\![c_1]\!]\ [\![a_{1,1}]\!]\ldots[\![a_{1,ar(c_1)}]\!])\to[\![E_1]\!])\\
&\qquad\ldots\\
&\qquad(([\![c_k]\!]\ [\![a_{k,1}]\!]\ldots[\![a_{k,ar(c_k)}]\!])\to\bot)\\
&\qquad\ldots\\
&\qquad(([\![c_n]\!]\ [\![a_{n,1}]\!]\ldots[\![a_{n,ar(c_n)}]\!])\to[\![E_n]\!])\\
&\qquad(p_{n+1}\to\bot)\ldots(p_N\to\bot))\\
\equiv\ &[\![\texttt{case}\ x\ \texttt{of}\ (c_1\ a_{1,1}\ldots a_{1,ar(c_1)})\ \texttt{->}\ E_1;\\
&\qquad\ldots;\\
&\qquad(c_{k-1}\ a_{k-1,1}\ldots a_{k-1,ar(c_{k-1})})\ \texttt{->}\ E_{k-1};\\
&\qquad(c_{k+1}\ a_{k+1,1}\ldots a_{k+1,ar(c_{k+1})})\ \texttt{->}\ E_{k+1};\\
&\qquad\ldots;\\
&\qquad(c_n\ a_{n,1}\ldots a_{n,ar(c_n)})\ \texttt{->}\ E_n;]\!]
\end{aligned}
$$

□

4.3.3.4 Die „case of error"-Transformation

Die `error`-Funktion ist eine von Haskell zur Verfügung gestellte Funktion, die den semantischen Wert $\bot$ hat (siehe [ABB+99, Abschnitt 3.1]). Die „case of error"-Transformation wird in [PS94, Abschnitt 3.14] und [San95, Abschnitt 3.3.4] wie folgt definiert:

$$\texttt{case}\ (\texttt{error}\ E)\ \texttt{of}\ \mathit{Alts} \overset{\text{(coe)}}{\texttt{===>}} \texttt{error}\ E$$

Lemma 4.22. *(coe) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Da die `error`-Funktion den Wert $\bot$ hat, übersetzen wir diese Funktion als $\bot$. Wir unterscheiden zwei Fälle, je nachdem, ob die `case`-Alternativen eine default-Alternative enthält oder nicht. Wir benutzen in beiden Fällen die Gleichheit $\bot \sim_c R[\bot]$ aus Satz 3.33.

Betrachten wir zunächst den Fall, dass *Alts* keine default-Alternative enthält: Seien *Alts′* die zusätzlich durch die Übersetzung entstanden Alternativen des `case`-Ausdrucks.

$$[\![\texttt{case (error } E\texttt{) of } \mathit{Alts}]\!] \equiv (\texttt{case } \bot\ [\![\mathit{Alts}]\!]\ \mathit{Alts}') \sim_c \bot \equiv\ [\![\texttt{error } E]\!]$$

Betrachten wir nun den Fall, dass *Alts* eine default-Alternative enthält.

$$\begin{aligned}
&[\![\texttt{case (error } E\texttt{) of } \mathit{Alts}; x \texttt{ -> } E]\!] \\
&\equiv (\texttt{letrec } y = [\![(\texttt{error } E)]\!] \texttt{ in } (\texttt{case } y\ \ldots)) \\
&\equiv (\texttt{letrec } y = \bot \texttt{ in } (\texttt{case } y\ \ldots)) \\
&\sim_c \bot \\
&\equiv [\![\texttt{error } E]\!]
\end{aligned}$$

Insgesamt haben wir somit gezeigt:

$$[\![\texttt{case (error } E\texttt{) of } \mathit{Alts}]\!] \sim_c [\![\texttt{error } E]\!]$$

□

4.3.3.5 Die „floating case out of case"-Transformation

Diese Transformation[8] ist analog zur (ccase)-Regel für FUNDIO.

$$\texttt{case}\ \left(\begin{array}{l}\texttt{case } E \texttt{ of}\\ \quad P_1 \texttt{ -> } R_1\\ \quad \ldots;\\ \quad P_n \texttt{ -> } R_n\end{array}\right)\ \texttt{of}\ \begin{array}{l}Q_1 \texttt{ -> } S_1\\ \ldots\\ Q_m \texttt{ -> } S_m\end{array} \quad \overset{\text{(fcooc)}}{\texttt{===>}} \quad \begin{array}{l}\texttt{case } E \texttt{ of}\\ \quad P_1 \texttt{ -> case } R_1 \texttt{ of}\\ \qquad\qquad Q_1 \texttt{ -> } S_1\\ \qquad\qquad \ldots\\ \qquad\qquad Q_m \texttt{ -> } S_m\\ \quad \ldots\\ \quad P_n \texttt{ -> case } R_n \texttt{ of}\\ \qquad\qquad Q_1 \texttt{ -> } S_1\\ \qquad\qquad \ldots\\ \qquad\qquad Q_m \texttt{ -> } S_m\end{array}$$

Lemma 4.23. *(fcooc) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

[8] Vgl. [PS94, Abschnitt 3.11] und [San95, Abschnitt 3.5.2].

Beweis. Wir betrachten vier Fälle:

I. Beide `case`-Ausdrücke enthalten keine default-Alternative.

II. Beide `case`-Ausdrücke enthalten eine default-Alternative.

III. Nur die Alternativen $(P_i \texttt{ -> } R_i)$ für $i = 1, \ldots, n$ enthalten eine default-Alternative.

IV. Nur die Alternativen $(Q_i \texttt{ -> } S_i)$ für $i = 1, \ldots, m$ enthalten eine default-Alternative.

In allen Fällen sind die übersetzten Ausdrücke der linken und rechten Seite der Transformationsregel kontextuell äquivalent. Die dazugehörigen Berechnungen sind im Anhang in Abschnitt A.6 angegeben. □

Da durch die (fcooc)-Transformation die Größe des Codes wächst (die m Alternativen existieren nach der Transformation n-mal), wird die Transformation in der Realität anders ausgeführt[9], indem so genannte „join points“ benutzt werden, d.h. die rechten Seiten der Alternativen werden wie folgt durch `letrec`-Bindungen geshared: Sei $Q_i = c_i \; y_{i,1} \ldots y_{i,n_i}$ für $i = 1, \ldots, m$, dann hat die rechte Seite der Transformation folgende Form

$$
\begin{array}{l}
\texttt{letrec} \quad s_1 = \lambda y_{1,1} \ldots y_{1,n_1} \texttt{ -> } S_1 \\
\qquad\quad \ldots \\
\qquad\quad s_m = \lambda y_{m,1} \ldots y_{m,n_m} \texttt{ -> } S_m \\
\texttt{in} \\
\quad \texttt{case } E \texttt{ of} \\
\qquad P_1 \texttt{ -> case } R_1 \texttt{ of} \\
\qquad\qquad\quad c_1 \; y_{1,1} \ldots y_{1,n_1} \texttt{ -> } s_1 \; y_{1,1} \ldots y_{1,n_1} \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\quad c_m \; y_{m,1} \ldots y_{m,n_m} \texttt{ -> } s_m \; y_{m,1} \ldots y_{m,n_m} \\
\qquad \ldots \\
\qquad P_n \texttt{ -> case } R_n \texttt{ of} \\
\qquad\qquad\quad c_1 \; y_{1,1} \ldots y_{1,n_1} \texttt{ -> } s_1 \; y_{1,1} \ldots y_{1,n_1} \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\quad c_m \; y_{m,1} \ldots y_{m,n_m} \texttt{ -> } s_m \; y_{m,1} \ldots y_{m,n_m}
\end{array}
$$

Diese Transformation ist korrekt, denn die s_i können zunächst mit der (bruinl)-Transformation in die rechten Seiten der Alternativen kopiert werden, danach können die Bindungen mit der (dcr-letrec)-Transformation entfernt werden und schließlich kann in jeder Alternative die (β-atom)-Transformation angewendet werden. Der erhaltene Ausdruck entspricht dem rechten Ausdruck aus der (fcooc)-Transformation.

[9] Vgl. [San95, Seiten 48-50], [PS94, Abschnitt 3.11] und [PS98, Abschnitte 5.1 und 5.2]

4.3.3.6 Die „case merging"-Transformation

Die „case merging"-Transformation[10] „mischt" die Alternativen verschachtelter `case`-Ausdrücke:

```
case x of                                          case x of
  c_1 a_{1,1} ... a_{1,ar(c_1)} -> t_1               c_1 a_{1,1} ... a_{1,ar(c_1)} -> t_1
  ...                                                ...
  c_k a_{k,1} ... a_{k,ar(c_k)} -> t_k               c_k a_{k,1} ... a_{k,ar(c_k)} -> t_k
  y ->                                  (cm) ===>    c_{k+1} b_{k+1,1} ... b_{k+1,ar(c_{k+1})} -> t_{k+1}[x/y]
    case x of                                        c_m b_{m,1} ... b_{m,ar(c_m)} -> t_m[x/y]
      c_{k+1} b_{k,1} ... b_{k,ar(c_{k+1})} -> t_{k+1}
      ...
      c_m b_{m,1} ... b_{m,ar(c_m)} -> t_m
```

wobei x eine Variable ist.

Lemma 4.24. *(cm) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Die Berechnung im Anhang in Abschnitt A.7 zeigt die Korrektheit. □

4.3.3.7 Die „alternative merging"-Transformation

Im Modul `ghc/compiler/simplCore/SimplUtils.lhs` wird mithilfe der Funktion `mkAlts` folgende Transformation durchgeführt, die `case`-Alternativen mit identischen rechten Seiten vereinigt:

```
case e of                                          case e of
  c_1 a_{1,1} ... a_{1,m_1} -> E_1;                  c_1 a_{1,1} ... a_{1,m_1} -> E_1;
  ...                                                ...
  c_i a_{i,1} ... a_{i,m_i} -> E;                    c_{i-1} a_{i-1,1} ... a_{i-1,m_{i-1}} -> E_{i-1};
  ...                                   (am) ===>    c_{j+1} a_{j+1,1} ... a_{j+1,m_{j+1}} -> E_{j+1};
  c_j a_{j,1} ... a_{j,m_j} -> E;                    ...
  c_{j+1} a_{j+1,1} ... a_{j+1,m_{j+1}} -> E_{j+1};  c_n a_{n,1} ... a_{n,m_n} -> E_n;
  ...                                                y -> E
  c_n a_{n,1} ... a_{n,m_n} -> E_n
```

wenn für $k = i, \ldots, j$ gilt: Die $a_{k,1} \ldots a_{k,m_k}$ kommen nicht frei in E vor.

Lemma 4.25. *(am) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

[10] Vgl. [PS94, Abschnitt 3.12] und [San95, Abschnitt 3.3.3].

Beweis. Seien $pat_{n+1} \ldots pat_m$ Pattern für Konstruktoren vom selben Typ wie e, $pat_{m+1} \ldots pat_N$ Pattern für Konstruktoren anderen Typs, die durch die Übersetzung entstanden seien. Für die zugehörigen Alternativen können wir die (crpl)-Reduktion anwenden, da e nicht zu einem solchen Konstruktor auswerten kann.

$$
\begin{aligned}
& [\![\texttt{case}\ e\ \texttt{of}\ \{P_1\ \texttt{->}\ E_1; \ldots P_i\ \texttt{->}\ E; \ldots P_j\ \texttt{->}\ E; P_{j+1}\ \texttt{->}\ E_{j+1}; \ldots P_n\ \texttt{->}\ E_n\}]\!] \\
\equiv\ & (\texttt{case}\ [\![e]\!]\ ([\![P_1]\!] \to [\![E_1]\!]) \ldots ([\![P_i]\!] \to [\![E]\!]) \ldots ([\![P_j]\!] \to [\![E]\!]) \\
& \qquad ([\![P_{j+1}]\!] \to [\![E_{j+1}]\!]) \ldots ([\![P_n]\!] \to [\![E_n]\!])\ (pat_{n+1} \to \bot) \ldots (pat_N \to \bot)) \\
\xleftarrow{gc}\ & (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ [\![e]\!]\ ([\![P_1]\!] \to [\![E_1]\!]) \ldots ([\![P_i]\!] \to [\![E]\!]) \ldots ([\![P_j]\!] \to [\![E]\!]) \\
& \qquad ([\![P_{j+1}]\!] \to [\![E_{j+1}]\!]) \ldots ([\![P_n]\!] \to [\![E_n]\!])(pat_{n+1} \to \bot) \ldots (pat_N \to \bot))) \\
\xleftarrow{(crpl)^*}\ & (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ [\![e]\!]\ ([\![P_1]\!] \to [\![E_1]\!]) \ldots ([\![P_i]\!] \to [\![E]\!]) \ldots ([\![P_j]\!] \to [\![E]\!]) \\
& \qquad ([\![P_{j+1}]\!] \to [\![E_{j+1}]\!]) \ldots ([\![P_n]\!] \to [\![E_n]\!])\ (pat_{n+1} \to [\![E]\!]) \ldots (pat_m \to [\![E]\!]) \\
& \qquad (pat_{m+1} \to \bot) \ldots (pat_N \to \bot))) \\
\xleftarrow{ucp}\ & (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ x\ ([\![P_1]\!] \to [\![E_1]\!]) \ldots ([\![P_i]\!] \to [\![E]\!]) \ldots ([\![P_j]\!] \to [\![E]\!]) \\
& \qquad ([\![P_{j+1}]\!] \to [\![E_{j+1}]\!]) \ldots ([\![P_n]\!] \to [\![E_n]\!])(pat_{n+1} \to [\![E]\!]) \ldots (pat_m \to [\![E]\!]) \\
& \qquad (pat_{m+1} \to \bot) \ldots (pat_N \to \bot))) \\
\equiv\ & [\![\texttt{case}\ e\ \texttt{of}\ \{P_1\ \texttt{->}\ E_1; \ldots; P_{i-1}\ \texttt{->}\ E_{i-1}; P_{j+1}\ \texttt{->}\ E_{j+1}; \ldots; P_n\ \texttt{->}\ E_n; y\ \texttt{->}\ E\}]\!]
\end{aligned}
$$

□

4.3.3.8 Die „case identity"-Transformation

Diese Transformation wird im Modul `ghc/compiler/simplCore/SimplUtils.lhs` mittels der Funktion `mkCase1` durchgeführt.

$$\texttt{case}\ e\ \texttt{of}\ \{P_1\ \texttt{->}\ P_1; \ldots; P_n\ \texttt{->}\ P_n\}\ \overset{\text{(ci)}}{\texttt{===>}}\ e$$

Wir zeigen im Folgenden, dass die Transformation dann $[\![\cdot]\!]$-korrekt ist, wenn die in Definition 3.73 definierte (streval)-Reduktion korrekt ist. Vermutlich lässt sich die Korrektheit der „case-identity"-Transformation auch ohne diese Bedingung zeigen, indem man eine entsprechende Transformation für den FUNDIO-Kalkül definiert und die Methode der Vertauschungs- und Gabeldiagramme verwendet.

Lemma 4.26. *Wenn Vermutung 3.74 wahr ist, dann ist (ci) eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Die Abstraktion $(\lambda i.i)$ ist eine strikte Abstraktion, denn

$$(\lambda i.i)\ \bot\ \xrightarrow{lbeta}\ (\texttt{letrec}\ i = \bot\ \texttt{in}\ i) \sim_c \bot,$$

wobei die kontextuelle Gleichheit aufgrund von Satz 3.33 gilt.

Somit können wir die (streval)-Reduktion für solche Abstraktionen anwenden.

Seien $pat_{n+1} \ldots pat_N$ Pattern für die Konstruktoren aus $\mathcal{C}$, die nicht durch $[\![P_1]\!]$, ..., $[\![P_n]\!]$ abgedeckt werden. Für die zu $pat_{n+1} \ldots pat_N$ zugehörigen Alternativen dürfen wir die (crpl)-Reduktion anwenden, da P_1, ..., P_n alle Konstruktoren abdecken zu denen e auswerten kann[11].

$$
\begin{array}{rl}
 & [\![\texttt{case}\ e\ \texttt{of}\ \{P_1\ \texttt{->}\ P_1; \ldots; P_n\ \texttt{->}\ P_n\}]\!] \\
\equiv & (\texttt{case}\ e\ ([\![P_1]\!] \to [\![P_1]\!]) \ldots ([\![P_n]\!] \to [\![P_n]\!])\ (pat_{n+1} \to \bot) \ldots (pat_N \to \bot)) \\
\xleftarrow{ucp} & (\texttt{letrec}\ y = e\ \texttt{in} \\
 & \quad (\texttt{case}\ y\ ([\![P_1]\!] \to [\![P_1]\!]) \ldots ([\![P_n]\!] \to [\![P_n]\!])\ (pat_{n+1} \to \bot) \ldots (pat_N \to \bot))) \\
\xleftarrow{(ccpcx)^*} & (\texttt{letrec}\ y = e\ \texttt{in} \\
 & \quad (\texttt{case}\ y\ ([\![P_1]\!] \to y) \ldots ([\![P_n]\!] \to y)\ (pat_{n+1} \to \bot) \ldots (pat_N \to \bot))) \\
\xrightarrow{(crpl)^*} & (\texttt{letrec}\ y = e\ \texttt{in} \\
 & \quad (\texttt{case}\ y\ ([\![P_1]\!] \to y) \ldots ([\![P_n]\!] \to y)\ (pat_{n+1} \to y) \ldots (pat_N \to y))) \\
\xleftarrow{(betavar)^*} & (\texttt{letrec}\ y = [\![e]\!]\ \texttt{in} \\
 & \quad (\texttt{case}\ y\ ([\![P_1]\!] \to ((\lambda x.x)\ y)) \ldots ([\![P_n]\!] \to ((\lambda x.x)\ y)) \\
 & \qquad (pat_{n+1} \to ((\lambda x.x)\ y)) \ldots (pat_N \to ((\lambda x.x)\ y)))) \\
\xleftarrow{streval} & (\lambda x.x)\ [\![e]\!] \\
\xrightarrow{lbeta} & (\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ x) \\
\xrightarrow{ucp} & [\![e]\!]
\end{array}
$$

□

Enthält der `case`-Ausdruck eine default-Alternative der Form `y -> y`, dann kann der `case`-Ausdruck ebenfalls durch sein erstes Argument ersetzt werden, was analog zum vorangegangenen Beweis gezeigt werden kann.

4.3.3.9 Die „case elimination"-Transformation

Wir definieren diese Transformation, wie sie im GHC[12] benutzt wird. Sie unterscheidet sich von [PS94, Abschnitt 3.9] und [San95, Abschnitt 3.3.2] dahingehend, dass Sharing beachtet wird.

$$\texttt{case}\ e\ \texttt{of}\ y\ \texttt{->}\ E \overset{\text{(ce)}}{\texttt{===>}} \texttt{let}\ y = e\ \texttt{in}\ E \qquad \text{wenn } e \neq \bot.$$

[11] Dies ist eine Bedingung an `case`-Ausdrücke in $L_{GHCCore}$, siehe Abschnitt 2.3.2.

[12] Sie wird in der Funktion `mkCase` im Modul `ghc/compiler/simplCore/SimplUtils.lhs` ausgeführt.

Lemma 4.27. *(ce) ist keine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Betrachte die Ausdrücke s und t mit $s \overset{\text{(ce)}}{\texttt{===>}} t$, wobei c eine Konstante sei:

$$\begin{aligned} s &= \texttt{case (unsafePerformIO getChar) of } y \texttt{ -> } ((\lambda z \texttt{ -> } c)\ y) \\ t &= \texttt{let } y = \texttt{(unsafePerformIO getChar) in } ((\lambda z \texttt{ -> } c)\ y) \end{aligned}$$

Sei nun $P = \emptyset$, so gilt $[\![s]\!]\Uparrow(P)$, aber $[\![t]\!]\Downarrow(P)$:

$$\begin{aligned} [\![s]\!] &\equiv (\texttt{letrec } l = (\texttt{case (IO } \mathcal{B}) \ \ldots) \texttt{ in } (\texttt{case } l \ \ldots (pat_i \ \rightarrow \ ((\lambda z.c)\ l)) \ldots)) \\ &\xrightarrow{n,IOr,?} \text{Ein IO-Paar wird benötigt!} \\ [\![t]\!] &\equiv (\texttt{letrec } y = (\texttt{case (IO } \mathcal{B}) \ \ldots)) \texttt{ in } ((\lambda z.c)\ y)) \\ &\xrightarrow{n,lbeta} (\texttt{letrec } y = (\texttt{case (IO } \mathcal{B}) \ \ldots)) \texttt{ in } (\texttt{letrec } z = y \texttt{ in } c)) \\ &\xrightarrow{n,lll} (\texttt{letrec } y = (\texttt{case (IO } \mathcal{B}) \ \ldots)), z = y \texttt{ in } c) \end{aligned}$$

Somit gilt $[\![s]\!] \not\sim_c [\![t]\!]$. □

Im Folgenden schränken wir die Regel durch Bedingungen an x ein, und zeigen, dass diese Form der „case-elimination“ $[\![\cdot]\!]$-korrekt ist.

Lemma 4.28. *Wenn e eine Abstraktion, ein primitiver Operator (mit positiver Stelligkeit), ein Literal oder eine (evtl. ungesättigte) Konstruktoranwendung ist, dann ist (ce) eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Betrachten wir zunächst die Fälle, dass e eine Abstraktion oder ein primitiver Operator mit positiver Stelligkeit ist. Dann ist $[\![e]\!]$ eine Abstraktion und wir können die (case-lam)-Reduktion des FUNDIO-Kalküls anwenden:

$$\begin{aligned} & [\![\texttt{case } e \texttt{ of } y \texttt{ -> } E]\!] \\ \equiv\ & (\texttt{letrec } x = [\![e]\!] \texttt{ in } (\texttt{case } x\ (pat_1 \rightarrow [\![E]\!][x/y]) \ldots (pat_N \rightarrow [\![E]\!][x/y]))) \\ \xrightarrow{cp}\ & (\texttt{letrec } x = [\![e]\!] \texttt{ in } (\texttt{case } [\![e]\!]\ (pat_1 \rightarrow [\![E]\!][x/y]) \ldots (pat_N \rightarrow [\![E]\!][x/y]))) \\ \xrightarrow{case-lam}\ & (\texttt{letrec } x = [\![e]\!] \texttt{ in } (\texttt{letrec } \{\} \texttt{ in } [\![E]\!][x/y])) \\ \xrightarrow{llet}\ & (\texttt{letrec } x = [\![e]\!] \texttt{ in } [\![E]\!][x/y]) \\ =_\alpha\ & (\texttt{letrec } y = [\![e]\!] \texttt{ in } [\![E]\!]) \\ \equiv\ & [\![\texttt{let } y = e \texttt{ in } E]\!] \end{aligned}$$

Wenn e ein Literal oder eine Konstante ist, so ist $[\![e]\!]$ ein Konstruktor der Stelligkeit 0 und die Berechnung ist analog zur Vorherigen mit der Ausnahme, dass anstelle der Reduktionsfolge $\xrightarrow{cp}\ .\ \xrightarrow{case-lam}$ eine (case-in)-Reduktion steht.

Nun betrachten wir den Fall, dass e eine gesättigte Konstruktoranwendung ist. O.B.d.A. sei $e = c\ a_1 \ldots a_{ar(c)}$.

$$[\![\texttt{case } (c\ a_1 \ldots a_{ar(c)}) \texttt{ of } y \texttt{ -> } E]\!]$$

$$
\begin{aligned}
&\equiv (\texttt{letrec}\ x = (c\ [\![a_1]\!] \ldots [\![a_{ar(c)}]\!])\ \texttt{in} \\
&\quad (\texttt{case}\ x\ (pat_1 \to [\![E]\!][x/y]) \ldots (c\ y_1\ \ldots\ y_{ar(c)} \to [\![E]\!][x/y]) \ldots (pat_N \to [\![E]\!][x/y])))\\
&\xrightarrow{case} (\texttt{letrec}\ x = (c\ z_1 \ldots z_{ar(c)}), z_1 = [\![a_1]\!], \ldots, z_{ar(c)} = [\![a_{ar(c)}]\!])\ \texttt{in} \\
&\quad (\texttt{letrec}\ y_1 = z_1, \ldots, y_{ar(c)} = z_{ar(c)}\ \texttt{in}\ [\![E]\!][x/y]))\\
&\xrightarrow{(gc)^*} (\texttt{letrec}\ x = (c\ z_1 \ldots z_{ar(c)}), z_1 = [\![a_1]\!], \ldots, z_{ar(c)} = [\![a_{ar(c)}]\!])\ \texttt{in}\ [\![E]\!][x/y])\\
&\xrightarrow{(ucp)^*} (\texttt{letrec}\ x = (c\ [\![a_1]\!] \ldots [\![a_{ar(c)}]\!])\ \texttt{in}\ [\![E]\!][x/y])\\
&=_\alpha (\texttt{letrec}\ y = (c\ [\![a_1]\!] \ldots [\![a_{ar(c)}]\!])\ \texttt{in}\ [\![E]\!])\\
&\equiv [\![\texttt{let}\ y = c\ a_1 \ldots a_{ar(c)}\ \texttt{in}\ E]\!]
\end{aligned}
$$

Abschließend sei e nun eine ungesättigte Konstruktoranwendung. Sei $e = c\ a_1 \ldots a_n$ mit $n < ar(c)$.

$$
\begin{aligned}
&[\![\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of}\ y\ \texttt{->}\ E]\!]\\
&\equiv (\texttt{letrec}\ x = (\lambda x_1 \ldots x_{ar(c)}.(c\ x_1 \ldots x_{ar(c)})\ [\![a_1]\!] \ldots [\![a_n]\!]\ \texttt{in} \\
&\quad (\texttt{case}\ x\ (pat_1 \to [\![E]\!][x/y]) \ldots (pat_N \to [\![E]\!][x/y])))\\
&\xrightarrow{(lll)^*} (\texttt{letrec}\ x = (\lambda x_{n+1} \ldots x_{ar(c)}.(c\ x_1 \ldots x_{ar(c)})), x_1 = [\![a_1]\!], \ldots, x_n = [\![a_n]\!]\ \texttt{in} \\
&\quad (\texttt{case}\ x\ (pat_1 \to [\![E]\!][x/y]) \ldots (pat_N \to [\![E]\!][x/y])))\\
&\xrightarrow{case-lam} (\texttt{letrec}\ x = (\lambda x_{n+1} \ldots x_{ar(c)}.(c\ x_1 \ldots x_{ar(c)})), x_1 = [\![a_1]\!], \ldots, x_n = [\![a_n]\!]\ \texttt{in} \\
&\quad (\texttt{letrec}\ \{\}\ \texttt{in}\ [\![E]\!][x/y]))\\
&\xrightarrow{llet} (\texttt{letrec}\ x = (\lambda x_{n+1} \ldots x_{ar(c)}.(c\ x_1 \ldots x_{ar(c)})), x_1 = [\![a_1]\!], \ldots, x_n = [\![a_n]\!]\ \texttt{in}\ [\![E]\!][x/y])\\
&\xleftarrow{llet} (\texttt{letrec}\ x = (\lambda x_1 \ldots x_{ar(c)}.(c\ x_1 \ldots x_{ar(c)}))\ [\![a_1]\!] \ldots [\![a_n]\!]\ \texttt{in}\ [\![E]\!][x/y])\\
&=_\alpha (\texttt{letrec}\ y = (\lambda x_1 \ldots x_{ar(c)}.(c\ x_1 \ldots x_{ar(c)}))\ [\![a_1]\!] \ldots [\![a_n]\!]\ \texttt{in}\ [\![E]\!])\\
&\equiv [\![\texttt{let}\ y = (c\ a_1 \ldots a_n)\ \texttt{in}\ E]\!]
\end{aligned}
$$

□

Man kann leicht nachrechnen, dass die (ce)-Transformation auch dann $[\![\cdot]\!]$-korrekt ist, wenn e eine Variable ist, an die ein Ausdruck e' gebunden ist, für den die Bedingungen an e aus Lemma 4.28 gelten.

4.3.4 Transformationen für `let`- und `case`-Ausdrücke

4.3.4.1 Die „floating applications inwards"-Transformationen

Verschiebung der Applikation in einen `let`-Ausdruck[13]:

$$(\texttt{let(rec)}\ Bind\ \texttt{in}\ E)\ arg \overset{\text{(fai-let)}}{\texttt{===>}} \texttt{let(rec)}\ Bind\ \texttt{in}\ (E\ arg)$$

[13] Vgl. [PS94, Abschnitt 3.2] und [San95, Abschnitt 3.4.1].

Lemma 4.29. *(fai-let) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.

$$\begin{aligned}
&[\![(\texttt{let(rec)}\ Bind\ \texttt{in}\ E)\ arg]\!] \equiv ((\texttt{letrec}\ [\![Bind]\!]\ \texttt{in}\ [\![E]\!])\ [\![arg]\!]) \\
&\xrightarrow{lapp} (\texttt{letrec}\ [\![Bind]\!]\ \texttt{in}\ ([\![E]\!]\ [\![arg]\!])) \equiv [\![\texttt{let(rec)}\ Bind\ \texttt{in}\ (E\ arg)]\!]
\end{aligned}$$

Somit wurde gezeigt:

$$[\![\texttt{(let Bind in E) arg}]\!] \sim_c [\![\texttt{let Bind in (E arg)}]\!]$$

□

Verschiebung der Applikation in einen `case`-Ausdruck[14]:

$$\left(\begin{array}{l}\texttt{case}\ E\ \texttt{of} \\ \quad P_1 \to E_1; \\ \quad \ldots; \\ \quad P_n \to E_n\end{array}\right) arg \overset{\text{(fai-case)}}{\texttt{===>}} \begin{array}{l}\texttt{case}\ E\ \texttt{of} \\ \quad P_1 \to E_1\ arg; \\ \quad \ldots; \\ \quad P_n \to E_n\ arg\end{array}$$

Lemma 4.30. *(fai-case) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Betrachten wir zunächst den Fall, dass keine default-Alternative in den Alternativen des `case`-Ausdrucks vorhanden ist:

$$\begin{aligned}
&[\![(\texttt{case}\ E\ \texttt{of}\ P_1\ \texttt{->}\ E_1; \ldots; P_n\ \texttt{->}\ E_n)\ arg]\!] \\
&\equiv ([\![(\texttt{case}\ E\ \texttt{of}\ P_1\ \texttt{->}\ E_1; \ldots; P_n\ \texttt{->}\ E_n)]\!]\ [\![arg]\!]) \\
&\equiv ((\texttt{case}\ [\![E]\!]\ ([\![P_1]\!] \to [\![E_1]\!]) \ldots ([\![P_n]\!] \to [\![E_n]\!])\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))\ [\![arg]\!]) \\
&\xrightarrow{capp} (\texttt{case}\ [\![E]\!]\ ([\![P_1]\!] \to ([\![E_1]\!]\ [\![arg]\!])) \ldots ([\![P_n]\!] \to ([\![E_n]\!]\ [\![arg]\!])) \\
&\qquad\qquad (p_{n+1} \to (\bot\ [\![arg]\!])) \ldots (p_N \to (\bot\ [\![arg]\!]))) \\
&\sim_c (\texttt{case}\ [\![E]\!]\ ([\![P_1]\!] \to ([\![E_1]\!]\ [\![arg]\!])) \ldots ([\![P_n]\!] \to ([\![E_n]\!]\ [\![arg]\!])) \\
&\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot) \\
&\equiv [\![\texttt{case}\ E\ \texttt{of}\ P_1\ \texttt{->}\ (E_1\ arg); \ldots; P_n\ \texttt{->}\ (E_n\ arg))]\!]
\end{aligned}$$

Die verwendete Gleichheit gilt aufgrund von Satz 3.33.

Betrachten wir nun den Fall, dass eine default-Alternative existiert. Falls neben der default-Alternative weitere Alternativen existieren, seien $p_{n+1} \ldots p_m$ die fehlenden Pattern vom gleichen Typ, wie die gegebenen Pattern und $p_{m+1} \ldots p_N$ die Pattern von allen anderen Typen. Falls nur die default-Alternative existiert, seien $p_{n+1} \ldots p_m$ Pattern für sämtliche Konstruktoren aus $\mathcal{C}$.

$$\begin{aligned}
&[\![(\texttt{case}\ E\ \texttt{of}\ P_1\ \texttt{->}\ E_1; \ldots; P_n\ \texttt{->}\ E_n; x\ \texttt{->}\ s)\ arg]\!] \\
&\equiv ([\![(\texttt{case}\ E\ \texttt{of}\ P_1\ \texttt{->}\ E_1; \ldots; P_n\ \texttt{->}\ E_n)]\!]\ [\![arg]\!])
\end{aligned}$$

[14] Vgl. [PS94, Abschnitt 3.2] und [San95, Abschnitt 3.5.1].

$$
\begin{array}{ll}
& \equiv ((\texttt{letrec}\ y = [\![E]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ [\![y]\!]\ ([\![P_1]\!] \to [\![E_1]\!]) \ldots ([\![P_n]\!] \to [\![E_n]\!]) \\
& \qquad\qquad (p_{n+1} \to [\![s]\!][y/x]) \ldots (p_m \to [\![s]\!][y/x]) \\
& \qquad\qquad (p_{m+1} \to \bot) \ldots (p_N \to \bot)))\ [\![arg]\!]) \\
\xrightarrow{lapp} & (\texttt{letrec}\ y = [\![E]\!]\ \texttt{in} \\
& \quad ((\texttt{case}\ [\![y]\!]\ ([\![P_1]\!] \to [\![E_1]\!]) \ldots ([\![P_n]\!] \to [\![E_n]\!]) \\
& \qquad\qquad (p_{n+1} \to [\![s]\!][y/x]) \ldots (p_m \to [\![s]\!][y/x]) \\
& \qquad\qquad (p_{m+1} \to \bot) \ldots (p_N \to \bot))\ [\![arg]\!])) \\
\xrightarrow{capp} & (\texttt{letrec}\ y = [\![E]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ [\![y]\!]\ ([\![P_1]\!] \to ([\![E_1]\!][\![arg]\!])) \ldots ([\![P_n]\!] \to ([\![E_n]\!][\![arg]\!])) \\
& \qquad\qquad (p_{n+1} \to ([\![s]\!][y/x][\![arg]\!])) \ldots (p_m \to ([\![s]\!][y/x][\![arg]\!])) \\
& \qquad\qquad (p_{m+1} \to (\bot\ [\![arg]\!])) \ldots (p_N \to (\bot\ [\![arg]\!])))) \\
\sim_c & (\texttt{letrec}\ y = [\![E]\!]\ \texttt{in} \\
& \quad (\texttt{case}\ [\![y]\!]\ ([\![P_1]\!] \to ([\![E_1]\!][\![arg]\!])) \ldots ([\![P_n]\!] \to ([\![E_n]\!][\![arg]\!])) \\
& \qquad\qquad (p_{n+1} \to ([\![s]\!][y/x][\![arg]\!])) \ldots (p_m \to ([\![s]\!][y/x][\![arg]\!])) \\
& \qquad\qquad (p_{m+1} \to \bot\ ) \ldots (p_N \to \bot\ ))) \\
& \equiv [\![\texttt{case}\ E\ \texttt{of}\ P_1\ \texttt{->}\ (E_1\ arg); \ldots; P_n\ \texttt{->}\ (E_n\ arg); x\ \texttt{->}\ (s\ arg)]\!]
\end{array}
$$

Somit wurde gezeigt:

$$
\left[\!\!\left[\left(\begin{array}{l} \texttt{case}\ E\ \texttt{of} \\ \quad P_1 \to E_1; \\ \quad \ldots; \\ \quad P_n \to E_n \end{array} \right)\ arg \right]\!\!\right] \sim_c \left[\!\!\left[\begin{array}{l} \texttt{case}\ E\ \texttt{of} \\ \quad P_1 \to E_1\ arg; \\ \quad \ldots; \\ \quad P_n \to E_n\ arg \end{array} \right]\!\!\right]
$$

□

4.3.4.2 Die „constructor reuse"-Transformationen

Diese Transformationen werden in [PS94, Abschnitt 3.15] und [San95, Abschnitt 3.3.2] wie folgt definiert:

Die Regel für `case`:

$$
\begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad \ldots \\ \quad c\ a_1 \ldots a_n\ \texttt{->}\ \ C[c\ a_1 \ldots\ a_n] \\ \quad \ldots \end{array} \quad \overset{\text{(cr-case)}}{\texttt{===>}} \quad \begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad \ldots \\ \quad c\ a_1 \ldots a_n\ \texttt{->}\ C[x] \\ \quad \ldots \end{array}
$$

wobei x eine Variable ist.

Lemma 4.31. *(cr-case) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Betrachten wir zunächst den Fall, dass der `case`-Ausdruck keine default-Alternative enthält. Seien $Alts'$ die zusätzlichen Alternativen, die durch die Übersetzung hinzugefügt werden.

$$\begin{aligned}
&[\![\texttt{case } x \texttt{ of } \ldots, c\ a_1 \ldots a_n \texttt{ -> } C[c\ a_1 \ldots\ a_n], \ldots]\!] \\
\equiv\ &(\texttt{case } [\![x]\!]\ [\![\ldots]\!]\ (([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!]) \to [\![C[c\ a_1 \ldots\ a_n]]\!])\ [\![\ldots]\!]\ Alts') \\
\equiv\ &(\texttt{case } x\ [\![\ldots]\!]\ (([\![c]\!]\ a_1 \ldots a_n) \to [\![C]\!][\![c]\!]\ a_1 \ldots\ a_n])\ [\![\ldots]\!]\ Alts') \\
\xleftarrow{ccpcx}\ &(\texttt{case } x\ [\![\ldots]\!]\ (([\![c]\!]\ a_1 \ldots a_n) \to [\![C]\!][x])\ [\![\ldots]\!]\ Alts') \\
\equiv\ &[\![\texttt{case } x\ \ldots,\ c\ a_1 \ldots a_n \texttt{ -> } C[x],\ \ldots]\!]
\end{aligned}$$

Betrachten wir nun den Fall, dass auch eine default-Alternative vorhanden ist. O.B.d.A. sei dies die Alternative y `->` s. Seien $Alts'$ die durch die Übersetzung zusätzlich entstandenen Alternativen.

$$\begin{aligned}
&[\![\texttt{case } x \texttt{ of } \ldots, c\ a_1 \ldots a_n \texttt{ -> } C[c\ a_1 \ldots\ a_n], \ldots, y \texttt{ -> } s]\!] \\
\equiv\ &(\texttt{letrec } z = x \texttt{ in } (\texttt{case } z\ [\![\ldots]\!]\ (([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!]) \to [\![C[c\ a_1 \ldots\ a_n]]\!])\ Alts')) \\
\equiv\ &(\texttt{letrec } z = x \texttt{ in } (\texttt{case } z\ [\![\ldots]\!]\ (([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!]) \to [\![C]\!][c\ a_1 \ldots\ a_n])\ Alts')) \\
\xrightarrow{cpx}\ &(\texttt{letrec } z = x \texttt{ in } (\texttt{case } x\ [\![\ldots]\!]\ (([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!]) \to [\![C]\!][c\ a_1 \ldots\ a_n])\ Alts')) \\
\xrightarrow{ccpcx}\ &(\texttt{letrec } z = x \texttt{ in } (\texttt{case } x\ [\![\ldots]\!]\ (([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!]) \to [\![C]\!][x])\ Alts')) \\
\xleftarrow{cpx}\ &(\texttt{letrec } z = x \texttt{ in } (\texttt{case } z\ [\![\ldots]\!]\ (([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!]) \to [\![C]\!][x])\ Alts')) \\
\equiv\ &[\![\texttt{case } x\ \ldots,\ c\ a_1 \ldots a_n \texttt{ -> } C[x],\ \ldots, y \texttt{ -> } s]\!]
\end{aligned}$$

□

Die Regel für `let`[15]:

$$\begin{array}{l} \texttt{let } x = c\ a_1 \ldots a_n \\ \texttt{in } C[c\ a_1 \ldots a_n] \end{array} \quad \overset{\text{(cr-let)}}{\texttt{===>}} \quad \begin{array}{l} \texttt{let } x = c\ a_1 \ldots a_n \\ \texttt{in } C[x] \end{array}$$

wenn die a_i atomar sind

Lemma 4.32. *(cr-let) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

[15] Für die Regel für `let` haben wir zusätzlich zu [San95, Seite 27] die Bedingung hinzugefügt, dass die Argumente der Konstruktoranwendung atomar sein müssen. In [San95] war dies nicht notwendig, da die Kernsprache keine anderen Konstruktoranwendungen erlaubte. Die Bedingung gilt jedoch auch in der aktuellen Implementierung, da der GHC `let`-gebundene Konstruktoranwendungen nur in dieser Form zulässt. Dies wird zum einen in [PM02, Seite 399] erwähnt, zum anderen ist dies im Quellcode des Moduls `ghc/compiler/coreSyn/CoreSyn.lhs` dokumentiert. Eine Konstruktoranwendung mit nicht atomaren Argumenten kann (mithilfe von korrekten Programmtransformationen) in die geforderte Form gebracht werden, indem die (uinl)-Transformation mehrfach rückwärts angewendet wird. Für den FUNDIO-Kalkül wird dieses Vorgehen mit der analogen (ucp)-Reduktion in [Sch03a, Seite 65] beschrieben.

Beweis. Für $i = 1, \ldots n$ ist $[\![a_i]\!]$ eine Variable oder eine Konstante, da a_i atomar ist. Somit liegt die Konstruktoranwendung $[\![c\ a_1 \ldots a_n]\!]$ in L_{cheap} und die Korrektheit kann mithilfe der (cpcheap)-Regel gezeigt werden:

$$\begin{aligned}
& [\![\texttt{let } x = c\ a_1\ \ldots\ a_n \texttt{ in } C[c\ a_1\ \ldots\ a_n]]\!] \\
\equiv\ & (\texttt{letrec } x = [\![c\ a_1 \ldots a_n]\!] \texttt{ in } [\![C[c\ a_1\ \ldots\ a_n]]\!]) \\
\equiv\ & (\texttt{letrec } x = [\![c\ a_1 \ldots a_n]\!] \texttt{ in } [\![C]\!][[\![c\ a_1\ \ldots\ a_n]]\!]) \\
\xleftarrow{cpcheap}\ & (\texttt{letrec } x = [\![c\ a_1 \ldots a_n]\!] \texttt{ in } [\![C]\!][x]) \\
\equiv\ & [\![\texttt{let } x = c\ a_1\ \ldots\ a_n \texttt{ in } C[x]]\!]
\end{aligned}$$

□

4.3.5 Transformationen, die Striktheit ausnutzen

Die folgenden Transformationen nutzen Striktheit aus. Die $[\![\cdot]\!]$-Korrektheit der Transformationen zeigen wir nur unter der Annahme, dass die in Definition 3.73 eingeführte (streval)-Reduktion korrekt ist.

4.3.5.1 Die „let-to-case"-Transformation

Diese Transformation[16] nutzt Striktheit aus, um einen durch ein `let`-gebundenen Ausdruck früher auszuwerten, wenn bekannt ist, dass der Wert des Ausdrucks benötigt wird.

$\texttt{let } v = E_1 \texttt{ in } E_2 \overset{\text{(ltc)}}{\texttt{===>}} \texttt{case } E_1 \texttt{ of } v \texttt{ -> } E_2$
wenn v einen Konstruktortyp hat, E_2 strikt in v und E_1 keine WHNF ist.

Lemma 4.33. *Wenn Vermutung 3.74 wahr ist, dann ist (ltc) eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis.

$$\begin{aligned}
& [\![\texttt{let } v = E_1 \texttt{ in } E_2]\!] \\
\equiv\ & (\texttt{letrec } v = [\![E1]\!] \texttt{ in } [\![E_2]\!]) \\
\xleftarrow{lbeta}\ & (\lambda v.[\![E_2]\!])\ [\![E_1]\!] \\
\xrightarrow{streval}\ & (\texttt{letrec } w = [\![E_1]\!] \texttt{ in } (\texttt{case } w\ (pat_1 \to ((\lambda v.[\![E_2]\!])\ w)) \ldots (pat_N \to ((\lambda v.[\![E_2]\!])\ w)))) \\
\xrightarrow{(betavar)^*}\ & (\texttt{letrec } w = [\![E_1]\!] \texttt{ in } (\texttt{case } w\ (pat_1 \to [\![E_2]\!][w/v]) \ldots (pat_N \to [\![E_2]\!][w/v]))) \\
\equiv\ & [\![\texttt{case } E_1 \texttt{ of } v \texttt{ -> } E_2]\!]
\end{aligned}$$

□

[16] Vgl. [PS94, Abschnitt 3.4] und [San95, Abschnitt 3.6.1].

4.3.5.2 Die „unboxing let-to-case“-Transformation

Diese Transformation[17] ist eine angepasste Variante für spezielle Konstruktoren und insbesondere „unboxed values“:

$$\texttt{let } v = E_1 \texttt{ in } E_2 \overset{\text{(ultc)}}{\texttt{===>}} \texttt{case } E_1 \texttt{ of } c\ a_1 \ldots a_n \texttt{ -> let } v = c\ a_1 \ldots a_n \texttt{ in } E_2$$

wenn v einen Konstruktortyp hat, der aus einem einzigen Datenkonstruktor c besteht, und E_2 strikt in v ist.

Lemma 4.34. *Wenn Vermutung 3.74 wahr ist, dann ist (ultc) eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Die Typbedingung interpretieren wir derart, dass wir für alle `case`-Alternativen, bis auf die Alternative für den Konstruktor $[\![c]\!]$, die (crpl)-Reduktion anwenden dürfen.

$$\begin{aligned}
& [\![\texttt{let } v = E_1 \texttt{ in } E_2]\!] \\
\equiv\ & (\texttt{letrec } v = [\![E1]\!] \texttt{ in } [\![E_2]\!]) \\
\xleftarrow{lbeta}\ & (\lambda v.[\![E_2]\!])\ [\![E_1]\!] \\
\xrightarrow{streval}\ & (\texttt{letrec } w = [\![E_1]\!] \texttt{ in } (\texttt{case } w\ (pat_1 \to ((\lambda v.[\![E_2]\!])\ w)) \ldots (pat_N \to ((\lambda v.[\![E_2]\!])\ w)))) \\
\xrightarrow{(betavar)^*}\ & (\texttt{letrec } w = [\![E_1]\!] \texttt{ in } (\texttt{case } w\ (pat_1 \to [\![E_2]\!][w/v]) \ldots (pat_N \to [\![E_2]\!][w/v]))) \\
\xrightarrow{(crpl)^*}\ & (\texttt{letrec } w = [\![E_1]\!] \texttt{ in} \\
& \quad (\texttt{case } w\ (pat_1 \to \bot) \ldots ([\![c]\!]\ a_1 \ldots a_n) \to [\![E_2]\!][w/v]) \ldots (pat_N \to \bot))) \\
\xrightarrow{ccpcx}\ & (\texttt{letrec } w = [\![E_1]\!] \texttt{ in} \\
& \quad (\texttt{case } w\ (pat_1 \to \bot) \ldots ([\![c]\!]\ a_1 \ldots a_n) \to [\![E_2]\!][([\![c]\!]\ a_1 \ldots a_n)/v]) \ldots (pat_N \to \bot))) \\
\xrightarrow{ucp}\ & (\texttt{letrec } \{\} \texttt{ in} \\
& \quad (\texttt{case } [\![E_1]\!]\ (pat_1 \to \bot) \ldots ([\![c]\!]\ a_1 \ldots a_n) \to [\![E_2]\!][([\![c]\!]\ a_1 \ldots a_n)/v]) \ldots (pat_N \to \bot))) \\
\xrightarrow{gc}\ & (\texttt{case } [\![E_1]\!]\ (pat_1 \to \bot) \ldots ([\![c]\!]\ a_1 \ldots a_n) \to [\![E_2]\!][([\![c]\!]\ a_1 \ldots a_n)/v]) \ldots (pat_N \to \bot)) \\
\xleftarrow{gc}\ & (\texttt{case } [\![E_1]\!] \\
& \quad (pat_1 \to \bot) \ldots \\
& \quad ([\![c]\!]\ a_1 \ldots a_n) \to (\texttt{letrec } v = ([\![c]\!]\ a_1 \ldots a_n) \texttt{ in } [\![E_2]\!][([\![c]\!]\ a_1 \ldots a_n)/v])) \ldots \\
& \quad (pat_N \to \bot)) \\
\xleftarrow{cpcheap}\ & (\texttt{case } [\![E_1]\!] \\
& \quad (pat_1 \to \bot) \ldots \\
& \quad ([\![c]\!]\ a_1 \ldots a_n) \to (\texttt{letrec } v = ([\![c]\!]\ a_1 \ldots a_n) \texttt{ in } [\![E_2]\!])) \ldots \\
& \quad (pat_N \to \bot)) \\
\equiv\ & [\![\texttt{case } E_1 \texttt{ of } \{c\ a_1 \ldots a_n \texttt{ -> let } v = c\ a_1 \ldots a_n \texttt{ in } E_2\}]\!]
\end{aligned}$$

[17] Vgl. [PS94, Abschnitt 3.5] und [San95, Abschnitt 3.6.2].

□

4.3.5.3 Die „floating case out of let"-Transformation

Die „floating case out of let"-Transformation[18] ist wie folgt definiert:

$$\begin{array}{l}\texttt{let } v = \texttt{case } E_1 \texttt{ of } \{c_1\ a_{1,1} \ldots a_{1,m_1} \texttt{ -> } t_1; \ldots; c_n\ a_{n,1} \ldots a_{n,m_n} \texttt{ -> } t_n\} \\ \texttt{in } E_3]\!] \\ \overset{\text{(fcool)}}{\texttt{===>}}\ \texttt{case } E_1 \texttt{ of} \\ \qquad \{c_1\ a_{1,1} \ldots a_{1,m_1} \texttt{ -> let } v = t_1 \texttt{ in } E_3; \\ \qquad\ \ldots \\ \qquad\ c_n\ a_{n,1} \ldots a_{n,m_n} \texttt{ -> let } v = t_n \texttt{ in } E_3\}\end{array}$$

wenn E_3 strikt in v und v nicht frei in E_1 ist.

Lemma 4.35. *Wenn Vermutung 3.74 wahr ist, dann ist (fcool) eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Die Korrektheit wird gezeigt, indem die beiden Ausdrücke mittels der Übersetzung $[\![\cdot]\!]$ nach L_{FUNDIO} übersetzt werden und dann mithilfe von korrekten Programmtransformationen und der Regel (streval) ineinander überführt werden. Die zugehörige Berechnung ist im Anhang in Abschnitt A.8 angegeben. □

4.3.6 Andere Transformationen

4.3.6.1 Eta-Expansion und -Reduktion

$$v = \lambda x_1 \ldots x_n \texttt{ ->}\ f\ x_1 \ldots x_n \quad \overset{(\eta\text{-exp})}{\texttt{===>}} \quad v = \lambda x_1 \ldots x_n \ldots x_m \texttt{ ->}\ f\ x_1 \ldots x_n \ldots x_m$$

wenn f Stelligkeit m hat und $n < m$.

Das hierbei zugrunde gelegte Konzept der Stelligkeit unterscheidet sich von dem sonst gebräuchlichen. Die Literatur[19] gibt nur eine ungenaue Definition, indem die Stelligkeit als „maximale Anzahl an Lambdas" des Ausdrucks bezeichnet wird, womit die Anzahl an Argumenten, gemeint ist, die übergeben werden können, ohne das „Arbeit" verrichtet wird, wie z.B. das Auswerten eines `case`- oder `let`-Ausdrucks.

[18] Vgl. [PS94, Abschnitt 3.6] und [San95, Abschnitt 3.5.3].
[19] Vgl. [PS94, Abschnitt 3.13] und [San95, Abschnitt 3.7.2].

Lemma 4.36. *(η-exp) ist keine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Betrachte folgende Ausdrücke s und t mit $s \overset{(\eta\text{-exp})}{\texttt{===>}} t$, sowie deren Übersetzungen $[\![s]\!]$ und $[\![t]\!]$:

$$\begin{aligned}
s &= \texttt{let } fun = (\lambda x_1\ x_2\ \texttt{->}\ x_1)\ (\texttt{unsafePerformIO getChar}) \\
&\quad \texttt{in case } (fun\ \texttt{False}) \texttt{ of } \{\texttt{'a' -> 'a'; 'b' -> } (fun\ \texttt{False})\} \\
t &= \texttt{let } fun = \lambda y\ \texttt{->}\ ((\lambda x_1\ x_2\ \texttt{->}\ x_1)\ (\texttt{unsafePerformIO getChar})\ y) \\
&\quad \texttt{in case } (fun\ \texttt{False}) \texttt{ of } \{\texttt{'a' -> 'a'; 'b' -> } (fun\ \texttt{False})\} \\
[\![s]\!] &= (\texttt{letrec } fun = ((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case } (\texttt{IO } \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)) \\
&\quad \texttt{in } (\texttt{case } (fun\ \texttt{False})\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)) \\
[\![t]\!] &= (\texttt{letrec } fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case } (\texttt{IO } \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)) \\
&\quad \texttt{in } (\texttt{case } (fun\ \texttt{False})\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))
\end{aligned}$$

Sei $P = \{(\mathcal{B}, \texttt{'b'})\}$, so gilt $[\![s]\!]\Downarrow(P)$, aber $[\![t]\!]\Uparrow(P)$. Die zugehörige Berechnung ist im Anhang in Abschnitt A.9 zu finden. Somit folgt $[\![s]\!] \not\sim_c [\![t]\!]$ □

Zusätzlich zur eben dargestellten Transformation wird die η-Erweiterung für besondere `case`-Ausdrücke durchgeführt:

$$\begin{array}{lcl}
\texttt{case } e \texttt{ of} & & \lambda y\ \texttt{->}\ \texttt{case } e \texttt{ of} \\
\quad p_1\ \texttt{->}\ e_1 & \overset{(\eta\text{-exp-case})}{\texttt{===>}} & \quad p_1\ \texttt{->}\ e_1\ y \\
\quad \ldots & & \quad \ldots \\
\quad p_n\ \texttt{->}\ e_n & & \quad p_n\ \texttt{->}\ e_n\ y
\end{array}$$

wenn

- e ist eine Variable und
- alle rechten Seiten der Alternativen sind Funktionen
- alle rechten Seiten der Alternativen sind WHNFs.

Eigentlich können wir in $L_{GHCCore}$ nicht ohne Weiteres feststellen, ob die rechten Seiten der Alternativen Funktionen sind. Wir nehmen aber an, dass wir uns diese Information beim Übergang der getypten Kernsprache nach $L_{GHCCore}$ merken, denn in der getypten Kernsprache müssen die rechten Seiten alle einen funktionalen Typ besitzen.

Lemma 4.37. *Die Regel (η-exp-case) ist keine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Zum einen wäre die Transformation falsch, wenn e an einen nichtterminierenden Ausdruck gebunden ist. Denn in diesem Fall terminiert der linke Ausdruck nicht, der rechte Ausdruck befindet sich jedoch in WHNF.

Selbst wenn wir annehmen, dass e an einen terminierenden Ausdruck gebunden ist, ist die Transformation nicht korrekt, was folgendes Beispiel zeigt:

Betrachte folgende Ausdrücke s, $t \in L_{GHCCore}$, wobei c eine Konstante sei:

$s =$ `letrec` $z = ($`unsafePerformIO getChar`$); f = \lambda x$ `->` `case` z `of` $\{u$ `->` $(\lambda w$ `->` $w)\}$
`in case` $(f$ `True`$)$ `of` $\{v$ `->` `'a'`$\}$

$t =$ `letrec` $z = ($`unsafePerformIO getChar`$); f = \lambda x$ `->` $(\lambda y$ `->` `case` z `of` $\{u$ `->` $(\lambda w$ `->` $w)$ $y\}$
`in case` $(f$ `True`$)$ `of` $\{v$ `->` `'a'`$\}$

t kann durch Anwendung einer (η-case)-Transformation aus s erhalten werden. Übersetzt man die Ausdrücke s und t in FUNDIO und so erhält man:

$$
\begin{aligned}
[\![s]\!] \equiv\ & (\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot)),\\
& \qquad f = (\lambda x.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w)) \ldots (p_N \to (\lambda w.w)))))\\
& \texttt{in}\ (\texttt{letrec}\ v' = (f\ \texttt{True})\ \texttt{in}\ (\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))))\\
[\![t]\!] \equiv\ & (\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot)),\\
& \qquad f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))\\
& \texttt{in}\ (\texttt{letrec}\ v' = (f\ \texttt{True})\ \texttt{in}\ (\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))))
\end{aligned}
$$

Sei nun $P = \emptyset$. Dann gilt $[\![t]\!]\Downarrow(P)$ und $[\![s]\!]\Uparrow(P)$, was die Berechnung im Anhang in Abschnitt A.10 zeigt. □

Im Anhang in Abschnitt B.2 ist ein Programm angegeben, dass dem Beispiel aus dem obigen Beweis weitgehend entspricht. Die Auswertung dieser Programms zeigt, dass das Verhalten des Programms durch die (η-exp-case)-Transformation verändert wird.

Im Folgenden stellen wir je eine Variante der Eta-Reduktion sowie der Eta-Expansion dar und beweisen, dass diese Transformationen korrekt bzgl. der FUNDIO-Semantik sind.

Sei die Transformation (η-red) auf $L_{GHCCore}$ wie folgt definiert:

$$
\begin{array}{l}
\texttt{let(rec)} \\
\quad f = g_1; g_1 = g_2; \ldots g_k = u; \ldots \\
\texttt{in}\ \lambda x_1 \ldots x_n\ \texttt{->}\ (f\ x_1 \ldots\ x_n)
\end{array}
\quad \overset{(\eta\text{-red})}{\texttt{===>}} \quad
\begin{array}{l}
\texttt{let(rec)} \\
\quad f = g_1; g_1 = g_2; \ldots g_k = u; \ldots \\
\texttt{in}\ f
\end{array}
$$

wenn eine der folgenden Bedingungen gilt

(1) $u = \lambda y_1 \ldots y_m$ `->` e

(2) $u = primop$ und $ar(primop) = m$

(3) $u = c_i\ a_1 \ldots a_{m'}$ und $ar(c_i) = (m + m')$

und $m \geq n$

Lemma 4.38. *(η-red) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Wir betrachten zunächst die Fälle (1) und (2). In beiden Fällen gilt $[\![u]\!] \equiv (\lambda y_1.(\ldots(\lambda y_m.v)\ldots)$, wobei $v \equiv [\![e]\!]$ in Fall (1) bzw. v der Rumpf der Übersetzung des primitiven Operators ist.

$$
\begin{array}{rl}
& [\![\texttt{let(rec)}\ f = g_1; g_1 = g_2; \ldots g_k = u; \ldots\ \texttt{in}\ \lambda x_1 \ldots x_n\ \texttt{->}\ (f\ x_1 \ldots\ x_n)]\!] \\
\equiv & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, g_k = (\lambda y_1.(\ldots(\lambda y_m.v)\ldots); Env \\
& \quad \texttt{in}\ (\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.(\ldots((f\ x_1)\ x_2)\ldots\ x_n)\ldots)) \\
\xrightarrow{cp} & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, g_k = (\lambda y_1.(\ldots(\lambda y_m.v)\ldots); Env \\
& \quad \texttt{in}\ (\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.(\ldots(((\lambda y_1.(\ldots(\lambda y_m.v)\ldots)\ x_1)\ x_2)\ldots\ x_n)\ldots)) \\
\xrightarrow{(betavar)^*} & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, g_k = (\lambda y_1.(\ldots(\lambda y_m.v)\ldots); Env \\
& \quad \texttt{in}\ (\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.(\lambda y_{n+1}.(\ldots(\lambda y_m.v[x_i/y_i]_{i=1}^n)\ldots)) \\
=_\alpha & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, g_k = (\lambda y_1.(\ldots(\lambda y_m.v)\ldots); Env \\
& \quad \texttt{in}\ (\lambda y_1.(\ldots(\lambda y_m.v)\ldots)) \\
\xleftarrow{cp} & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, g_k = (\lambda y_1.(\ldots(\lambda y_m.v)\ldots); Env \\
& \quad \texttt{in}\ f) \\
\equiv & [\![\texttt{let(rec)} f = g_1; g_1 = g_2; \ldots, g_k = u; \ldots\ \texttt{in}\ f]\!]
\end{array}
$$

Nun betrachten wir Fall (3), d.h. u ist eine ungesättigte Konstruktoranwendung, wobei die Stelligkeit des Konstruktors mindestens so groß ist, dass noch n Argumente hinzugefügt werden können. Wir benutzen in der folgenden Berechnung die Notation $(\lambda z_1 \ldots z_n.s)$ für Ausdrücke der Form $(\lambda z_1.(\lambda z_2.(\ldots(\lambda z_n.s))\ldots)$

$$
\begin{array}{rl}
& [\![\texttt{let(rec)}\ f = g_1; g_1 = g_2; \ldots g_k = c_i\ a_1 \ldots a_{m'}; \ldots\ \texttt{in}\ \lambda x_1 \ldots x_n\ \texttt{->}\ (f\ x_1 \ldots x_n)]\!] \\
\equiv & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, \\
& \qquad g_k = (\ldots(\lambda y_1 \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'}))\ [\![a_1]\!])\ \ldots [\![a_{m'}]\!]), Env \\
& \quad \texttt{in}\ (\lambda x_1 \ldots x_n.(\ldots(f\ x_1)\ldots x_n)\ldots)) \\
\xrightarrow{(lll)^*} & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, \\
& \qquad g_k = (\lambda y_{m'+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'})) \\
& \qquad y_1 = [\![a_1]\!], \ldots, y_{m'} = [\![a_{m'}]\!], Env \\
& \quad \texttt{in}\ (\lambda x_1 \ldots x_n.(\ldots(f\ x_1)\ldots x_n)\ldots)) \\
\xrightarrow{(cp)} & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, \\
& \qquad g_k = (\lambda y_{m'+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'})) \\
& \qquad y_1 = [\![a_1]\!], \ldots, y_{m'} = [\![a_{m'}]\!], Env \\
& \quad \texttt{in}\ (\lambda x_1 \ldots x_n.(\ldots((\lambda y_{m'+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'}))\ x_1)\ldots x_n)\ldots)) \\
\xrightarrow{(betavar)^*} & (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, \\
& \qquad g_k = (\lambda y_{m'+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'})) \\
& \qquad y_1 = [\![a_1]\!], \ldots, y_{m'} = [\![a_{m'}]\!], Env \\
& \quad \texttt{in}\ (\lambda x_1 \ldots x_n\ y_{m'+n+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1 \ldots y_{m'}\ x_1 \ldots x_n\ y_{m'+n+1} \ldots y_{m+m'})))
\end{array}
$$

$$
\begin{aligned}
&=_\alpha (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, \\
&\qquad g_k = (\lambda y_{m'+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'})) \\
&\qquad y_1 = [\![a_1]\!], \ldots, y_{m'} = [\![a_{m'}]\!], Env \\
&\quad \texttt{in}\ (\lambda y_{m'+1} \ldots y_{m'+n}\ y_{m'+n+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1 \ldots y_{m+m'}))) \\
&\xleftarrow{cp} (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, \\
&\qquad g_k = (\lambda y_{m'+1} \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'})) \\
&\qquad y_1 = [\![a_1]\!], \ldots, y_{m'} = [\![a_{m'}]\!], Env \\
&\quad \texttt{in}\ f) \\
&\xleftarrow{(lll)^*} (\texttt{letrec}\ f = g_1, g_1 = g_2, \ldots, \\
&\qquad g_k = (\ldots(\lambda y_1 \ldots y_{m+m'}.([\![c_i]\!]\ y_1\ \ldots y_{m+m'})))[\![a_1]\!]) \ldots [\![a_{m'}]\!]) \\
&\quad \texttt{in}\ f) \\
&\equiv [\![\texttt{let(rec)}\ f = g_1; g_1 = g_2; \ldots; g_k = u; \ldots\ \texttt{in}\ f]\!]
\end{aligned}
$$

□

Wie wir bereits festgehalten haben, ist die allgemeine Eta-Expansion nicht $[\![\cdot]\!]$-korrekt. Eine abgeschwächte Form, der ein einfacheres Konzept von Stelligkeit zugrunde liegt, ist jedoch korrekt.

Definition 4.39. *Sei ar_η wie folgt definiert:*

$$
ar_\eta(x) = \begin{cases}
m, & \text{wenn } x \text{ ein prim. Operator mit Stelligkeit } m \text{ ist} \\
m, & \text{wenn } x \text{ ein Konstruktor mit Stelligkeit } m \text{ ist} \\
1 + ar_\eta(s) & \text{wenn } x = \lambda y.s \\
max\{0, ar_\eta(a) - 1\}, & \text{wenn } x = (a\ b) \text{ und } b \in CHEAP \\
0, & \text{sonst}
\end{cases}
$$

$$
f \overset{\text{(eeta-exp)}}{\texttt{===>}} \lambda x_1 \ldots x_n\ \texttt{->}\ (f\ x_1 \ldots x_n) \qquad \text{wenn } ar_\eta(f) = n
$$

Lemma 4.40. *(eeta-exp) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation*

Beweis. Betrachten wir einen Ausdruck f mit $ar_\eta(f) = k > 0$.

$[\![f]\!]$ kann zu einem Ausdruck $\lambda y_1 \ldots y_k.s$ transformiert werden, indem alle Anwendungen von einem Argument aus L_{cheap} auf eine Abstraktion, wie folgt beta-reduziert werden. Sei $((\lambda y.t)\ t')$ eine solche Anwendung:

$$
\begin{aligned}
((\lambda y.t)\ t') &\xrightarrow{lbeta} (\texttt{letrec}\ y = t'\ \texttt{in}\ t) \\
&\xrightarrow{(cpcheap)^*} (\texttt{letrec}\ y = t' \texttt{in}\ t[t'/y]) \xrightarrow{gc} (t[t'/y])
\end{aligned}
$$

Dieses Verfahren wird solange iteriert, bis keine solchen Anwendungen mehr existieren.

Expansion von $(\lambda y_1 \ldots y_k.s)$ zu $\lambda x_1 \ldots x_k.((\lambda y_1 \ldots y_k.s)\ x_1 \ldots x_k)$ entspricht einer Umbenennung von gebundenen Variablen:

$$\lambda x_1 \ldots x_k.((\lambda y_1 \ldots y_k.s)\ x_1 \ldots x_k) \xrightarrow{(betavar)^*} (\lambda x_1 \ldots x_k.s[x_i/y_i]_{i=1}^k)$$

Anschließend kann die oben beschriebene Beta-Reduktion rückgängig gemacht werden und der erhaltene Ausdruck entspricht $[\![\lambda x_1 \ldots x_k \texttt{ -> } (f\ x_1 \ldots x_k)]\!]$ und somit gilt $[\![f]\!] \sim_c [\![\lambda x_1 \ldots x_k \texttt{ -> } (f\ x_1 \ldots x_k)]\!]$. □

Die Transformation ist auch korrekt, wenn f eine Variable ist, und es existiert eine `let(rec)`-Bindung $f = g$ und $ar_\eta(g) = n$, denn man kann wie im vorherigen Beweis $[\![g]\!]$ zu einer Abstraktion transformieren und dann mittels der (cp)-Reduktion anstelle von f kopieren, die (eeta-exp)-Transformation anwenden und schließlich mittels einer rückwärts angewendeten (cp)-Reduktion f anstelle von g ersetzen.

4.3.6.2 Die „constant folding"-Transformation

Diese Transformation[20] erlaubt es zur Compilezeit laufzeitunabhängige Ausdrücke auszuwerten:

$$primop\ lit_1 \ldots lit_{ar(c)} \overset{\text{(cf)}}{\texttt{===>}} erg$$

wobei *primop* ein seiteneffektfreier primitiver Operator ist, $lit_1 \ldots lit_{ar(c)}$ unboxed Werte sind und erg das Ergebnis von $primop\ lit_1 \ldots lit_{ar(c)}$ ist.

Lemma 4.41. *(cf) ist eine $[\![\cdot]\!]$-korrekte Programmtransformation.*

Beweis. Dies ist offensichtlich, da die übersetzten Ausdrücke mithilfe von deterministischen Reduktionen des FUNDIO-Kalküls ineinander überführt werden können, und diese Reduktionen sind korrekte Programmtransformationen. Wir illustrieren dies an einem Beispiel:

$$\begin{aligned}
& [\![\texttt{1\# +\# 2\#}]\!] \\
\equiv\ & (((\lambda a_1.(\lambda a_2.(\texttt{case}\ a_1 \ldots ([\![\texttt{1\#}]\!] \to (\texttt{case}\ a_2 \ldots ([\![\texttt{2\#}]\!] \to [\![\texttt{3\#}]\!]) \ldots)) \ldots))))\ [\![\texttt{1\#}]\!])\ [\![\texttt{2\#}]\!]) \\
\xrightarrow{(lll)^*}\ & (\texttt{letrec}\ a_1 = [\![\texttt{1\#}]\!]; a_2 = [\![\texttt{2\#}]\!]\ \texttt{in}\ (\texttt{case}\ a_1 \ldots ([\![\texttt{1\#}]\!] \to (\texttt{case}\ a_2 \ldots ([\![\texttt{2\#}]\!] \to [\![\texttt{3\#}]\!]) \ldots)) \ldots)) \\
\xrightarrow{case}\ & (\texttt{letrec}\ a_1 = [\![\texttt{1\#}]\!]; a_2 = [\![\texttt{2\#}]\!]\ \texttt{in}\ (\texttt{letrec}\ \{\}\ \texttt{in}\ (\texttt{case}\ a_2 \ldots ([\![\texttt{2\#}]\!] \to [\![\texttt{3\#}]\!]) \ldots))) \\
\xrightarrow{llet}\ & (\texttt{letrec}\ a_1 = [\![\texttt{1\#}]\!]; a_2 = [\![\texttt{2\#}]\!]\ \texttt{in}\ (\texttt{case}\ a_2 \ldots ([\![\texttt{2\#}]\!] \to [\![\texttt{3\#}]\!]) \ldots)) \\
\xrightarrow{case}\ & (\texttt{letrec}\ a_1 = [\![\texttt{1\#}]\!]; a_2 = [\![\texttt{2\#}]\!]\ \texttt{in}\ [\![\texttt{3\#}]\!]) \\
\xrightarrow{(gc)^*}\ & [\![\texttt{3\#}]\!]
\end{aligned}$$

□

[20] Vgl. [San95, Abschnitt 3.7.1].

4.3.7 Ergebnisse

Wir haben nun gesehen, welche lokalen Programmtransformationen auch bei uneingeschränkter Benutzung des `unsafePerformIO` korrekt bzgl. der FUNDIO-Semantik sind. Wir halten dies im folgenden Theorem fest.

Theorem 4.42. *Die Transformationen (β-atom), (β), (flool-let), (flool-letrec), (flooacs), (dcr-let), (dcr-letrec), (uinl), (bruinl), (cheapinl), (cokc), (cokc-default), (dbe), (dae), (coe), (fcooc), (cm), (am), (fai-let), (fai-case), (cr-case), (cr-let), (η-red), (eeta-exp), (cf) sind $[\![\cdot]\!]$-korrekt.*

Die Transformationen (ci),(ltc),(ultc), (fcool) sind $[\![\cdot]\!]$-korrekt, wenn Vermutung 3.74 wahr ist.

Beweis. Dies folgt aus den Lemmata 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.14, 4.15, 4.17, 4.18, 4.19, 4.20, 4.21, 4.22, 4.23, 4.24, 4.25, 4.29, 4.30, 4.31, 4.32, 4.38, 4.40, 4.41, 4.26, 4.33, 4.34 und 4.35. □

Diese Transformationen können somit auch weiterhin im GHC verwendet werden.

Des Weiteren sind einige Programmtransformationen nicht korrekt, was wir ebenso festhalten:

Theorem 4.43. *Die Transformationen (inl), (ce), (η-exp), (η-exp-case) sind nicht $[\![\cdot]\!]$-korrekt.*

Beweis. Dies folgt aus den Lemmata 4.13, 4.27, 4.36 und 4.37 □

Diese Transformationen müssen in der Implementierung abgeschaltet bzw. modifiziert werden.

4.4 Globale Transformationen

Wir werden nun kurz auf die einzelnen globalen Transformationen eingehen, wobei eine genaue Überprüfung der Transformationen innerhalb dieser Arbeit zu aufwendig ist. Im Folgenden werden wir zunächst eine Transformation darstellen, deren Korrektheit leicht zu zeigen ist. Im Anschluss daran werden drei Transformationen besprochen, die nicht korrekt sind. Der Abschnitt endet mit einer Übersicht über weitere Transformationen, deren Korrektheit im Rahmen dieser Arbeit nicht abschließend untersucht werden konnte.

4.4.1 Korrekte Transformationen

4.4.1.1 Die „let floating in“-Transformation

Diese Transformation[21] verschiebt `let`-Bindungen nach innen, wobei jedoch keine Bindungen in den Rumpf einer untergeordneten Abstraktion verschoben werden.

Aufgrund der Korrektheit der (flool-let)-, (flool-letrec)-, (flooacs)- und (fai-let)-Transformation folgt, dass `let(rec)`-Bindungen in andere `let(rec)` Bindungen, in das erste Argument eines `case`-Ausdrucks und in Anwendungen verschoben werden können.

Zu zeigen ist nun nur noch, dass Bindungen in die `case`-Alternativen verschoben werden können, was jedoch mithilfe der (brcp)-Reduktion für den FUNDIO-Kalkül leicht bewerkstelligt werden kann.

Anstelle eines vollständigen Beweises zeigen wir die Korrektheit eines Beispiels, dass sowohl in [PS98] als auch in [PPS96] angegeben ist.

Beispiel 4.44. *Seien s und t wie folgt definiert, wobei x nicht frei in z vorkommt.*

$$
\begin{array}{l}
s = \texttt{let}\ x = y + 1\ \texttt{in case}\ z\ \texttt{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{[] ->}\ \ x * x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p{:}ps\ \texttt{->}\ 1 \\
\\
t = \texttt{case}\ z\ \texttt{of} \\
\qquad\quad \texttt{[] ->}\ \ \texttt{let}\ x = y + 1\ \texttt{in}\ x * x \\
\qquad\quad p : ps{-} > 1
\end{array}
$$

Wir zeigen nun, dass $[\![s]\!] \sim_c [\![t]\!]$ gilt:

$$
\begin{array}{rl}
[\![s]\!] & \equiv (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ (\texttt{case}\ [\![z]\!] \\
& \qquad\qquad\qquad\qquad\qquad\qquad ([\![\texttt{[]}]\!] \to [\![x * x]\!]) \\
& \qquad\qquad\qquad\qquad\qquad\qquad ((\![\texttt{(:)}]\!]\ p\ ps) \to [\![1]\!]) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (pat_3 \to \bot) \ldots (pat_N \to \bot))) \\
\xrightarrow{brcp} & (\texttt{case}\ [\![z]\!] \\
& \qquad ([\![\texttt{[]}]\!] \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ [\![x * x]\!])) \\
& \qquad ((\![\texttt{(:)}]\!]\ p\ ps) \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ [\![1]\!])) \\
& \qquad (pat_3 \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ \bot)) \ldots (pat_N \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ \bot))) \\
\sim_c & (\texttt{case}\ [\![z]\!] \\
& \qquad ([\![\texttt{[]}]\!] \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ [\![x * x]\!])) \\
& \qquad ((\![\texttt{(:)}]\!]\ p\ ps) \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ [\![1]\!])) \\
& \qquad (pat_3 \to \bot) \ldots (pat_N \to \bot))
\end{array}
$$

[21] Vgl. [PPS96, Abschnitt 3.1], [San95, Abschnitt 5.1] und [PS98, Abschnitt 7.1].

$$
\begin{aligned}
&\xrightarrow{gc} (\texttt{case}\ [\![z]\!] \\
&\qquad ([\![\texttt{[]}]\!] \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}\ [\![x*x]\!])) \\
&\qquad (([\![\texttt{(:)}]\!]\ p\ ps) \to [\![1]\!]) \\
&\qquad (pat_3 \to \bot) \ldots (pat_N \to \bot)) \\
&\equiv (\texttt{case}\ [\![z]\!] \\
&\qquad ([\![\texttt{[]}]\!] \to (\texttt{letrec}\ x = [\![y+1]\!]\ \texttt{in}[\![x*x]\!])) \\
&\qquad (([\![\texttt{(:)}]\!]\ p\ ps) \to [\![1]\!]) \\
&\qquad (pat_3 \to \bot) \ldots (pat_N \to \bot))
\end{aligned}
$$

4.4.2 Nicht korrekte Transformationen

4.4.2.1 Die „full-laziness"-Transformation

Im Gegenteil zur „let floating in" Transformation, verschiebt die „full-laziness"- oder auch „let floating out"-Transformation[22] Bindungen nach außen. Da Bindungen, die sich im Rumpf einer Abstraktion befinden, auch nach außen verschoben werden, ist die Transformation nicht korrekt, womit sich auch schon in ähnlicher Weise in [Kut00, Abschnitt 5.4.1] für einen nichtdeterministischen Lambda-Kalkül, der dem FUNDIO-Kalkül ähnlich ist, beschäftigt wurde.

Das folgende Beispiel zeigt, dass das Verschieben von Bindungen aus Abstraktionen heraus keine $[\![\cdot]\!]$-korrekte Programmtransformation ist.

Beispiel 4.45. *Wir betrachten die folgenden Ausdrücke s und t, wobei sich t nur dadurch von s unterscheidet, dass die Bindung $z =$* `unsafePerformIO getChar` *aus der Abstraktion hinaus geschoben wurde.*

$$
\begin{aligned}
s \ &= \texttt{let}\ f = \lambda x\ \texttt{->}\ \texttt{let}\ z = \texttt{unsafePerformIO getChar}\ \texttt{in}\ z \\
&\quad\ \texttt{in case}\ f\ \texttt{'a' of}\ y\ \texttt{->}\ f\ \texttt{'b'} \\
t \ &= \texttt{let}\ f = \texttt{let}\ z = \texttt{unsafePerformIO getChar in}\ \lambda x\ \texttt{->}\ z \\
&\quad\ \texttt{in case}\ f\ \texttt{'a' of}\ y\ \texttt{->}\ f\ \texttt{'b'}
\end{aligned}
$$

Die Berechnung im Anhang in Abschnitt A.11 zeigt, dass gilt: $[\![s]\!] \not\sim_c [\![t]\!]$.

Im Anhang in Abschnitt B.2 findet sich ein entsprechendes Programm, dessen Verhalten durch die „full-laziness"-Transformation verändert wird.

4.4.2.2 Die CSE-Transformation

Die in [Chi98] beschriebene „Common Subexpression Elimination"- Transformation (CSE) ersetzt gleiche Unterausdrücke durch eine Variable, wobei der ursprüngliche Ausdruck an die Variable gebunden und durch eine `let`-Bindung geshared wird.

[22] Vgl. [PPS96, Abschnitt 3.2], [San95, Abschnitt 5.2] und [PS98, Abschnitt 7.2].

Durch Inlining und anschließende (dcr)-Transformation kann die Transformation rückgängig gemacht werden. Da Inlining im generellen nicht $[\![\cdot]\!]$-korrekt ist, gilt gleiches für CSE, was folgendes Beispiel zeigt:

Beispiel 4.46. *Wir betrachten die folgenden Ausdrücke s und t, wobei t aus s mit einer CSE-Transformation erhalten werden kann:*

$$
\begin{array}{l}
s \;= \texttt{case unsafePerformIO getChar of} \\
\qquad\quad y \texttt{ -> case unsafePerformIO getChar of} \\
\qquad\qquad\quad y' \texttt{ -> 'a'}
\end{array}
$$

$$
\begin{array}{l}
t \;= \texttt{let } x = \texttt{unsafePerformIO getChar in} \\
\qquad\quad \texttt{case } x \texttt{ of} \\
\qquad\quad\; y \texttt{ -> case } x \texttt{ of} \\
\qquad\qquad\quad y' \texttt{ -> 'a'}
\end{array}
$$

Man kann leicht nachrechnen, dass für $P = \{(\mathcal{B}, a')\}$ gilt: $[\![t]\!]\Downarrow(P)$ und $[\![s]\!]\Uparrow(P)$.

4.4.2.3 Die „static argument"-Transformation

Die in [San95, Abschnitt 7.1] beschriebene Transformation wird im GHC nicht mehr ausgeführt, analog zu den Untersuchungen in [PPRS00] und [PS00] für eine parallele funktionale Programmiersprache, die nichtdeterministisch ist, zeigt sich jedoch schnell, dass die die „static argument"-Transformation nicht $[\![\cdot]\!]$-korrekt ist.

Wir illustrieren dies an einem Beispiel, wobei wir die Berechnung dem Leser überlassen.

Beispiel 4.47. *Seien s und t die folgenden Ausdrücke:*

$$
\begin{array}{l}
s \;= \texttt{let } f = \lambda a\ b \texttt{ -> case unsafePerformIO getChar of} \\
\qquad\qquad\qquad\qquad\quad \texttt{'d' -> } 0 \\
\qquad\qquad\qquad\qquad\quad y \texttt{ -> } f\ a\ b \\
\qquad \texttt{in } f\ 0\ 1
\end{array}
$$

$$
\begin{array}{l}
t \;= \texttt{let } f = \lambda a\ b \texttt{ -> let } f' = \texttt{case unsafePerformIO getChar of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{'d' -> } 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad y \texttt{ -> } f' \\
\qquad\qquad\qquad\qquad\quad \texttt{in } f' \\
\qquad \texttt{in } f\ 0\ 1
\end{array}
$$

s kann durch die „static argument"-Transformation zu t transformiert werden, da die Argumente a und b statisch sind, d.h. sie werden innerhalb der Definition von f nicht verändert und im rekursiven Aufruf von f werden sie an gleicher Position unverändert verwendet.

Bei der Auswertung von t wird ein Zeichen gelesen, wobei dabei die rechte Seite der Bindung für f' mit 0 oder f' überschrieben wird. D.h. bei weiteren rekursiven Aufrufen wird kein Zeichen gelesen.

Bei Auswertung von s findet pro rekursivem Aufruf die Auswertung von `unsafePerformIO getChar` *statt. D.h. die IO-Multimenge* $P = \{(\mathcal{B}, \texttt{'d'}), (\mathcal{B}, \texttt{'e'})\}$ *unterscheidet die Ausdrücke* $[\![s]\!]$ *und* $[\![t]\!]$.

4.4.3 Nicht abschließend untersuchte Transformationen

4.4.3.1 Die Demand-Analyse

Die Demand-Analyse wird durchgeführt, um unter anderem Striktheitsinformation (siehe [PP93]) zu erhalten. Des Weiteren ist die „constructed product result"-Analyse (siehe [BGP]) als Teil der Demand-Analyse implementiert. Anhand der erhaltenen Information kann dann die „Worker/Wrapper"-Transformation (siehe z.B. [PS98]) durchgeführt werden, die als eigener Optimierungspass implementiert ist.

4.4.3.2 Die „UsageSP"-Analyse

Basierend auf [WP99] wird hier ein Typsystem benutzt, mit dem es möglich ist zusätzlich Information darüber zu speichern, wie oft und in welchem Kontext freie Variablen vorkommen. Der Vorteil hierbei ist, dass u.U. auch in Rümpfe von Abstraktionen usw. kopiert werden kann, wenn diese Abstraktionen nur einmal ausgewertet werden. Da wir eine entsprechende Erweiterung der (ucp)-Reduktion für den FUNDIO-Kalkül nicht untersucht haben können wir hier keine Aussage über die Korrektheit dieser Transformation treffen.

4.4.3.3 Deforestation

Die auf [Wad90] basierende Transformation entfernt unnötiges Erzeugen von Datenstrukturen, die nur für Zwischenberechnungen benötigt werden. Ein Standardbeispiel ist der Ausdruck `sum (map double) [1..n]`: Die Liste der Zahlen von 1 bis n wird erzeugt und gleichzeitig durch die Summation wieder abgebaut.

Genauere Details der Implementierung dieser Transformation im GHC finden sich in [Gil96].

4.4.3.4 Specialising

Die in [Jon94] beschriebene Transformation erschafft bei überladenen Operatoren, wie z.B. `(+)`, eigene „spezielle" Funktionen für jeden Typen, um somit so genannte „Dictionary"-Parameter zur Auflösung der Überladung (siehe [WB89]) zu vermeiden.

Als weiterer separater Optimierungspass ist die „specialising over constructors"-Transformation implementiert.

In [PS00] wird „Specialising“ für die dort verwendete nichtdeterministische Sprache als problematisch eingestuft. Diese Ergebnisse lassen sich jedoch nicht einfach auf den FUNDIO-Kalkül übertragen, was wir im Folgenden begründen.

Wir betrachten zunächst das Beispiel aus [PS00, Seite 4], wobei wir die Typisierung jedoch weglassen.

Seien s und t die folgenden Ausdrücke aus $L_{GHCCore}$, wobei t aus s durch „specialising“ erhalten wird. Hierbei sei f eine überladene Funktion der Klasse `MyClass` und die Überladung sei durch hinzufügen des „Dictionary“-Parameters *dict* aufgelöst worden. Des Weiteren sei `dictMyType` das passende Dictionary für den Typ `MyType`, der Instanz der Klasse `MyClass` sei.

$$\begin{array}{rl} s = & \texttt{let } f = \lambda dict \texttt{ -> } e \\ & \texttt{in } f \texttt{ dictMyType } (f \texttt{ dictMyType } y) \end{array} \quad \overset{\text{(specialising)}}{\texttt{===>}} \quad \begin{array}{rl} t = & \texttt{let } f = \lambda dict \texttt{ -> } e \\ & \texttt{in let } f' = (f \texttt{ dictMyType}) \\ & \quad \texttt{in } f' \ (f' \ y) \end{array}$$

In [PS00] wird argumentiert, dass falls e keine WHNF ist und nichtdeterministisch eine Funktion generiert, bei Auswertung von s zweimal generiert wird, während nur eine nichtdeterministische Generierung statt fände, wenn t ausgewertet wird, da dort f' überschrieben wird.

Man könnte für unsere Semantik bezüglich der Übersetzung $[\![\cdot]\!]$ genauso argumentieren, indem man e durch den Ausdruck

$$\begin{array}{l} \texttt{case unsafePerformIO getChar of} \\ \quad y \texttt{ -> } \lambda x \texttt{ -> } x \end{array}$$

ersetzt.

Allerdings haben wir bisher außer Acht gelassen, wie die Definition für f entsteht, denn sie ist durch die Auflösung der Überladung entstanden.

Nehmen wir an die Typklasse `MyClass` hat nur die Klassenfunktion f und „Dictionaries“ für diese Klasse sind durch den einstelligen Konstruktor `MyClassDict` implementiert, dann hat s nach [WB89] folgende Form:

$$\begin{array}{rl} s = & \texttt{letrec } f = \lambda dict \texttt{ -> case } dict \texttt{ of } \{\texttt{MyClassDict } f_1 \texttt{ -> } f_1\}; \\ & \quad \texttt{dictMyType } = \texttt{ MyClassDict } e' \\ & \texttt{in } f \texttt{ dictMyType } (f \texttt{ dictMyType } y) \end{array}$$

Der durch „Specialising“ entstandene Ausdruck t hat die Form:

$$\begin{array}{rl} t = & \texttt{letrec } f = \lambda dict \texttt{ -> case } dict \texttt{ of } \{\texttt{MyClassDict } f_1 \texttt{ -> } f_1\}; \\ & \quad \texttt{dictMyType } = \texttt{ MyClassDict } e'; \\ & \quad f' = f \texttt{ dictMyType} \\ & \texttt{in } f' \ (f' \ y) \end{array}$$

Die Berechnung im Anhang in Abschnitt A.12 zeigt, dass $[\![s]\!] \sim_c [\![t]\!]$ gilt.

4.4.4 Ergebnisse

Für die vorzunehmenden Modifikationen des GHC können wir nun wie folgt vorgehen:

- Die „let floating in“-Transformation kann weiterhin durchgeführt werden, da diese $[\![\cdot]\!]$-korrekt ist.
- Die „full laziness“-Transformation, „static argument transformation“ und „common subexpression elimination“ sollten nicht durchgeführt werden, da diese offensichtlich nicht korrekt bezüglich der FUNDIO-Semantik sind.
- Die restlichen globalen Transformationen wurden nicht abschließend untersucht und sollten deshalb i.A. nicht ausgeführt werden. Wir werden jedoch eine Möglichkeit schaffen, diese für Testzwecke auszuführen, wobei wir dies innerhalb der Programmdokumentation ausreichend kennzeichnen.

Im nächsten Kapitel werden wir die Implementierung der Ergebnisse dieses Kapitels bezüglich der lokalen und globalen Transformationen beschreiben.

Kapitel 5

Implementierung

Im Folgenden werden wesentliche Änderungen beschrieben, die wir an der Implementierung des GHC vorgenommen haben, wobei wir auf Implementierungsdetails weitgehend verzichten und an den entsprechenden Stellen auf den dokumentierten Quellcode[1] verweisen.

Den modifizierten GHC nennen wir HasFuse, wobei dies eine Abkürzung für „Haskell with FUNDIO-based side effects“ ist. Der Quellcode von HasFuse ist im World Wide Web über die Adresse `http://www.ki.informatik.uni-frankfurt.de/~sabel` zu finden.

Wir beschreiben zunächst die Änderungen am Simplifier, d.h. das Abschalten bzw. Modifizieren der lokalen Transformationen. Im Anschluss daran beschreiben wir, welche globalen Transformationen weiterhin ausgeführt werden und welche wir abgeschaltet haben. Das Kapitel endet mit einer Übersicht über die ausgeführten Transformationen in HasFuse und verschiedenen Möglichkeiten der Steuerung der Optimierung.

5.1 Der Simplifier

Der Simplifier bearbeitet das zu optimierende Programm der Kernsprache, indem es den Code entsprechend der Termstruktur zunächst zerlegt, dabei den Kontext auf einem Stack ablegt und abschließend das Programm von unter her wieder zusammen setzt[2].

Dabei finden die Programmtransformationen, wie sie in Kapitel 4 behandelt wurden, auf Teilausdrücken sowohl beim Zerlegen als auch beim Zusammensetzen statt.

Dieses Verfahren wird iteriert, bis keine Transformation mehr anwendbar ist, oder eine maximale Anzahl an Iterationen erreicht ist.

Wir haben den Quellcode des Simplifiers genau untersucht, wobei viele Ergebnisse dieser Untersuchung bereits in Kapitel 4 eingeflossen sind.

[1] Modifizierte Teile des Quellcodes sind mit der Markierung `[DS]` versehen

[2] Einen umfangreichen Überblick über die Funktionsweise des Simplifiers gibt [PM02].

Im Folgenden beschreiben wir die Änderungen, die wir aufgrund der nicht korrekten Transformationen aus Theorem 4.43 vorgenommen haben. Außerdem gehen wir auf die Eta-Reduktion ein, da die Implementierung leicht verändert wurde, so dass sie der (η-red)-Transformation entspricht.

5.1.1 Inlining von speziellen Ausdrücken

5.1.1.1 Vorkommen-Analyse

Jeder Iteration des Simplifiers geht eine „occurrence analyse“[3] voran, die für jede gebundene Variable die Anzahl der freien Vorkommen der selbigen zählt. Diese Information wird beim späteren Inlining benötigt.

Die gewonnene Information wird mithilfe des Datentyps `OccInfo` für jede gebundene Variable gespeichert, der[4] wie folgt definiert ist:

```
data OccInfo = NoOccInfo
             | IAmDead
             | OneOcc InsideLam OneBranch
             | IAmALoopBreaker

type InsideLam = Bool

type OneBranch = Bool
```

Die einzelnen Datenkonstruktoren haben folgende Interpretationen:

- `NoOccInfo` zeigt an, dass keine Information über die Anzahl an freien Vorkommen vorliegt, oder dass mehr als ein freies Vorkommen pro `case`-Zweig existiert.
- `IAmDead` bedeutet, dass die gebundene Variable keinerlei freie Vorkommen besitzt.
- Die Markierung `OneOcc` wird benutzt, wenn die Variable mindestens einmal textuell und höchstens einmal pro `case`-Zweig frei vorkommt. Das Prädikat `InsideLam` ist erfüllt, wenn ein Vorkommen innerhalb eines Rumpfs einer Abstraktion ist. Das `OneBranch`-Prädikat ist wahr, wenn nur ein textuelles Vorkommen existiert.
- Die gebundene Variable wird mit `IAmALoopBreaker` markiert, falls die freien Vorkommen nicht ersetzt werden sollen, da sie eine rekursive Bindungsgruppe „brechen“. Damit wird erreicht, dass der Simplifier terminiert, wobei für genauere Details auf [PM02, Seiten 404-410] verwiesen sei.

[3] engl. für Vorkommen-Analyse
[4] Im Modul `ghc/compiler/basicTypes/BasicTypes.lhs`

In [PM02, Seite 403] wird der Begriff eines „one-shot-lambdas“ definiert, wobei dies solche Abstraktionen sind, die höchstens einmal ausgewertet werden. Vorkommen von Variablen in solchen Abstraktion wurden in der „occurence analyse“ nicht als `InsideLam` markiert. Wir haben die Implementierung an dieser Stelle[5] derart verändert, dass solche Vorkommen nun auch dementsprechend markiert werden. Zum anderen wurde beispielsweise die Variable x im Ausdruck λx `->` λy `->` $\ldots x \ldots$ nicht als `InsideLam` markiert, was wir ebenso änderten.

Somit brauchen wir diese Spezialfälle nicht behandeln und die von der „occurence analyse“ gelieferte Information entspricht exakt dem oben beschriebenen.

Im Folgenden stellen wir dar, wie wir die allgemein nicht $[\![\cdot]\!]$-korrekte „Inlining-Transformation“ im Quellcode derart verändert haben, dass diese nur korrekt ausgeführt wird.

Sei $x = e$ eine `let-` bzw. `letrec`-Bindung. Inlining ersetzt freie Vorkommen der Variablen x durch e, wobei dies auf folgende Weisen geschehen kann:

- *Inlining vor Simplifikation*: Alle freien Vorkommen von x werden durch e ersetzt, bevor e und der Geltungsbereich von x vereinfacht wurden.
- *Inlining beim Aufruf*: Das freie Vorkommen von x wird durch e zu dem Zeitpunkt ersetzt, an dem der Simplifier das freie Vorkommen vereinfachen will.
- *Inlining nach Simplifikation*: Alle freien Vorkommen von x werden durch e ersetzt, nachdem e und der Geltungsbereich von x durch den Simplifier bearbeitet wurden.

In den folgenden Abschnitten betrachten wir diese drei Formen des Inlinings genauer.

5.1.1.2 Inlining vor Simplifikation

Inlining für alle freie Vorkommen von x und anschließendes Löschen der Bindung $x = e$, bevor e oder der Geltungsbereich von x vereinfacht wurde, findet statt, falls das Prädikat `preInlineUnconditionally`[6] für x erfüllt ist, wobei hierfür eine der folgenden Bedingungen gelten muss:

1. x ist mit `IAmDead` markiert, d.h. x hat kein freies Vorkommen im zugehörigen Geltungsbereich.
2. x ist mit `OneOcc False True` markiert, d.h. die Variable hat ein textuelles freies Vorkommen im Geltungsbereich, dass nicht im Rumpf einer Abstraktion ist.

[5] Im Modul `ghc/compiler/SimplCore/OccurAnal.lhs`.

[6] Vgl. [PM02, Seite 417]. Die Definition des Prädikats befindet sich im Modul `ghc/compiler/simplCore/SimplMonad.lhs`.

Inlining ist in diesen Fällen korrekt: Fall 1 ist aufgrund der (dcr)- bzw. (dcr-letrec)-Transformation zulässig. In Fall 2 kann die (uinl)-Transformation angewendet werden. Alle verwendeten Transformationen sind aufgrund von Theorem 4.42 $[\![\cdot]\!]$-korrekt.

5.1.1.3 Inlining beim Aufruf

Das Ersetzen eines freien Vorkommens von x durch e, wird durch die Funktion `callSiteInline`[7] gesteuert, die ein Ergebnis vom Typ `Maybe CoreExpr` liefert. Hierbei wird `Nothing` zurück gegeben, wenn kein Inlining statt finden soll, ansonsten wird der neue Code für x als Ergebnis erhalten.

`callSiteInline` selbst benutzt das Prädikat `exprIsCheap`[8] für e, um das Ergebnis zu berechnen. Für HasFuse haben wir die Implementierung dieses Prädikats derart verändert, dass Ausdrücke, die das Prädikat erfüllen, in der Menge $CHEAP$ liegen, die wir in Definition 4.16 für die (cheapinl)-Transformation definiert haben.

Die Modifikation gegenüber dem GHC besteht darin, dass das Prädikat für weniger Ausdrücke erfüllt ist: Im GHC werden auch `case`-, `let`-Ausdrücke, deren Unterausdrücke das Prädikat erfüllen, und Anwendungen von Variablen als „cheap“ berechnet. Für HasFuse konnten wir diese Definition nicht benutzen, da wir die $[\![\cdot]\!]$-Korrektheit der (cheapinl)-Transformation für solche Ausdrücke nicht gezeigt haben, wobei wir Vermuten das selbiges zumindest in Teilen möglich ist.

Des Weiteren werden im GHC Ausdrücke der Form `error z` als „cheap“ betrachtet. Diese Ausdrücke sind semantisch gleich zu $\bot$. Das Kopieren scheint auch in FUNDIO erlaubt, denn wenn die Auswertung x erreicht, terminiert das Programm nicht und die Bindung $x = e$ wird nie aktualisiert. Steht anstelle von x direkt $\bot$ so terminiert das Programm ebenso nicht. Trotzdem werden in HasFuse solche `error`-Ausdrücke nicht als „cheap“ berechnet, da wir entsprechendes nicht bewiesen haben.

Die Funktion `callSiteInline`[9] liefert nur dann ein Ergebnis der Form `Just` e, wenn eine der folgenden Bedingungen erfüllt ist:

1. x ist mit `OneOcc` markiert, hat somit höchstens ein freies Vorkommen pro `case`-Zweig und
   a) `exprIsCheap` e oder
   b) x ist als `OneOcc False _` markiert, d.h. x kommt nicht im Rumpf einer Abstraktion frei vor.
2. `exprIsCheap` e und x ist mit `NoOccInfo` markiert, d.h. diese Bedingung ist unabhängig von der Anzahl und Art der freien Vorkommen von x.

[7] Vgl. [PM02, Abschnitt 7.1]. Die Funktion `callSiteInline` ist im Modul `ghc/compiler/coreSyn/CoreUnfold.lhs` zu finden.
[8] Das Prädikat ist im Modul `ghc/compiler/coreSyn/CoreUtils.lhs` definiert.
[9] Zur Erfüllung werden weitere Bedingungen benötigt, die wir hier nicht betrachten möchten.

Unter diesen Bedingungen ist Inlining in HasFuse $[\![\cdot]\!]$-korrekt, denn in den Fällen 1.a) oder 2. entspricht die Transformation der (cheapinl)-Transformation und, falls Bedingung 1.b) gilt, der (bruinl)-Transformation. Aufgrund von Theorem 4.42 sind beide Transformationen $[\![\cdot]\!]$-korrekt.

5.1.1.4 Inlining nach Simplifikation

Die Entscheidung darüber, ob alle freien Vorkommen von x durch e ersetzt werden und die Bindung $x = e$ gelöscht wird, nachdem e und der Bindungsbereich durch den Simplifier vereinfacht wurden, ist von der Erfüllung des Prädikats `postInlineUnconditionally`[10] abhängig. Dieses kann nur dann wahr werden, wenn das Prädikat `exprIsTrivial`[11] für e, sowie einige Zusatzbedingungen, die wir nicht betrachten möchten, erfüllt sind.

`exprIsTrivial` ist erfüllt für Konstanten und Variablen sowie für Konstruktoren und primitive Operatoren mit positiver Stelligkeit. Für HasFuse wurde gegenüber dem GHC geändert, dass die positive Stelligkeit für primitive Operatoren explizit geprüft wird. Somit gilt: `exprIsTrivial` $e \implies$ `exprIsCheap` e

Das Ersetzen aller freien Vorkommen von x entspricht mehrmaligem Anwenden der (cheapinl)-Transformation, das abschließende Löschen einer Anwendung der (dcr)- bzw. (dcr-letrec)-Transformation. Aufgrund von Theorem 4.42 ist dieses Vorgehen $[\![\cdot]\!]$-korrekt.

5.1.2 Eta-Reduktion

Die (eta-red)-Transformation ist sehr ähnlich zur Implementierung der Eta-Reduktion im GHC, die mittels der Funktion `tryEtaReduce`[12] durchgeführt wird.

Das Prüfen der Eigenschaften für u (aus der Definition der (eta-red)-Transformation) geschieht dabei mit dem Prädikat `isEvaldUnfolding`[13], welches selbst auf das Prädikat `exprIsValue`[14] zurück greift.

Dieses Prädikat wurde für HasFuse wie folgt verändert:

- Im Gegensatz zu HasFuse ist im GHC das Prädikat auch bei ungesättigten Anwendungen auf primitive Operatoren, sowie bei Anwendungen auf eine Variable, deren Stelligkeit größer als die Anzahl der Argumente ist, erfüllt.

[10] Vgl. [PM02, Seite 417f.]. Das Prädikat wird im Modul `ghc/compiler/simplCore/SimplMonad.lhs` definiert.
[11] Die Implementierung befindet sich im Modul `ghc/compiler/coreSyn/CoreUtils.lhs`. Siehe auch [PM02, Abschnitt 2.2.1].
[12] Im Modul `ghc/compiler/simplCore/SimplUtils.lhs`.
[13] Im Modul `ghc/compiler/coreSyn/CoreSyn.lhs`.
[14] Im Modul `ghc/compiler/coreSyn/CoreUtils.lhs`.

- `exprIsValue` greift bei speziellen Konstruktoranwendungen[15] auf das Prädikat `exprOkForSpeculation`[16] zurück, indem geprüft wird, ob das Prädikat für alle Argumente erfüllt ist.

  Ein Ausdruck erfüllt das Prädikat `exprOkForSpeculation`, wenn es korrekt ist, den Ausdruck auszuwerten, obwohl der Programmkontext dies nicht vorsieht, bzw. keine Auswertung des Ausdrucks durchzuführen, obwohl dies vorgesehen ist.

  Da wir dieses Prädikat nicht bzgl. der FUNDIO-Semantik untersucht haben, haben wir an der entsprechenden Stelle in `exprIsValue` den Aufruf des Prädikats durch `False` ersetzt.

Mit diesen Änderungen entspricht die Eta-Reduktion in HasFuse der (eta-red)-Transformation und diese ist $[\![\cdot]\!]$-korrekt.

5.1.3 Eta-Expansion

Im GHC wird die Eta-Expansion mittels der Funktion `tryEtaExpansion`[17] durchgeführt.

Hierbei wird die Stelligkeit des zu expandierenden Ausdrucks mit der Funktion `exprEtaExpandArity`[18] berechnet. Die so ausgeführte Transformation entspricht den Transformation (η-exp) und (η-exp-case), welche aufgrund von Theorem 4.43 nicht $[\![\cdot]\!]$-korrekt sind.

Für HasFuse wurde die $[\![\cdot]\!]$-korrekte (eeta-exp)-Transformation realisiert, indem die Funktion `exprEtaExpandArity` neu definiert wurde:

```
exprEtaExpandArity = exprArity
```

Hierbei ist `exprArity`[19] eine weitere bereits im GHC implementierte Funktion, die der Funktion ar_η aus Definition 4.39 entspricht, wobei jedoch `exprArity` x, wenn x eine Variable ist, die mit der Variablen abgespeicherte Stelligkeitsinformation zurück gibt. Das Setzen dieser Information erfolgt jedoch ebenso mit der Funktion `exprArity`[20]. Somit entspricht die im HasFuse angewendete Eta-Expansion der um Variablen erweiterten[21] (eeta-exp)-Transformation.

[15] Genauer sind dies Konstruktoranwendungen deren Konstruktoren einen „unlifted“ Typ haben, wobei dies bedeutet, dass der nichtterminierende Ausdruck $\bot$ nicht zur Menge der Ausdrücke dieses Typs gehört, wie dies ansonsten üblich ist.

[16] Dieses wird im Modul `ghc/compiler/coreSyn/CoreUtils.lhs` definiert.

[17] Diese Funktion ist im Modul `ghc/compiler/simplCore/SimplUtils.lhs` definiert

[18] Definiert im Modul `ghc/compiler/coreSyn/CoreUtils.lhs`.

[19] Im Modul `ghc/compiler/coreSyn/CoreUtils.lhs`.

[20] Dies geschieht beispielsweise in der Funktion `completeLazyBind` im Modul `ghc/compiler/simplCore/Simplify.lhs`.

[21] Vgl. hierzu die Bemerkung im Anschluss an den Beweis zu Lemma 4.40.

5.1.4 Case-Elimination

Die „case elimination“ wird von der Funktion `mkCase1`[22] durchgeführt.

Sei

$$\texttt{case}\ e\ \texttt{of}\ y\ \texttt{->}\ E$$

der betrachtete `case`-Ausdruck. Die (ce)-Transformation wird von `mkCase1` in drei Fällen durchgeführt:

- Das Prädikat `exprIsValue` ist für e erfüllt. In diesem Fall wird auch für HasFuse die (ce)-Transformation durchgeführt, da dies aufgrund von Lemma 4.28 $[\![\cdot]\!]$-korrekt ist.

- E ist strikt in e und e ist eine Variable.

  Die (ce)-Transformation entspricht in diesem Fall einer rückwärtigen Anwendung der (ltc)-Transformation, die $[\![\cdot]\!]$-korrekt ist, falls Vermutung 3.74 wahr ist.

  Für HasFuse bleibt dieser Fall unverändert, wobei die Transformation nur ausgeführt werden kann, wenn Striktheitsinformation vorhanden ist, d.h. die Demand-Analyse durchgeführt wurde. Somit kann die Transformation durch Ausschalten der Demand-Analyse ebenfalls ausgeschaltet werden.

- Für e ist das Prädikat `exprOkforSpeculation` erfüllt.

  Wie bereits erwähnt wurde, haben wir dieses Prädikat nicht untersucht. Für HasFuse haben wir die (ce)-Transformation für diesen Fall ausgeschaltet.

5.2 Globale Optimierungen und Optimierungsstufen

In Kapitel 2 haben wir gesehen, dass der GHC mit drei verschiedenen Optimierungsstufen aufgerufen werden kann, wobei je nach Optimierungsstufe unterschiedlich viele globale Transformationen stattfinden.

Wir haben diese drei Optimierungsstufen für HasFuse beibehalten, jedoch die Anzahl der ausgeführten Transformationen derart verändert[23], dass gilt:

- In Optimierungsstufe 0 wird nur der (modifizierte) Simplifier aufgerufen, wobei wie bisher die „case-merging“-Transformation und die Eta-Reduktion nicht durchgeführt werden.

[22] Diese ist im Modul `ghc/compiler/simplCore/simplUtils.lhs` definiert.
[23] Diese Änderungen wurden im Modul `ghc/compiler/Main/DriverState.hs` vorgenommen.

- In Optimierungsstufe 1 werden alle Transformationen ausgeführt, deren $[\![\cdot]\!]$-Korrektheit bewiesen wurde. D.h. zusätzlich zur Optimierung in Stufe 0 wird „case-merging“ und die Eta-Reduktion im Simplifier durchgeführt. Außerdem wird die globale „Floating-In“-Transformation durchgeführt. Transformationen, deren $[\![\cdot]\!]$-Korrektheit von der Gültigkeit von Vermutung 3.74 abhängt, werden i.A.[24] nicht ausgeführt, da die Striktheitsanalyse (als Teil der Demand-Analyse) ausgeschaltet ist.

  Wir haben für HasFuse jedoch das Flag `-fstrictness` hinzugefügt, mit dem die Striktheitsanalyse angeschaltet werden kann.

- In Optimierungsstufe 2 werden zusätzlich zu Stufe 1 einige Transformationen durchgeführt, deren Korrektheit nicht abschließend untersucht wurde.

  Diese Optimierungsstufe sollte nur zu Testzwecken benutzt werden, da sie keine völlige Sicherheit garantiert. Bisher ist uns jedoch kein Programm bekannt, dass nach Compilierung in Stufe 2 ein anderes (bzw. nicht korrektes) Verhalten als nach Compilierung in Stufe 1 aufweist.

Die nicht $[\![\cdot]\!]$-korrekten globalen Transformationen („full-laziness“, „common subexpression elimination“ und „static argument“-Transformation) werden nie ausgeführt und können auch nicht durch andere Optionen angeschaltet werden. Selbiges gilt für RULES-Pragmas[25], deren Korrektheit von der jeweiligen Regel abhängt.

Die Tabellen 5.1 und 5.2 geben einen genauen Überblick über die ausgeführten Transformationen je nach Optimierungsstufe, des Weiteren finden sich in den Tabellen Angaben darüber, wie einzelne Transformationen an- oder abgeschaltet werden können.

[24] Die (ci)-Transformation wird stets ausgeführt, da die Transformation unabhängig von Striktheitsinformation ist. Vermutung 3.74 wurde im Beweis der (ci)-Transformation ausschließlich dafür benutzt, diesen kurz zu halten. Ein Beweis ohne Benutzen dieser Vermutung scheint möglich, wie in Abschnitt 4.3.3.8 begründet wurde.

[25] Siehe Abschnitt 2.3.3.

Tranformation	`-O`-**Stufe** 0	1	2	**korrekt**[a]	**Kommentare**
Eta-Expansion	✓	✓	✓	✓	kann mit -fno-do-lambda-eta-expansion abgeschaltet werden
let-to-case	×	✓	✓	✓[b]	
case-merging	×	✓	✓	✓	kann mit -fcase-merge an-, mit -fno-case-merge abgeschaltet werden
Eta-Reduktion	×	✓	✓	✓	kann mit -fdo-eta-reduction an-, mit f-no-do-eta-reduction abgeschaltet werden
RULES-Pragmas	×	×	×	?[c]	
Interface-Pragmas	×	×	×	?[d]	kann mit -fno-ignore-interface-pragmas angeschaltet werden, führt aber möglicherweise zu Fehlern. Zudem sind RULES-Pragmas völlig abgeschaltet.

[a] bzgl. der Übersetzung $[\![\cdot]\!]$
[b] wenn Vermutung 3.74 wahr ist
[c] abhängig vom Pragma
[d] abhängig vom Pragma

Tabelle 5.1: Ausgeführte lokale Transformationen je nach Optimierungsstufe

Tranformation	`-O`-Stufe 0	1	2	**korrekt**[a]	**Kommentare**
Full-laziness	×	×	×	×	
CSE	×	×	×	×	
Let Floating In	×	✓	✓	✓	
Striktheitsanalyse	×	×	✓	?	kann mit -fstrictness angeschaltet werden.
CPR-Analyse	×	×	✓	?	wird nur ausgeführt, wenn die Striktheitsanalyse ausgeführt wird, kann mit -fno-cpr-off angeschaltet, mit -fcpr-off ausgeschaltet werden
Worker-Wrapper	×	×	✓	?	
Specialising	×	×	✓	?	
Specialising over constructors	×	×	✓	?	
Deforestation	×	×	✓	?	kann mit -ffoldr-build-on teilweise angeschaltet, mit -fno-foldr-build-on ausgeschaltet werden.
UsageSP-Analyse	×	×	×	?	kann mit -fusagesp in Optimierungsstufe 2 angeschaltet werden.

[a] bzgl. der Übersetzung $[\![\cdot]\!]$

Tabelle 5.2: Ausgeführte globale Transformationen je nach Optimierungsstufe

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Ziel dieser Arbeit war es, einen Compiler für Haskell derart zu verändern, dass er Programme, die `unsafePerformIO` verwenden, bezüglich der FUNDIO-Semantik korrekt compiliert. Hierfür wurde der GHC als Compiler benutzt, der modifizierte Compiler trägt den Namen HasFuse.

In *Kapitel 2* wurden zunächst die Grundlagen des Funktionalen Programmierens dargestellt und eine kurze Übersicht über die Programmiersprache Haskell gegeben. Anschließend wurden Erweiterungen von Haskell, die der GHC zur Verfügung stellt, sowie der GHC selbst dargestellt. An dieser Stelle haben wir die Kernsprache des GHC definiert, da auf dieser viele Programmtransformationen durchgeführt werden, die später untersucht wurden. Das Kapitel endet mit einer Untersuchung der bisherigen Realisierung von IO in Haskell und im GHC, wobei wir anhand eines Beispiels die Problematik von `unsafePerformIO` und Programmtransformationen demonstrierten.

Da die im GHC verwendeten Transformationen auf Korrektheit bezüglich der FUNDIO-Semantik geprüft werden sollten, wurde in *Kapitel 3* zunächst der entsprechende Kalkül einschließlich der Normalordnungsreduktion und zusätzlicher, bereits als korrekt bewiesene, Programmtransformationen definiert. Im Anschluss haben wir einige neue Transformationen definiert, die wir analog zu [Sch03a] mithilfe von vollständigen Sätzen an Vertauschungs- und Gabeldiagrammen als korrekt bewiesen haben, eine Ausnahme hierbei bildet die (streval)-Regel, deren Beweis als zu aufwendig innerhalb dieser Arbeit erschien.

Mithilfe der nun umfangreichen Menge an Programmtransformationen für den FUNDIO-Kalkül beschäftigten wir uns in *Kapitel 4* mit den Programmtransformationen, die im GHC durchgeführt werden. Hierfür wurde zunächst ein adäquate Übersetzung definiert, die Ausdrücke der GHC-Kernsprache in Ausdrücke des FUNDIO-Kalküls übersetzt. Im Anschluss daran fand eine ausgiebige Untersuchung der lokalen Transformationen, die im GHC angewendet werden, statt, wobei diese entweder – mithilfe der Transformationen für den FUNDIO-Kalkül aus Kapitel 3 – als korrekt, oder durch Gegenbeispiele als nicht korrekt bewiesen wurden. Bei nicht

korrekten Transformationen wurden Spezialfälle betrachtet und gezeigt, dass diese korrekt sind.

Da die Untersuchung der lokalen Transformationen bereits sehr umfangreich war, wurden die globalen Transformationen nicht ausführlich betrachtet und nur kurz dargestellt.

Für die in *Kapitel 5* beschriebene Implementierung war dies jedoch unproblematisch, da die entsprechenden Transformationen relativ leicht abgeschaltet werden konnten. Des Weiteren wurden die lokalen Transformationen anhand der Ergebnisse aus Kapitel 4 modifiziert.

Insgesamt arbeitet der Compiler (zumindest in den von uns untersuchten Teilen) korrekt bzgl. der FUNDIO-Semantik, allerdings ist die Performance der übersetzten Programme verbesserungswürdig, da die mächtige Optimierung des GHC im HasFuse nicht vollständig durchgeführt wird.

6.2 Ausblick

Abschließend zeigen wir einige Erweiterungen und Verbesserungen auf, die durch zukünftige Forschung erreicht werden könnten.

6.2.1 Anwendbarkeit von nichtdeterministischem IO

Zum einen können nun mithilfe der festgelegten Semantik und dem HasFuse die bereits existierenden Anwendungen von `unsafePerformIO` dahin gehend untersucht werden, ob sie bezüglich der Semantik sinnvoll sind, d.h. ob das vom Programmierer erwartete Verhalten der FUNDIO-Semantik bzgl. der Übersetzung $[\![\cdot]\!]$ entspricht.

Zum anderen ergeben sich durch die festgelegte Semantik, die die Verwendung von `unsafePerformIO` nicht mehr, wie bisher, auf Spezialfälle einschränkt, neue Möglichkeiten der Programmierung von Seiteneffekten. Hierbei ist vorstellbar, dass Programme gänzlich mit direktem IO auskommen, d.h. die monadische Verpackung nicht mehr notwendig ist. Hierzu wäre es wünschenswert die Praxistauglichkeit von nicht-striktem direkt-aufgerufenem IO durch größere Anwendungen zu testen und eventuell, wie bereits in [Sch03a] angesprochen, entsprechende Sequentialisierungsmechanismen, aber auch Operatoren zur parallelen oder konkurrenten Auswertung zur Verfügung zu stellen und diese im Umfeld von `unsafePerformIO` zu untersuchen.

6.2.2 Erweiterungen des FUNDIO

Würde der FUNDIO-Kalkül getypt, so würde die Sprache der Kernsprache des GHC mehr entsprechen und eventuell könnte das Problem gelöst werden, dass

`unsafePerformIO` dazu benutzt werden kann, das Typsystem von Haskell auszuhebeln[1], indem nur monomorph getypte `IO`-Konstrukte für jedes Paar von Typen zugelassen werden.

Des Weiteren ist es sinnvoll, Vermutung 3.74 zu beweisen, um somit Korrektheit der bisher unter dieser Annahme gezeigten Transformationen vollends zu garantieren.

6.2.3 Überprüfung weiterer Compilerphasen

Wir haben innerhalb dieser Arbeit den Schwerpunkt unserer Untersuchungen auf die Überprüfung der Optimierungen auf der Kernsprache des GHC gelegt. Hierbei haben wir angenommen, dass das Back-End, also insbesondere die STG-Maschine in Normalordnung reduziert, und dabei Sharing beachtet. Diese Annahme sollte durch weitere Untersuchungen belegt werden. Im Anhang in Abschnitt B.3 ist ein Beispiel angegeben, für das der GHC ein merkwürdiges Übersetzungsverhalten aufweist, wobei wir annehmen, dass dieses Verhalten durch das Back-End verursacht wird.

Ebenso könnte man sich mit dem Front-End des Compilers beschäftigen, und die Übersetzung von Haskell zur Compiler-internen Kernsprache untersuchen.

6.2.4 Steigerung der Effizienz

Da durch die Modifikation des GHC einige möglicherweise für die Effizienz der übersetzten Programme wichtige Transformationen abgeschaltet wurden, könnten weitere Untersuchungen bzgl. der Optimierungstransformationen sinnvoll sein.

Zum einen sollte eine Überprüfung auf Korrektheit der globalen Transformationen durchgeführt werden, wobei die Striktheitsanalyse mit anschließender Worker/Wrapper-Transformation besonders wichtig erscheint. Hierzu ist gleichzeitig ein Beweis von Vermutung 3.74 notwendig, damit die Ergebnisse der Analyse entsprechend verwendet werden können.

Zum anderen könnten die als falsch erwiesenen Transformationen „full-laziness“ und „common subexpression elimination“ möglicherweise abgeändert wieder verwendet werden. Hierfür könnten die Ergebnisse aus [Kut00] bezüglich der „Deterministischen Subausdrücke“ dienen, die jedoch zunächst auf den FUNDIO-Kalkül übertragen werden müssen. Diese Ergebnisse könnten auch bei der lokalen Transformation des „Inlinings“ verwendet werden.

Ebenso können evtl. neue Programmtransformationen entwickelt werden, die speziell für die FUNDIO-Semantik zugeschnitten sind.

Für alle Betrachtungen bzgl. der Optimierung scheint es jedoch notwendig, den Effekt jeder einzelnen Transformation zu messen. Eine entsprechende Benchmark-Suite existiert bereits in Form der „nofib-Suite“ ([Par93]) für Haskell-Programme, diese sollte

[1] Vgl. hierzu Abschnitt 2.4.2.

jedoch um Programme, die nichtdeterministisches IO benutzen, erweitert werden. Hierbei ist auch vorstellbar Programme mit monadischem und nichtmonadischem IO untereinander zu vergleichen, um somit möglicherweise Effizienzvorteile für die eine oder die andere Programmiertechnik heraus zu arbeiten.

Anhang A

Umfangreichere Berechnungen

In diesem Anhang geben wir einige größere Berechnungen an, die in Beweisen dieser Arbeit benötigt wurden.

A.1 Berechnung zu Lemma 3.47

A.1.1 Beispiel zu Fall VI.

$$
\begin{array}{rl}
 & (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= s)\\
 & \qquad y_1 = R_1^-[(\texttt{case}\ x_m\ \ldots((c_i\ \overrightarrow{z}) \to C[x_m])\ldots)], \ldots, y_j = R_j^-[y_j]\\
 & \texttt{in}\ R_0^-[y_j])\\
\xrightarrow{iR,ccpcx} & (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= t)\\
 & \qquad y_1 = R_1^-[(\texttt{case}\ x_m\ \ldots((c_i\ \overrightarrow{z}) \to C[(c_i\ \overrightarrow{z})])\ldots)], \ldots, y_j = R_j^-[y_j]\\
 & \texttt{in}\ R_0^-[y_j])\\
\xrightarrow{n,case-e} & (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= r_1)\\
 & \qquad y_1 = R_1^-[(\texttt{letrec}\ \overrightarrow{z} = \overrightarrow{u}\ \texttt{in} C[(c_i\ \overrightarrow{z})])], \ldots, y_j = R_j^-[y_j]\\
 & \texttt{in}\ R_0^-[y_j])\\
\xrightarrow{n,(lll)^*} & (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= r)\\
 & \qquad \overrightarrow{z} = \overrightarrow{u}, y_1 = R_1^-[C[(c_i\ \overrightarrow{z})]], \ldots, y_j = R_j^-[y_j]\\
 & \texttt{in}\ R_0^-[y_j])\\
\hline
 & (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{a}), x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= s)\\
 & \qquad y_1 = R_1^-[(\texttt{case}\ x_m\ \ldots((c_i\ \overrightarrow{z}) \to C[x_m])\ldots)], \ldots, y_j = R_j^-[y_j]\\
 & \texttt{in}\ R_0^-[y_j])\\
\xrightarrow{n,case-e} & (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},\\
 & \qquad y_1 = R_1^-[(\texttt{letrec}\ \overrightarrow{z} = \overrightarrow{u}\ \texttt{in}\ C[x_m])], \ldots, y_j = R_j^-[y_j]\\
 & \texttt{in}\ R_0^-[y_j])
\end{array}
$$

$$\xrightarrow{n,(lll)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{z} = \overrightarrow{u}, y_1 = R_1^-[C[x_m]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,(cpx)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{z} = \overrightarrow{u}, y_1 = R_1^-[C[x_1]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,cpcx} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{v}), \overrightarrow{v} = \overrightarrow{u}, \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{z} = \overrightarrow{u}, y_1 = R_1^-[C[c_i\ \overrightarrow{v}]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,(cpx)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{v}), \overrightarrow{v} = \overrightarrow{u}, \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{z} = \overrightarrow{u}, y_1 = R_1^-[C[c_i\ \overrightarrow{u}]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,(xch)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{v}), \overrightarrow{v} = \overrightarrow{u}, \overrightarrow{z} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{u} = \overrightarrow{z}, y_1 = R_1^-[C[c_i\ \overrightarrow{u}]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,(cpx)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{v}), \overrightarrow{v} = \overrightarrow{u}, \overrightarrow{z} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{u} = \overrightarrow{z}, y_1 = R_1^-[C[c_i\ \overrightarrow{z}]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,(cpx)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{v} = \overrightarrow{u}, \overrightarrow{z} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{u} = \overrightarrow{z}, y_1 = R_1^-[C[c_i\ \overrightarrow{z}]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,(xch)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{v} = \overrightarrow{u}, \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1},$$
$$\overrightarrow{z} = \overrightarrow{u}, y_1 = R_1^-[C[c_i\ \overrightarrow{z}]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$
$$\xrightarrow{iR,(gc)^*} (\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{u}), \overrightarrow{u} = \overrightarrow{a}, x_2 = x_1, \ldots, x_m = x_{m-1}, \qquad (= r)$$
$$\overrightarrow{z} = \overrightarrow{u}, y_1 = R_1^-[C[c_i\ \overrightarrow{z}]], \ldots, y_j = R_j^-[y_j]$$
$$\texttt{in}\ R_0^-[y_j])$$

Die Existenz der $(n, (lll)^*)$-Reduktionen ergibt sich aus Lemma 3.39.

A.2 Berechnung zu Lemma 3.53

A.2.1 Beispiel zu Fall I.

$$
\begin{array}{rll}
 & (\texttt{letrec } x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= s) \\
 & \quad y_1 = R_B^-[(\texttt{case } (\texttt{case } x_m \ (p_1 \to t_1) \ldots) \ \mathit{Alts})], & \\
 & \quad y_2 = R_2^-[y_1] \ldots, y_j = R_j^-[y_{j-1}] & \\
 & \texttt{in } R_C^-[y_j]) & \\
\xrightarrow{iR,ccase} & (\texttt{letrec } x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= t) \\
 & \quad y_1 = R_B^-[(\texttt{case } x_m \ (p_1 \to (\texttt{case } t_1 \ \mathit{Alts})) \ldots)], & \\
 & \quad y_2 = R_2^-[y_1] \ldots, y_j = R_j^-[y_{j-1}] & \\
 & \texttt{in } R_C^-[y_j]) & \\
\xrightarrow{n,cp-e} & (\texttt{letrec } x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= r) \\
 & \quad y_1 = R_B^-[(\texttt{case } (\lambda v.w) \ (p_1 \to (\texttt{case } t_1 \ \mathit{Alts})) \ldots)], & \\
 & \quad y_2 = R_2^-[y_1] \ldots, y_j = R_j^-[y_{j-1}] & \\
 & \texttt{in } R_C^-[y_j]) & \\
\hline
 & (\texttt{letrec } x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= s) \\
 & \quad y_1 = R_B^-[(\texttt{case } (\texttt{case } x_m \ (p_1 \to t_1) \ldots) \ \mathit{Alts})], & \\
 & \quad y_2 = R_2^-[y_1] \ldots, y_j = R_j^-[y_{j-1}] & \\
 & \texttt{in } R_C^-[y_j]) & \\
\xrightarrow{n,cp-e} & (\texttt{letrec } x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= r') \\
 & \quad y_1 = R_B^-[(\texttt{case } (\texttt{case } (\lambda v.w) \ (p_1 \to t_1) \ldots) \ \mathit{Alts})], & \\
 & \quad y_2 = R_2^-[y_1] \ldots, y_j = R_j^-[y_{j-1}] & \\
 & \texttt{in } R_C^-[y_j]) & \\
\xrightarrow{iR,ccase} & (\texttt{letrec } x_1 = (\lambda v.w), x_2 = x_1, \ldots, x_m = x_{m-1}, & (= r) \\
 & \quad y_1 = R_B^-[(\texttt{case } (\lambda v.w) \ (p_1 \to (\texttt{case } t_1 \ \mathit{Alts})) \ldots)], & \\
 & \quad y_2 = R_2^-[y_1] \ldots, y_j = R_j^-[y_{j-1}] & \\
 & \texttt{in } R_C^-[y_j]) &
\end{array}
$$

A.3 Berechnung zu Beispiel 4.2

$$
\begin{array}{l}
\left[\!\!\left[\left(\begin{array}{l} \lambda i \texttt{ -> case } i \texttt{ of} \\ \qquad (IO\ m) \texttt{ -> case } m \text{ realWorld\# } \texttt{of} \\ \qquad\qquad (s, r) \texttt{ -> } r \end{array} \right) \text{ getChar} \right]\!\!\right] \\
\equiv \left(\left[\!\!\left[\left(\begin{array}{l} \lambda i \texttt{ -> case } i \texttt{ of} \\ \qquad (IO\ m) \texttt{ -> case } m \text{ realWorld\# } \texttt{of} \\ \qquad\qquad (s, r) \texttt{ -> } r \end{array} \right) \right]\!\!\right] \ [\![\text{getChar}]\!] \right)
\end{array}
$$

$$
\begin{aligned}
&\equiv ((\lambda i.(\texttt{case}\ i\ (([\![IO]\!]\ m) \to (\texttt{case}\ (m\ [\![\text{realWorld\#}]\!])\ ((s,r) \to r)\ \ldots))\ \ldots))\\
&\quad ([\![IO]\!]\ (\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots))))\\
&\xrightarrow{lbeta} (\texttt{letrec}\ i = ([\![IO]\!]\ (\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots)))\\
&\quad \texttt{in}\ (\texttt{case}\ i\ (([\![IO]\!]\ m) \to (\texttt{case}\ (m\ [\![\text{realWorld\#}]\!])\ ((s,r) \to r)\ \ldots))\ \ldots))\\
&\xrightarrow{case}\xrightarrow{lll} (\texttt{letrec}\ i = ([\![IO]\!]\ y),\\
&\qquad\qquad y = (\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots)),\\
&\qquad\qquad m = y\\
&\quad \texttt{in}\ (\texttt{case}\ (m\ [\![\text{realWorld\#}]\!])\ ((s,r) \to r)\ \ldots))\\
&\xrightarrow{cp} (\texttt{letrec}\ i = ([\![IO]\!]\ y),\\
&\qquad\qquad y = (\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots)),\\
&\qquad\qquad m = y\\
&\quad \texttt{in}\ (\texttt{case}\ ((\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots))\\
&\qquad\qquad [\![\text{realWorld\#}]\!])\\
&\qquad\quad ((s,r) \to r)\ \ldots))\\
&\xrightarrow{(lll)^*} (\texttt{letrec}\ i = ([\![IO]\!]\ y),\\
&\qquad\qquad y = (\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots)),\\
&\qquad\qquad m = y, w = [\![\text{realWorld\#}]\!]\\
&\quad \texttt{in}\ (\texttt{case}\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots)\\
&\qquad\quad ((s,r) \to r)\ \ldots))\\
&\xrightarrow{(gc)^*} (\texttt{letrec}\ w = [\![\text{realWorld\#}]\!]\\
&\quad \texttt{in}\ (\texttt{case}\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w,p_1)) \ldots (p_n \to (w,p_n))\ (p_{n+1} \to \bot) \ldots)\\
&\qquad\quad ((s,r) \to r)\ \ldots))\\
&\xrightarrow{ccase} (\texttt{letrec}\ w = [\![\text{realWorld\#}]\!]\ \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (\texttt{case}\ (w,p_1)\ ((s,r) \to r)\ \ldots))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_n \to (\texttt{case}\ (w,p_n)\ ((s,r) \to r)\ \ldots))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_{n+1} \to (\texttt{case}\ \bot\ ((s,r) \to r)\ \ldots))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_N \to (\texttt{case}\ \bot\ ((s,r) \to r)\ \ldots))))\\
&\sim_c (\texttt{letrec}\ w = [\![\text{realWorld\#}]\!]\ \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (\texttt{case}\ (w,p_1)\ ((s,r) \to r)\ \ldots))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_n \to (\texttt{case}\ (w,p_n)\ ((s,r) \to r)\ \ldots))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))\\
&\xrightarrow{case} (\texttt{letrec}\ w = [\![\text{realWorld\#}]\!]\ \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (\texttt{letrec}\ s = w, r = p_1\ \texttt{in}\ r))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_n \to (\texttt{letrec}\ s = w, r = p_n\ \texttt{in}\ r))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))\\
&\xrightarrow{ucp} (\texttt{letrec}\ w = [\![\text{realWorld\#}]\!]\ \texttt{in}\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (\texttt{letrec}\ s = w, r = p_1\ \texttt{in}\ p_1))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_n \to (\texttt{letrec}\ s = w, r = p_n\ \texttt{in}\ p_n))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))
\end{aligned}
$$

$\xrightarrow{(gc)^*}$ (case (IO $\mathcal{B}$) $(p_1 \to p_1) \ldots (p_n \to p_n)$ $(p_{n+1} \to \bot) \ldots (p_N \to \bot)$)

A.4 Berechnung zu Beispiel 4.3

$$
\begin{aligned}
&\left[\!\!\left[\lambda c \texttt{ -> } \left(\begin{array}{c} \lambda i \texttt{ -> case } i \texttt{ of} \\ (IO\ m) \texttt{ -> case } m \text{ realWorld\# } \texttt{of} \\ (s, r) \texttt{ -> } r \end{array} \right) \text{ (putChar } c) \right]\!\!\right] \\
&\equiv (\lambda c.((\lambda i.(\texttt{case } i\ ((\llbracket IO \rrbracket\ m) \to (\texttt{case } (m\ \llbracket \text{realWorld\#} \rrbracket)\ ((s,r) \to r)\ \ldots))\ \ldots)) \\
&\qquad ((\lambda x.(\llbracket IO \rrbracket\ (\lambda w.(\texttt{case } x \\
&\qquad\qquad (p_1 \to (\texttt{case } (\texttt{IO } x)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket))))) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (p_n \to (\texttt{case } (\texttt{IO } x)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
&\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))))\ c))) \\
&\xrightarrow{betavar} (\lambda c.((\lambda i.(\texttt{case } i\ ((\llbracket IO \rrbracket\ m) \to (\texttt{case } (m\ \llbracket \text{realWorld\#} \rrbracket)\ ((s,r) \to r)\ \ldots))\ \ldots)) \\
&\qquad (\llbracket IO \rrbracket\ (\lambda w.(\texttt{case } c \\
&\qquad\qquad (p_1 \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (p_n \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
&\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))))) \\
&\xrightarrow{lbeta} (\lambda c.(\texttt{letrec } i = (\llbracket IO \rrbracket\ (\lambda w.(\texttt{case } c \\
&\qquad\qquad\qquad (p_1 \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
&\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad (p_n \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
&\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))) \\
&\qquad \texttt{in } (\texttt{case } i\ ((\llbracket IO \rrbracket\ m) \to (\texttt{case } (m\ \llbracket \text{realWorld\#} \rrbracket)\ ((s,r) \to r)\ \ldots))\ \ldots))) \\
&\xrightarrow{case} (\lambda c.(\texttt{letrec } i = (\llbracket IO \rrbracket\ m'), \\
&\qquad\qquad m' = (\lambda w.(\texttt{case } c \\
&\qquad\qquad\qquad (p_1 \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
&\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad (p_n \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
&\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))) \\
&\qquad\qquad \texttt{in } (\texttt{letrec } m = m' \texttt{ in } (\texttt{case } (m\ \llbracket \text{realWorld\#} \rrbracket)\ ((s,r) \to r)\ \ldots)))
\end{aligned}
$$

$$
\begin{array}{l}
\xrightarrow{llet} (\lambda c.(\texttt{letrec}\ i = (\llbracket IO \rrbracket\ m'), \\
\qquad\qquad m' = (\lambda w.(\texttt{case}\ c \\
\qquad\qquad\qquad\qquad (p_1 \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad (p_n \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (w, \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w, \llbracket \texttt{()} \rrbracket)))) \\
\qquad\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))), \\
\qquad\qquad m = m' \\
\qquad \texttt{in}\ (\texttt{case}\ (m\ \llbracket \text{realWorld\#} \rrbracket)\ ((s, r) \to r)\ \ldots)) \\
\xrightarrow{cp} \ldots \\
\xrightarrow{(lll)^*} \ldots \\
\xrightarrow{(gc)^*} (\lambda c.(\texttt{letrec}\ w' = \llbracket \text{realWorld\#} \rrbracket \\
\qquad \texttt{in}\ (\texttt{case}\ (\texttt{case}\ c \\
\qquad\qquad\qquad (p_1 \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (w', \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w', \llbracket \texttt{()} \rrbracket)))) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad (p_n \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (w', \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w', \llbracket \texttt{()} \rrbracket)))) \\
\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)) \\
\qquad\qquad ((s, r) \to r)\ \ldots))) \\
\xrightarrow{ccase} (\lambda c.(\texttt{letrec}\ w' = \llbracket \text{realWorld\#} \rrbracket \\
\qquad \texttt{in}\ (\texttt{case}\ c \\
\qquad\qquad (p_1 \to (\texttt{case}\ (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (w', \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w', \llbracket \texttt{()} \rrbracket))) \\
\qquad\qquad\qquad\qquad ((s, r) \to r)\ \ldots)) \\
\qquad\qquad \ldots \\
\qquad\qquad (p_n \to (\texttt{case}\ (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (w', \llbracket \texttt{()} \rrbracket)) \ldots (p_N \to (w', \llbracket \texttt{()} \rrbracket))) \\
\qquad\qquad\qquad\qquad ((s, r) \to r)\ \ldots)) \\
\qquad\qquad (p_{n+1} \to (\texttt{case}\ \bot\ \ldots)) \ldots (p_N \to (\texttt{case}\ \bot\ \ldots))))) \\
\\
\xrightarrow{(ccase)^*} (\lambda c.(\texttt{letrec}\ w' = \llbracket \text{realWorld\#} \rrbracket \\
\qquad \texttt{in}\ (\texttt{case}\ c \\
\qquad\qquad (p_1 \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (\texttt{case}\ (w', \llbracket \texttt{()} \rrbracket)\ ((s, r) \to r)\ \ldots)) \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad (p_N \to (\texttt{case}\ (w', \llbracket \texttt{()} \rrbracket)\ ((s, r) \to r)\ \ldots)))) \\
\qquad\qquad \ldots \\
\qquad\qquad (p_n \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to (\texttt{case}\ (w', \llbracket \texttt{()} \rrbracket)\ ((s, r) \to r)\ \ldots)) \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad (p_N \to (\texttt{case}\ (w', \llbracket \texttt{()} \rrbracket)\ ((s, r) \to r)\ \ldots)))) \\
\qquad\qquad (p_{n+1} \to (\texttt{case}\ \bot\ \ldots)) \ldots (p_N \to (\texttt{case}\ \bot\ \ldots)))))
\end{array}
$$

$$
\begin{aligned}
&\xrightarrow{(case)^*} \ldots \\
&\xrightarrow{(cpcx)^*} \ldots \\
&\xrightarrow{(gc)^*} (\lambda c.(\texttt{case}\ c\ (p_1 \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to ([\![\texttt{()}]\!]))\ldots\ (p_N \to ([\![\texttt{()}]\!])))) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (p_n \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to ([\![\texttt{()}]\!]))\ldots\ (p_N \to ([\![\texttt{()}]\!])))) \\
&\qquad\qquad (p_{n+1} \to (\texttt{case}\ \bot\ \ldots))\ldots(p_N \to (\texttt{case}\ \bot\ \ldots))))\\
&\\
&\sim_c (\lambda c.(\texttt{case}\ c\ (p_1 \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to ([\![\texttt{()}]\!]))\ldots\ (p_N \to ([\![\texttt{()}]\!])))) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (p_n \to (\texttt{case}\ (\texttt{IO}\ c)\ (p_1 \to ([\![\texttt{()}]\!]))\ldots\ (p_N \to ([\![\texttt{()}]\!])))) \\
&\qquad\qquad (p_{n+1} \to \bot)\ldots(p_N \to \bot)))
\end{aligned}
$$

A.5 Berechnung zu Lemma 4.15

Wir müssen noch die Fälle betrachten, in denen die `case`-Alternativen eine default-Alternative enthalten. Hierbei ist zusätzlich zu unterscheiden, ob $C'[x]$ die rechte Seite der default-Alternative oder einer anderen Alternative ist.

Betrachten wir zunächst den Fall, dass $C'[x]$ nicht die rechte Seite der default-Alternative ist: Seien $pat_{n+1}, \ldots pat_m$ die Pattern, die durch die Übersetzung der default-Alternative neu entstehen, und $pat_{m+1}, \ldots pat_N$ die restlichen durch die Übersetzung entstandenen Alternativen.

$$
\begin{aligned}
&[\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[\texttt{case}\ e_1\ \texttt{of}\ \{P_1\ \texttt{->}\ B_1; \ldots; P_i\ \texttt{->}\ C'[x] \ldots; P_n\ \texttt{->}\ B_n; y\ \texttt{->}\ B_y\}]\!] \\
\equiv\ &(\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][(\texttt{case}\ [\![e_1]\!] \\
&\qquad\qquad ([\![P_1]\!] \to [\![B_1]\!])\ldots([\![P_i]\!] \to [\![C']\!][x])\ldots([\![P_n]\!] \to [\![B_n]\!]) \\
&\qquad\qquad (pat_{n+1} \to [\![B_y]\!])\ldots(pat_m \to [\![B_y]\!])] \\
&\qquad\qquad (pat_{m+1} \to \bot)\ldots(pat_N \to \bot))]) \\
\xrightarrow{ucpb}\ &(\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][(\texttt{case}\ [\![e_1]\!] \\
&\qquad\qquad ([\![P_1]\!] \to [\![B_1]\!])\ldots([\![P_i]\!] \to [\![C']\!][e])\ldots([\![P_n]\!] \to [\![B_n]\!]) \\
&\qquad\qquad (pat_{n+1} \to [\![B_y]\!])\ldots(pat_m \to [\![B_y]\!])] \\
&\qquad\qquad (pat_{m+1} \to \bot)\ldots(pat_N \to \bot))]) \\
\equiv\ &[\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[\texttt{case}\ e_1\ \texttt{of}\ \{P_1\ \texttt{->}\ B_1; \ldots; P_i\ \texttt{->}\ C'[e]; \ldots; P_n\ \texttt{->}\ B_n; y\ \texttt{->}\ B_y\}]\!]
\end{aligned}
$$

Nun betrachten wir den Fall, dass $C'[x]$ die rechte Seite der default-Alternative ist:

$$
\begin{aligned}
&[\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[\texttt{case}\ e_1\ \texttt{of}\ \{P_1\ \texttt{->}\ B_1; \ldots; P_n\ \texttt{->}\ B_n; y\ \texttt{->}\ C'[x]\}]\!] \\
\equiv\ &(\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][(\texttt{case}\ [\![e_1]\!] \\
&\qquad ([\![P_1]\!] \to [\![B_1]\!]) \ldots ([\![P_n]\!] \to [\![B_n]\!]) \\
&\qquad (pat_{n+1} \to [\![C'[x]]\!]) \ldots (pat_m \to [\![C'[x]]\!])] \\
&\qquad (pat_{m+1} \to \bot) \ldots (pat_N \to \bot))]) \\
\xrightarrow{ucpb}\ &(\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][(\texttt{case}\ [\![e_1]\!] \\
&\qquad ([\![P_1]\!] \to [\![B_1]\!]) \ldots ([\![P_n]\!] \to [\![B_n]\!]) \\
&\qquad (pat_{n+1} \to [\![C'[e]]\!]) \ldots (pat_m \to [\![C'[x]]\!])] \\
&\qquad (pat_{m+1} \to \bot) \ldots (pat_N \to \bot))]) \\
\xrightarrow{(ucpb)^*}\ &(\texttt{letrec}\ x = [\![e]\!]\ \texttt{in}\ [\![C]\!][(\texttt{case}\ [\![e_1]\!] \\
&\qquad ([\![P_1]\!] \to [\![B_1]\!]) \ldots ([\![P_n]\!] \to [\![B_n]\!]) \\
&\qquad (pat_{n+1} \to [\![C'[e]]\!]) \ldots (pat_m \to [\![C'[e]]\!])] \\
&\qquad (pat_{m+1} \to \bot) \ldots (pat_N \to \bot))]) \\
\equiv\ &[\![\texttt{let(rec)}\ x = e\ \texttt{in}\ C[\texttt{case}\ e_1\ \texttt{of}\ \{P_1\ \texttt{->}\ B_1; \ldots; P_n\ \texttt{->}\ B_n; y\ \texttt{->}\ C'[e]\}]\!]
\end{aligned}
$$

A.6 Berechnung zu Lemma 4.23

I. Beide `case`-Ausdrücke enthalten keine default-Alternative.

Seien $q_{m+1}, \ldots, q_N$ Pattern für alle Konstruktoren aus $\mathcal{C}$, die nicht durch die Pattern $[\![Q_1]\!], \ldots [\![Q_m]\!]$ abgedeckt werden. Analog seien $p_{n+1}, \ldots, p_N$ Pattern für alle Konstruktoren aus $\mathcal{C}$, die nicht durch die Pattern $[\![P_1]\!], \ldots, [\![P_n]\!]$ abgedeckt werden.

$$
\begin{aligned}
&[\![\texttt{case}\ (\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n\})\ \texttt{of} \\
&\quad \{Q_1\ \texttt{->}\ S_1;\ \ldots;\ Q_m\ \texttt{->}\ S_m\}]\!] \\
\equiv\ &(\texttt{case}\ [\![(\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n\})]\!] \\
&\quad ([\![Q_1]\!] \to [\![S_1]\!])\ \ldots\ ([\![Q_m]\!] \to [\![S_m]\!])\ (q_{m+1} \to \bot)\ \ldots\ (q_N \to \bot)) \\
\equiv\ &(\texttt{case}\ (\texttt{case}\ [\![E]\!]\ ([\![P_1]\!] \to [\![R_1]\!])\ \ldots\ ([\![P_n]\!] \to [\![R_n]\!])\ (p_{n+1} \to \bot)\ \ldots (p_N \to \bot)) \\
&\quad ([\![Q_1]\!] \to [\![S_1]\!])\ \ldots\ ([\![Q_m]\!] \to [\![S_m]\!])\ (q_{m+1} \to \bot)\ \ldots\ (q_N \to \bot)) \\
\xrightarrow{ccase}\ &(\texttt{case}\ E \\
&\quad ([\![P_1]\!] \to (\texttt{case}\ [\![R_1]\!]\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
&\qquad\qquad (q_{m+1} \to \bot) \ldots (q_N \to \bot))) \\
&\quad \ldots \\
&\quad ([\![P_n]\!] \to (\texttt{case}\ [\![R_n]\!]\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
&\qquad\qquad (q_{m+1} \to \bot) \ldots (q_N \to \bot))) \\
&\quad (p_{n+1} \to (\texttt{case}\ \bot\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
&\qquad\qquad (q_{m+1} \to \bot) \ldots (q_N \to \bot))) \\
&\quad \ldots \\
&\quad (p_N \to (\texttt{case}\ \bot\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
&\qquad\qquad (q_{m+1} \to \bot) \ldots (q_N \to \bot)))
\end{aligned}
$$

$$
\begin{aligned}
&\sim_c (\texttt{case}\ E \\
&\quad (\llbracket P_1 \rrbracket \to (\texttt{case}\ \llbracket R_1 \rrbracket\ (\llbracket Q_1 \rrbracket \to \llbracket S_1 \rrbracket) \ldots (\llbracket Q_m \rrbracket \to \llbracket S_m \rrbracket) \\
&\qquad\qquad (q_{m+1} \to \bot) \ldots (q_N \to \bot))) \\
&\quad \ldots \\
&\quad (\llbracket P_n \rrbracket \to (\texttt{case}\ \llbracket R_n \rrbracket\ (\llbracket Q_1 \rrbracket \to \llbracket S_1 \rrbracket) \ldots (\llbracket Q_m \rrbracket \to \llbracket S_m \rrbracket) \\
&\qquad\qquad (q_{m+1} \to \bot) \ldots (q_N \to \bot))) \\
&\quad (p_{n+1} \to \bot) \ldots (p_N \to \bot) \\
&\equiv (\texttt{case}\ E \\
&\quad (\llbracket P_1 \rrbracket \to \llbracket \texttt{case}\ R_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m\} \rrbracket \\
&\quad \ldots \\
&\quad (\llbracket P_n \rrbracket \to \llbracket \texttt{case}\ R_n\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m\} \rrbracket \\
&\quad (p_{n+1} \to \bot) \ldots (p_N \to \bot) \\
&\equiv \llbracket (\texttt{case}\ E\ \texttt{of} \\
&\quad P_1\ \texttt{->}\ \texttt{case}\ R_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m\}; \\
&\quad \ldots \\
&\quad P_n\ \texttt{->}\ \texttt{case}\ R_n\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m\} \rrbracket
\end{aligned}
$$

II. Beide `case`-Ausdrücke enthalten eine default-Alternative.

Seien $q_{m+1}, \ldots, q_k$ (analog $p_{n+1}, \ldots, p_l$) Pattern für alle Konstruktoren aus $\mathcal{C}$, die nicht durch die Pattern $\llbracket Q_1 \rrbracket, \ldots \llbracket Q_m \rrbracket$ (analog $\llbracket P_1 \rrbracket, \ldots, \llbracket P_n \rrbracket$) abgedeckt werden, aber vom selben Typ wie die durch sie abgedeckten Konstruktoren sind. Seien $q_{k+1}, \ldots, q_N$ (analog $p_{l+1}, \ldots, p_N$) Pattern für die restlichen Konstruktoren aus $\mathcal{C}$. Falls nur die default-Alternative y `->` E_2 (analog x `->` E_1) existiert, seien $q_{m+1}, \ldots, q_k$ (analog $p_{n+1}, \ldots, p_l$) Pattern für alle Konstruktoren aus $\mathcal{C}$.

$$
\begin{aligned}
&\llbracket \texttt{case}\ (\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n; x\ \texttt{->}\ E_1\})\ \texttt{of} \\
&\qquad \{Q_1\ \texttt{->}\ S_1;\ \ldots;\ Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\} \rrbracket \\
&\equiv (\texttt{letrec}\ z_y = \llbracket (\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n; x\ \texttt{->}\ E_1\}) \rrbracket \\
&\quad \texttt{in}\ (\texttt{case}\ z_y\ (\llbracket Q_1 \rrbracket \to \llbracket S_1 \rrbracket) \ldots (\llbracket Q_m \rrbracket \to \llbracket S_m \rrbracket) \\
&\qquad\qquad (q_{m+1} \to \llbracket E_2 \rrbracket[z_y/y]) \ldots (q_k \to \llbracket E_2 \rrbracket[z_y/y]) \\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))) \\
&\equiv (\texttt{letrec}\ z_y = (\texttt{letrec}\ z_x = \llbracket E \rrbracket \\
&\qquad\qquad \texttt{in}\ (\texttt{case}\ z_x\ (\llbracket P_1 \rrbracket \to \llbracket R_1 \rrbracket) \ldots (\llbracket P_n \rrbracket \to \llbracket R_n \rrbracket) \\
&\qquad\qquad\qquad\qquad (p_{n+1} \to \llbracket E_1 \rrbracket[z_x/x]) \ldots (p_l \to \llbracket E_1 \rrbracket[z_x/x]) \\
&\qquad\qquad\qquad\qquad (p_{l+1} \to \bot) \ldots (p_N \to \bot))) \\
&\quad \texttt{in}\ (\texttt{case}\ z_y\ (\llbracket Q_1 \rrbracket \to \llbracket S_1 \rrbracket) \ldots (\llbracket Q_m \rrbracket \to \llbracket S_m \rrbracket) \\
&\qquad\qquad (q_{m+1} \to \llbracket E_2 \rrbracket[z_y/y]) \ldots (q_k \to \llbracket E_2 \rrbracket[z_y/y]) \\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))) \\
&\xrightarrow{llet} (\texttt{letrec}\ z_x = \llbracket E \rrbracket, z_y = (\texttt{case}\ z_x\ (\llbracket P_1 \rrbracket \to \llbracket R_1 \rrbracket) \ldots (\llbracket P_n \rrbracket \to \llbracket R_n \rrbracket) \\
&\qquad\qquad\qquad\qquad (p_{n+1} \to \llbracket E_1 \rrbracket[z_x/x]) \ldots (p_l \to \llbracket E_1 \rrbracket[z_x/x]) \\
&\qquad\qquad\qquad\qquad (p_{l+1} \to \bot) \ldots (p_N \to \bot)) \\
&\quad \texttt{in}\ (\texttt{case}\ z_y\ (\llbracket Q_1 \rrbracket \to \llbracket S_1 \rrbracket) \ldots (\llbracket Q_m \rrbracket \to \llbracket S_m \rrbracket) \\
&\qquad\qquad (q_{m+1} \to \llbracket E_2 \rrbracket[z_y/y]) \ldots (q_k \to \llbracket E_2 \rrbracket[z_y/y]) \\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot)))
\end{aligned}
$$

$$
\begin{array}{l}
\xleftarrow{llet} (\texttt{letrec}\ z_x = [\![E]\!] \\
\quad \texttt{in}\ (\texttt{letrec}\ z_y = (\texttt{case}\ z_x\ ([\![P_1]\!] \to [\![R_1]\!]) \ldots ([\![P_n]\!] \to [\![R_n]\!]) \\
\qquad\qquad\qquad\qquad (p_{n+1} \to [\![E_1]\!][z_x/x]) \ldots (p_l \to [\![E_1]\!][z_x/x]) \\
\qquad\qquad\qquad\qquad (p_{l+1} \to \bot) \ldots (p_N \to \bot)) \\
\qquad \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
\qquad\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y]) \\
\qquad\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))))
\end{array}
$$

$$
\begin{array}{l}
\xrightarrow{ccase-in} (\texttt{letrec}\ z_x = [\![E]\!] \\
\quad \texttt{in}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to (\texttt{letrec}\ z_y = [\![R_1]\!] \\
\qquad\qquad\qquad\qquad \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot)))) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad ([\![P_n]\!] \to (\texttt{letrec}\ z_y = [\![R_n]\!]\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots))) \\
\qquad\qquad\qquad (p_{n+1} \to (\texttt{letrec}\ z_y = [\![E_1]\!][z_x/x]\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots))) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad (p_l \to (\texttt{letrec}\ z_y = [\![E_1]\!][z_x/x]\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots))) \\
\qquad\qquad\qquad (p_{l+1} \to (\texttt{letrec}\ z_y = \bot\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots))) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad (p_N \to (\texttt{letrec}\ z_y = \bot\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots)))
\end{array}
$$

$$
\begin{array}{l}
\sim_c (\texttt{letrec}\ z_x = [\![E]\!] \\
\quad \texttt{in}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to (\texttt{letrec}\ z_y = [\![R_1]\!] \\
\qquad\qquad\qquad\qquad \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot)))) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad ([\![P_n]\!] \to (\texttt{letrec}\ z_y = [\![R_n]\!]\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots))) \\
\qquad\qquad\qquad (p_{n+1} \to (\texttt{letrec}\ z_y = [\![E_1]\!][z_x/x]\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots))) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad (p_l \to (\texttt{letrec}\ z_y = [\![E_1]\!][z_x/x]\ \texttt{in}\ (\texttt{case}\ z_y\ \ldots))) \\
\qquad\qquad\qquad (p_{l+1} \to \bot) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad (p_N \to \bot)))
\end{array}
$$

$$
\begin{array}{l}
\equiv (\texttt{letrec}\ z_x = [\![E]\!] \\
\quad \texttt{in}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to [\![\texttt{case}\ R_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!]) \\
\qquad\qquad \ldots \\
\qquad\qquad ([\![P_n]\!] \to [\![\texttt{case}\ R_n\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!]) \\
\qquad\qquad (p_{n+1} \to [\![\texttt{case}\ E_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!][z_x/x]) \\
\qquad\qquad \ldots \\
\qquad\qquad (p_l \to [\![\texttt{case}\ E_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!][z_x/x]) \\
\qquad\qquad (p_{l+1} \to \bot) \\
\qquad\qquad \ldots \\
\qquad\qquad (p_N \to \bot))) \\
\equiv [\![\texttt{case}\ E\ \texttt{of}\ \\
\qquad P_1\ \texttt{->}\ \texttt{case}\ R_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}; \\
\qquad \ldots \\
\qquad P_n\ \texttt{->}\ \texttt{case}\ R_n\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\} \\
\qquad x\ \texttt{->}\ \texttt{case}\ E_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!]
\end{array}
$$

III. Nur die Alternativen $(P_i\ \texttt{->}\ R_i)$ für $i = 1, \ldots, n$ enthalten eine default-Alternative.

Seien $[\![Alts_Q]\!]$ die durch Übersetzung der Alternativen $Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m$ entstandenen Alternativen, wobei zu beachten ist, dass die Übersetzung nicht zu einem äußeren `letrec`-Ausdruck führen kann, da diese Alternativen keine default-Alternative enthalten.

Seien $p_{n+1}, \ldots, p_l$ Pattern für alle Konstruktoren aus $\mathcal{C}$, die nicht durch die Pattern $[\![P_1]\!], \ldots, [\![P_n]\!]$ abgedeckt werden, aber vom selben Typ wie die durch sie abgedeckten Konstruktoren sind. Seien $p_{l+1}, \ldots, p_N$ Pattern für die restlichen Konstruktoren aus $\mathcal{C}$.

Falls nur die default-Alternative $x\ \texttt{->}\ E_1$ existiert, seien $p_{n+1}, \ldots, p_l$ Pattern für alle Konstruktoren aus $\mathcal{C}$.

$$
\begin{array}{l}
\quad [\![\texttt{case}\ (\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n; x\ \texttt{->}\ E_1\})\ \texttt{of}\ Alts_Q]\!] \\
\equiv (\texttt{case}\ [\![(\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n; x\ \texttt{->}\ E_1\})]\!]\ [\![Alts_Q]\!]) \\
\equiv (\texttt{case}\ (\texttt{letrec}\ z_x = [\![E]\!]\ \texttt{in}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to [\![R_1]\!]) \ldots ([\![P_n]\!] \to [\![R_n]\!]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (p_{n+1} \to [\![E_1]\!][z_x/x]) \ldots (p_l \to [\![E_1]\!][z_x/x]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (p_{l+1} \to \bot) \ldots (p_N \to \bot))) \\
\qquad [\![Alts_Q]\!]) \\
\xrightarrow{lcase} (\texttt{letrec}\ z_x = [\![E]\!]\ \texttt{in}\ (\texttt{case}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to [\![R_1]\!]) \ldots ([\![P_n]\!] \to [\![R_n]\!]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (p_{n+1} \to [\![E_1]\!][z_x/x]) \ldots (p_l \to [\![E_1]\!][z_x/x]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (p_{l+1} \to \bot) \ldots (p_N \to \bot)) \\
\qquad\qquad\qquad\qquad [\![Alts_Q]\!]))
\end{array}
$$

$$
\begin{array}{ll}
\xrightarrow{ccase} & (\texttt{letrec}\ z_x = [\![E]\!] \\
& \quad \texttt{in}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to (\texttt{case}\ [\![R_1]\!]\ [\![Alts_Q]\!])) \ldots ([\![P_n]\!] \to (\texttt{case}\ [\![R_n]\!]\ [\![Alts_Q]\!])) \\
& \qquad\qquad (p_{n+1} \to (\texttt{case}\ [\![E_1]\!][z_x/x]\ [\![Alts_Q]\!])) \ldots (p_l \to (\texttt{case}\ [\![E_1]\!][z_x/x]\ [\![Alts_Q]\!])) \\
& \qquad\qquad (p_{l+1} \to (\texttt{case}\ \bot\ [\![Alts_Q]\!])) \ldots (p_N \to (\texttt{case}\ \bot\ [\![Alts_Q]\!])))) \\
\sim_c & (\texttt{letrec}\ z_x = [\![E]\!] \\
& \quad \texttt{in}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to (\texttt{case}\ [\![R_1]\!]\ [\![Alts_Q]\!])) \ldots ([\![P_n]\!] \to (\texttt{case}\ [\![R_n]\!]\ [\![Alts_Q]\!])) \\
& \qquad\qquad (p_{n+1} \to (\texttt{case}\ [\![E_1]\!][z_x/x]\ [\![Alts_Q]\!])) \ldots (p_l \to (\texttt{case}\ [\![E_1]\!][z_x/x]\ [\![Alts_Q]\!])) \\
& \qquad\qquad (p_{l+1} \to \bot) \ldots (p_N \to \bot))) \\
\equiv & (\texttt{letrec}\ z_x = [\![E]\!] \\
& \quad \texttt{in}\ (\texttt{case}\ z_x\ ([\![P_1]\!] \to [\![\texttt{case}\ R_1\ \texttt{of}\ Alts_Q]\!]) \ldots ([\![P_n]\!] \to [\![\texttt{case}\ R_n\ \texttt{of}\ Alts_Q]\!]) \\
& \qquad\qquad (p_{n+1} \to [\![\texttt{case}\ E_1\ \texttt{of}\ Alts_Q]\!][z_x/x]) \ldots (p_l \to [\![\texttt{case}\ E_1\ \texttt{of}\ Alts_Q]\!][z_x/x]) \\
& \qquad\qquad (p_{l+1} \to \bot) \ldots (p_N \to \bot))) \\
\equiv & [\![\texttt{case}\ E\ \texttt{of} \\
& \quad P_1\ \texttt{->}\ \texttt{case}\ R_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m\}; \\
& \quad \ldots \\
& \quad P_n\ \texttt{->}\ \texttt{case}\ R_n\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m\} \\
& \quad x\ \texttt{->}\ \texttt{case}\ E_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m\}]\!]
\end{array}
$$

IV. Nur die Alternativen $(Q_i\ \texttt{->}\ S_i)$ für $i = 1, \ldots, m$ enthalten eine default-Alternative.

Seien $p_{n+1}, \ldots p_N$ Pattern für alle Konstruktoren aus $\mathcal{C}$, die nicht durch die Pattern $[\![P_1]\!], \ldots, [\![P_n]\!]$ abgedeckt werden. Seien $q_{m+1}, \ldots, q_k$ Pattern für alle Konstruktoren aus $\mathcal{C}$, die nicht durch die Pattern $[\![Q_1]\!], \ldots [\![Q_m]\!]$ abgedeckt werden, aber vom selben Typ wie die durch sie abgedeckten Konstruktoren sind.

Seien $q_{k+1}, \ldots, q_N$ Pattern für die restlichen Konstruktoren aus $\mathcal{C}$. Falls nur die default-Alternative $y\ \texttt{->}\ E_2$ existiert, seien $q_{m+1}, \ldots, q_k$ Pattern für alle Konstruktoren aus $\mathcal{C}$.

$$
\begin{array}{ll}
& [\![\texttt{case}\ (\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n\})\ \texttt{of} \\
& \quad \{Q_1\ \texttt{->}\ S_1;\ \ldots;\ Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!] \\
\equiv & (\texttt{letrec}\ z_y = [\![(\texttt{case}\ E\ \texttt{of}\ \{P_1\ \texttt{->}\ R_1;\ \ldots;\ P_n\ \texttt{->}\ R_n\})]\!] \\
& \quad \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
& \qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y]) \\
& \qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))) \\
\equiv & (\texttt{letrec}\ z_y = (\texttt{case}\ [\![E]\!]\ ([\![P_1]\!] \to [\![R_1]\!]) \ldots ([\![P_n]\!] \to [\![R_n]\!])(p_{n+1} \to \bot) \ldots (p_N \to \bot)) \\
& \quad \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!]) \\
& \qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y]) \\
& \qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot)))
\end{array}
$$

$$
\begin{aligned}
&\xrightarrow{ccase-in} (\texttt{case}\ [\![E]\!]\\
&\quad([\![P_1]\!] \to (\texttt{letrec}\ z_y = [\![R_1]\!]\ \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!])\\
&\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y])\\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))))\\
&\quad \ldots\\
&\quad([\![P_n]\!] \to (\texttt{letrec}\ z_y = [\![R_n]\!]\ \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!])\\
&\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y])\\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))))\\
&\quad(p_{n+1} \to (\texttt{letrec}\ z_y = \bot\ \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!])\\
&\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y])\\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))))\\
&\quad \ldots\\
&\quad(p_N \to (\texttt{letrec}\ z_y = \bot\ \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!])\\
&\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y])\\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot)))))\\
&\sim_c (\texttt{case}\ [\![E]\!]\\
&\quad([\![P_1]\!] \to (\texttt{letrec}\ z_y = [\![R_1]\!]\ \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!])\\
&\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y])\\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))))\\
&\quad \ldots\\
&\quad([\![P_n]\!] \to (\texttt{letrec}\ z_y = [\![R_n]\!]\ \texttt{in}\ (\texttt{case}\ z_y\ ([\![Q_1]\!] \to [\![S_1]\!]) \ldots ([\![Q_m]\!] \to [\![S_m]\!])\\
&\qquad\qquad (q_{m+1} \to [\![E_2]\!][z_y/y]) \ldots (q_k \to [\![E_2]\!][z_y/y])\\
&\qquad\qquad (q_{k+1} \to \bot) \ldots (q_N \to \bot))))\\
&\quad(p_{n+1} \to \bot) \ldots (p_N \to \bot))\\
&\equiv\ (\texttt{case}\ [\![E]\!]\ ([\![P_1]\!] \to [\![\texttt{case}\ R_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!])\\
&\qquad\qquad \ldots\\
&\qquad\qquad ([\![P_n]\!] \to [\![\texttt{case}\ R_n\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!])\\
&\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot))\\
&\equiv [\![\texttt{case}\ E\ \texttt{of}\\
&\quad P_1\ \texttt{->}\ \texttt{case}\ R_1\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\};\\
&\quad \ldots\\
&\quad P_n\ \texttt{->}\ \texttt{case}\ R_n\ \texttt{of}\ \{Q_1\ \texttt{->}\ S_1; \ldots; Q_m\ \texttt{->}\ S_m; y\ \texttt{->}\ E_2\}]\!]
\end{aligned}
$$

A.7 Berechnung zu Lemma 4.24

$$
\begin{array}{l}
[\![\texttt{case}\ x\ \texttt{of}\ c_1\ a_{1,1}\ldots a_{1,ar(c_1)}\ \texttt{->}\ t_1,\\
\qquad\qquad \ldots\\
\qquad\qquad c_k\ a_{k,1}\ldots a_{k,ar(c_k)}\ \texttt{->}\ t_k,\\
\qquad\qquad y\ \texttt{->}\ \texttt{case}\ x\ \texttt{of}\ c_{k+1}\ b_{k,1}\ldots b_{k,ar(c_{k+1})}\ \texttt{->}\ t_{k+1},\\
\qquad\qquad\qquad\qquad \ldots\\
\qquad\qquad\qquad\qquad c_m\ b_{m,1}\ldots b_{m,ar(c_m)}\ \texttt{->}\ t_m]\!]\\
\equiv (\texttt{letrec}\ z = x\ \texttt{in}\\
\quad (\texttt{case}\ z\ ((c_1\ a_{1,1}\ldots a_{1,ar(c_1)}) \to [\![t_1]\!])\ldots((c_k\ a_{k,1}\ldots a_{k,ar(c_k)}) \to [\![t_k]\!])\\
\qquad ((c_{k+1}\ a_{k+1,1}\ldots a_{k+1,ar(c_{k+1})})\\
\qquad\quad \to (\texttt{case}\ x\\
\qquad\qquad ((c_1\ b_{1,1}\ldots b_{1,ar(c_1)}) \to \bot)\ldots((c_k\ b_{k,1}\ldots b_{k,ar(c_k)}) \to \bot)\\
\qquad\qquad ((c_{k+1}\ b_{k+1,1}\ldots b_{k+1,ar(c_{k+1})}) \to [\![t_{k+1}]\!][z/y])\\
\qquad\qquad \ldots\\
\qquad\qquad ((c_m\ b_{m,1}\ldots b_{m,ar(c_m)}) \to [\![t_m]\!][z/y])\\
\qquad\qquad ((c_{m+1}\ b_{m+1,1}\ldots b_{m+1,ar(c_{m+1})}) \to \bot)\ldots(c_N \to \bot)))\\
\qquad \ldots\\
\qquad ((c_n\ a_{n,1}\ldots a_{n,ar(c_n)})\\
\qquad\quad \to (\texttt{case}\ x\\
\qquad\qquad ((c_1\ b_{1,1}\ldots b_{1,ar(c_1)}) \to \bot)\ldots((c_k\ b_{k,1}\ldots b_{k,ar(c_k)}) \to \bot)\\
\qquad\qquad ((c_{k+1}\ b_{k+1,1}\ldots b_{k+1,ar(c_{k+1})}) \to [\![t_{k+1}]\!][z/y])\\
\qquad\qquad \ldots\\
\qquad\qquad ((c_m\ b_{m,1}\ldots b_{m,ar(c_m)}) \to [\![t_m]\!][z/y])\\
\qquad\qquad ((c_{m+1}\ b_{m+1,1}\ldots b_{m+1,ar(c_{m+1})}) \to \bot)\ldots(c_N \to \bot)))\\
\qquad ((c_{n+1}\ a_{n+1,1}\ldots a_{n+1,ar(c_{n+1})}) \to \bot)\ldots(c_N \to \bot)))\\
\xrightarrow{cpx} (\texttt{letrec}\ z = x\ \texttt{in}\ (\texttt{case}\ x\ldots\\
\xrightarrow{(ccpcx)^*} (\texttt{letrec}\ z = x\ \texttt{in}\\
\quad (\texttt{case}\ x\\
\qquad ((c_1\ a_{1,1}\ldots a_{1,ar(c_1)}) \to [\![t_1]\!])\ldots((c_k\ a_{k,1}\ldots a_{k,ar(c_k)}) \to [\![t_k]\!])\\
\qquad ((c_{k+1}\ a_{k+1,1}\ldots a_{k+1,ar(c_{k+1})})\\
\qquad\quad \to (\texttt{case}\ (c_{k+1}\ a_{k+1,1}\ldots a_{k+1,ar(c_{k+1})})\\
\qquad\qquad (c_1\ b_{1,1}\ldots b_{1,ar(c_1)}) \to \bot)\ldots((c_k\ b_{k,1}\ldots b_{k,ar(c_k)}) \to \bot)\\
\qquad\qquad ((c_{k+1}\ b_{k+1,1}\ldots b_{k+1,ar(c_{k+1})}) \to [\![t_{k+1}]\!][z/y])\\
\qquad\qquad \ldots\\
\qquad\qquad ((c_m\ b_{m,1}\ldots b_{m,ar(c_m)}) \to [\![t_m]\!][z/y])\\
\qquad\qquad ((c_{m+1}\ b_{m+1,1}\ldots b_{m+1,ar(c_{m+1})}) \to \bot)\ldots(c_N \to \bot)))\\
\qquad \ldots))
\end{array}
$$

$$
\begin{array}{ll}
\xrightarrow{(case-c)^*} & (\texttt{letrec } z = x \texttt{ in} \\
& (\texttt{case } x \\
& \quad ((c_1\ a_{1,1} \ldots a_{1,ar(c_1)}) \to [\![t_1]\!]) \ldots ((c_k\ a_{k,1} \ldots a_{k,ar(c_k)}) \to [\![t_k]\!]) \\
& \quad ((c_{k+1}\ a_{k+1,1} \ldots a_{k+1,ar(c_{k+1})}) \\
& \qquad \to (\texttt{letrec } b_{k+1,1} = a_{k+1,1},\ \ldots, b_{k+1,ar(c_{k+1})} = a_{k+1,ar(c_{k+1})} \\
& \qquad\quad \texttt{in } [\![t_{k+1}]\!][z/y])) \\
& \quad \ldots \\
& \quad ((c_m\ a_{m,1} \ldots a_{m,ar(c_m)}) \\
& \qquad \to (\texttt{letrec } b_{m,1} = a_{m,1},\ \ldots, b_{m,ar(c_{k+1})} = a_{m,ar(c_m)} \texttt{ in } [\![t_m]\!][z/y])) \\
& \quad ((c_{m+1}\ a_{m+1,1} \ldots a_{m+1,ar(c_{m+1})}) \\
& \qquad \to (\texttt{letrec } b_{m+1,1} = a_{m+1,1},\ \ldots, b_{m+1,ar(c_{m+1})} = a_{m+1,ar(c_{m+1})} \texttt{ in } \bot)) \\
& \quad \ldots \\
& \quad ((c_n\ a_{n,1} \ldots a_{n,ar(c_n)}) \\
& \qquad \to (\texttt{letrec } b_{n,1} = a_{n,1},\ \ldots, b_{n,ar(c_n)} = a_{n,ar(c_n)} \texttt{ in } \bot)) \\
& \quad ((c_{n+1}\ a_{n+1,1} \ldots a_{n+1,ar(c_{n+1})}) \to \bot) \ldots (c_N \to \bot))) \\
\xrightarrow{(cpx)^*}\xrightarrow{(gc)^*} & (\texttt{letrec } z = x \texttt{ in} \\
& (\texttt{case } x \\
& \quad ((c_1\ a_{1,1} \ldots a_{1,ar(c_1)}) \to [\![t_1]\!]) \ldots ((c_k\ a_{k,1} \ldots a_{k,ar(c_k)}) \to [\![t_k]\!]) \\
& \quad ((c_{k+1}\ a_{k+1,1} \ldots a_{k+1,ar(c_{k+1})}) \to [\![t_{k+1}]\!][z/y]) \\
& \quad \ldots \\
& \quad ((c_m\ a_{m,1} \ldots a_{m,ar(c_m)}) \to [\![t_m]\!][z/y]) \\
& \quad ((c_{m+1}\ a_{m+1,1} \ldots a_{m+1,ar(c_{m+1})}) \to \bot) \ldots (c_N \to \bot))) \\
\xleftarrow{(cpx)^*} & (\texttt{letrec } z = x \texttt{ in} \\
& (\texttt{case } x \\
& \quad ((c_1\ a_{1,1} \ldots a_{1,ar(c_1)}) \to [\![t_1]\!]) \ldots ((c_k\ a_{k,1} \ldots a_{k,ar(c_k)}) \to [\![t_k]\!]) \\
& \quad ((c_{k+1}\ a_{k+1,1} \ldots a_{k+1,ar(c_{k+1})}) \to [\![t_{k+1}]\!][x/y]) \\
& \quad \ldots \\
& \quad ((c_m\ a_{m,1} \ldots a_{m,ar(c_m)}) \to [\![t_m]\!][x/y]) \\
& \quad ((c_{m+1}\ a_{m+1,1} \ldots a_{m+1,ar(c_{m+1})}) \to \bot) \ldots (c_N \to \bot))) \\
\xrightarrow{gc} & (\texttt{case } x \\
& \quad ((c_1\ a_{1,1} \ldots a_{1,ar(c_1)}) \to [\![t_1]\!]) \ldots ((c_k\ a_{k,1} \ldots a_{k,ar(c_k)}) \to [\![t_k]\!]) \\
& \quad ((c_{k+1}\ a_{k+1,1} \ldots a_{k+1,ar(c_{k+1})}) \to [\![t_{k+1}]\!][x/y]) \\
& \quad \ldots \\
& \quad ((c_m\ a_{m,1} \ldots a_{m,ar(c_m)}) \to [\![t_m]\!][x/y]) \\
& \quad ((c_{m+1}\ a_{m+1,1} \ldots a_{m+1,ar(c_{m+1})}) \to \bot) \ldots (c_N \to \bot))
\end{array}
$$

$$
\begin{aligned}
\equiv\ & [\![\texttt{case}\ x\ \texttt{of}\\
& \quad c_1\ a_{1,1}\ldots a_{1,ar(c_1)})\ \texttt{->}\ t_1,\\
& \quad \ldots\\
& \quad c_k\ a_{k,1}\ldots a_{k,ar(c_k)})\ \texttt{->}\ t_k,\\
& \quad c_{k+1}\ a_{k+1,1}\ldots a_{k+1,ar(c_{k+1})}\ \texttt{->}\ t_{k+1}[x/y],\\
& \quad \ldots\\
& \quad c_m\ a_{m,1}\ldots a_{m,ar(c_m)}\ \texttt{->}\ t_m[x/y]]\!]
\end{aligned}
$$

A.8 Berechnung zu Lemma 4.35

Seien die Pattern $pat_{n+1},\ldots,pat_N$ so gewählt, dass sie die Konstruktoren aus $\mathcal{C} \setminus \{[\![c_1]\!],\ldots,[\![c_n]\!]\}$ abdecken.

$$
\begin{aligned}
& [\![\texttt{let}\ v = \texttt{case}\ E_1\ \texttt{of}\ \{c_1\ a_{1,1}\ldots a_{1,m_1}\ \texttt{->}\ t_1;\ldots;c_n\ a_{n,1}\ldots a_{n,m_n}\ \texttt{->}\ t_n\}\ \texttt{in}\ E_3]\!]\\
\equiv\ & (\texttt{letrec}\ v = (\texttt{case}\ [\![E_1]\!]\ ([\![c_1]\!]\ a_{1,1}\ldots a_{1,m_1} \to [\![t_1]\!])\ldots\\
& \qquad\qquad ([\![c_n]\!]\ a_{n,1}\ldots a_{n,m_n} \to [\![t_n]\!])\ldots\\
& \qquad\qquad (pat_{n+1} \to \bot)\ldots(pat_N \to \bot))\\
& \quad \texttt{in}\ [\![E_3]\!])\\
\xleftarrow{lbeta}\ & (\lambda v.[\![E_3]\!])\ (\texttt{case}\ [\![E_1]\!]\ ([\![c_1]\!]\ a_{1,1}\ldots a_{1,m_1} \to [\![t_1]\!])\ldots\\
& \qquad\qquad ([\![c_n]\!]\ a_{n,1}\ldots a_{n,m_n} \to [\![t_n]\!])\ldots\\
& \qquad\qquad (pat_{n+1} \to \bot)\ldots(pat_N \to \bot))\\
\xrightarrow{streval}\ & (\texttt{letrec}\ w = (\texttt{case}\ [\![E_1]\!]\ ([\![c_1]\!]\ a_{1,1}\ldots a_{1,m_1} \to [\![t_1]\!])\ldots\\
& \qquad\qquad ([\![c_n]\!]\ a_{n,1}\ldots a_{n,m_n} \to [\![t_n]\!])\ldots\\
& \qquad\qquad (pat_{n+1} \to \bot)\ldots(pat_N \to \bot))\\
& \quad \texttt{in}\ (\texttt{case}\ w\ (pat'_1 \to ((\lambda v.[\![E_3]\!])\ w))\ldots(pat'_N \to ((\lambda v.[\![E_3]\!])\ w))))\\
\xrightarrow{ccase-in}\ & (\texttt{case}\ [\![E_1]\!]\\
& \quad ([\![c_1]\!]\ a_{1,1}\ldots a_{1,m_1} \to (\texttt{letrec}\ w = [\![t_1]\!]\ \texttt{in}\ (\texttt{case}\ w\ (pat'_1 \to ((\lambda v.[\![E_3]\!])\ w))\ldots\\
& \qquad\qquad (pat'_N \to ((\lambda v.[\![E_3]\!])\ w)))\\
& \quad \ldots\\
& \quad ([\![c_n]\!]\ a_{n,1}\ldots a_{n,m_n} \to (\texttt{letrec}\ w = [\![t_n]\!]\ \texttt{in}\ (\texttt{case}\ w\ (pat'_1 \to ((\lambda v.[\![E_3]\!])\ w))\ldots\\
& \qquad\qquad (pat'_N \to ((\lambda v.[\![E_3]\!])\ w)))\\
& \quad (pat_{n+1} \to (\texttt{letrec}\ w = \bot\ \texttt{in}\ (\texttt{case}\ w\ (pat'_1 \to ((\lambda v.[\![E_3]\!])\ w))\ldots\\
& \qquad\qquad (pat'_N \to ((\lambda v.[\![E_3]\!])\ w)))\\
& \quad \ldots\\
& \quad (pat_N \to (\texttt{letrec}\ w = \bot\ \texttt{in}\ (\texttt{case}\ w\ (pat'_1 \to ((\lambda v.[\![E_3]\!])\ w))\ldots\\
& \qquad\qquad (pat'_N \to ((\lambda v.[\![E_3]\!])\ w))))
\end{aligned}
$$

$$
\begin{array}{rl}
& \sim_c (\texttt{case}\ [\![E_1]\!] \\
& \quad ([\![c_1]\!]\ a_{1,1} \ldots a_{1,m_1} \to (\texttt{letrec}\ w = [\![t_1]\!]\ \texttt{in}\ (\texttt{case}\ w\ (pat'_1 \to ((\lambda v.[\![E_3]\!])\ w)) \ldots \\
& \qquad\qquad (pat'_N \to ((\lambda v.[\![E_3]\!])\ w))) \\
& \quad \ldots \\
& \quad ([\![c_n]\!]\ a_{n,1} \ldots a_{n,m_n} \to (\texttt{letrec}\ w = [\![t_n]\!]\ \texttt{in}\ (\texttt{case}\ w\ (pat'_1 \to ((\lambda v.[\![E_3]\!])\ w)) \ldots \\
& \qquad\qquad (pat'_N \to ((\lambda v.[\![E_3]\!])\ w))) \\
& \quad (pat_{n+1} \to \bot)\ \ldots\ (pat_N \to \bot)) \\
\xleftarrow{(streval)^*} & (\texttt{case}\ [\![E_1]\!] \\
& \quad ([\![c_1]\!]\ a_{1,1} \ldots a_{1,m_1} \to ((\lambda v.[\![E_3]\!])\ [\![t_1]\!])) \\
& \quad \ldots \\
& \quad ([\![c_n]\!]\ a_{n,1} \ldots a_{n,m_n} \to ((\lambda v.[\![E_3]\!])\ [\![t_n]\!])) \\
& \quad (pat_{n+1} \to \bot)\ \ldots\ (pat_N \to \bot)) \\
\xrightarrow{(lbeta)^*} & (\texttt{case}\ [\![E_1]\!] \\
& \quad ([\![c_1]\!]\ a_{1,1} \ldots a_{1,m_1} \to (\texttt{letrec}\ v = [\![t_1]\!]\ \texttt{in}\ [\![E_3]\!])) \\
& \quad \ldots \\
& \quad ([\![c_n]\!]\ a_{n,1} \ldots a_{n,m_n} \to (\texttt{letrec}\ v = [\![t_n]\!]\ \texttt{in}\ [\![E_3]\!])) \\
& \quad (pat_{n+1} \to \bot)\ \ldots\ (pat_N \to \bot)) \\
\equiv & [\![\texttt{case}\ E_1\ \texttt{of} \\
& \qquad \{c_1\ a_{1,1} \ldots a_{1,m_1}\ \texttt{->}\ \texttt{let}\ v = t_1\ \texttt{in}\ E_3; \\
& \qquad\ \ldots \\
& \qquad\ c_n\ a_{n,1} \ldots a_{n,m_n}\ \texttt{->}\ \texttt{let}\ v = t_n\ \texttt{in}\ E_3\}]\!]
\end{array}
$$

Die verwendete kontextuelle Gleichheit gilt aufgrund von Satz 3.33.

A.9 Berechnung zu Lemma 4.36

Zu Zeigen: Sei $P = \{(\mathcal{B}, \texttt{'b'})\}$, so gilt $[\![s]\!]\Downarrow(P)$, aber $[\![t]\!]\Uparrow(P)$.

$$
\begin{array}{rl}
[\![s]\!] & = (\texttt{letrec}\ fun = ((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)) \\
& \quad \texttt{in}\ (\texttt{case}\ (fun\ \texttt{False})\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)) \\
\xrightarrow{n,lbeta} & (\texttt{letrec}\ fun = (\texttt{letrec}\ x'_1 = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)\ \texttt{in}\ (\lambda x_2.x'_1)) \\
& \quad \texttt{in}\ (\texttt{case}\ (fun\ \texttt{False})\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)) \\
\xrightarrow{n,llet} & (\texttt{letrec}\ fun = (\lambda x_2.x'_1), x'_1 = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots) \\
& \quad \texttt{in}\ (\texttt{case}\ (fun\ \texttt{False})\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)) \\
\xrightarrow{n,cp} & (\texttt{letrec}\ fun = (\lambda x_2.x'_1), x'_1 = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots) \\
& \quad \texttt{in}\ (\texttt{case}\ ((\lambda x_2.x'_1)\ \texttt{False})\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)) \\
\xrightarrow{n,lbeta} & (\texttt{letrec}\ fun = (\lambda x_2.x'_1), x'_1 = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots) \\
& \quad \texttt{in}\ (\texttt{case}\ (\texttt{letrec}\ x'_2 = \texttt{False}\ \texttt{in}\ x'_1)\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))
\end{array}
$$

$$\xrightarrow{n,lcase} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)$$
$$\texttt{in}\ (\texttt{letrec}\ x_2' = \texttt{False}\ \texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)))$$
$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots), x_2' = \texttt{False}$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,IOr,(\mathcal{B},\texttt{'b'})} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = (\texttt{case}\ \texttt{'b'}\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots), x_2' = \texttt{False}$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,case} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = (\texttt{letrec}\ \{\}\ \texttt{in}\ \texttt{'b'}), x_2' = \texttt{False}$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = \texttt{'b'}, x_2' = \texttt{False}$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,case} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = \texttt{'b'}, x_2' = \texttt{False}\ \texttt{in}\ (\texttt{letrec}\ \{\}\ \texttt{in}\ (fun\ \texttt{False})))$$
$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = \texttt{'b'}, x_2' = \texttt{False}\ \texttt{in}\ (fun\ \texttt{False}))$$
$$\xrightarrow{n,cp} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = \texttt{'b'}, x_2' = \texttt{False}\ \texttt{in}\ ((\lambda x_2.x_1')\ \texttt{False}))$$
$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = \texttt{'b'}, x_2' = \texttt{False}\ \texttt{in}\ (\texttt{letrec}\ x_2'' = \texttt{False}\ \texttt{in}\ x_1'))$$
$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = \texttt{'b'}, x_2' = \texttt{False}\ \texttt{in}\ (\texttt{letrec}\ x_2'' = \texttt{False}\ \texttt{in}\ x_1'))$$
$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda x_2.x_1'), x_1' = \texttt{'b'}, x_2' = \texttt{False}, x_2'' = \texttt{False}\ \texttt{in}\ x_1')$$

$$[\![t]\!] = (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y))$$
$$\texttt{in}\ (\texttt{case}\ (fun\ \texttt{False})\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,cp} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y))$$
$$\texttt{in}\ (\texttt{case}\ ((\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)\ y))\ \texttt{False})$$
$$(\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y))$$
$$\texttt{in}\ (\texttt{case}\ (\texttt{letrec}\ y' = \texttt{False}$$
$$\texttt{in}\ (((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y'))$$
$$(\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,lcase} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y))$$
$$\texttt{in}\ (\texttt{letrec}\ y' = \texttt{False}\ \texttt{in}$$
$$(\texttt{case}\ (((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y')$$
$$(\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)))$$
$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}$$
$$\texttt{in}\ (\texttt{case}\ (((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y')$$
$$(\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$
$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}$$
$$\texttt{in}\ (\texttt{case}\ ((\texttt{letrec}\ x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)\ \texttt{in}(\lambda x_2.x_1'))\ y')$$
$$(\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,lapp} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}$$
$$\texttt{in}\ (\texttt{case}\ (\texttt{letrec}\ x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)\ \texttt{in}\ ((\lambda x_2.x_1')\ y'))$$
$$(\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,lcase} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}$$
$$\texttt{in}\ (\texttt{letrec}\ x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)$$
$$\texttt{in}\ (\texttt{case}\ ((\lambda x_2.x_1')\ y')\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)$$
$$\texttt{in}\ (\texttt{case}\ ((\lambda x_2.x_1')\ y')\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)$$
$$\texttt{in}\ (\texttt{case}\ (\texttt{letrec}\ x_2' = y'\ \texttt{in}\ x_1')\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,lcase} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)$$
$$\texttt{in}\ (\texttt{letrec}\ x_2' = y'\ \texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots)))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots), x_2' = y'$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,IOr,(\mathcal{B},\texttt{'b'})} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = (\texttt{case}\ \texttt{'b'}\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots), x_2' = y'$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,case} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = (\texttt{letrec}\ \{\}\ \texttt{in}\ \texttt{'b'}), x_2' = y'$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y'$$
$$\texttt{in}\ (\texttt{case}\ x_1'\ (\texttt{'a'} \to \texttt{'a'})\ (\texttt{'b'} \to (fun\ \texttt{False}))\ \ldots))$$

$$\xrightarrow{n,case} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y'$$
$$\texttt{in}\ (\texttt{letrec}\ \{\}\ \texttt{in}\ (fun\ \texttt{False})))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y'$$
$$\texttt{in}\ (fun\ \texttt{False}))$$

$$\xrightarrow{n,cp} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y'$$
$$\texttt{in}\ ((\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y))\ \texttt{False}))$$

$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y'$$
$$\texttt{in}\ (\texttt{letrec}\ y'' = \texttt{False}\ \texttt{in}(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y'')))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y', y'' = \texttt{False}$$
$$\texttt{in}\ (((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y''))$$

$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y', y'' = \texttt{False}$$
$$\texttt{in}\ ((\texttt{letrec}\ x_1'' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)\ \texttt{in}\ (\lambda x_2.x_1''))\ y''))$$

$$\xrightarrow{n,lapp} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y', y'' = \texttt{False}$$
$$\texttt{in}\ (\texttt{letrec}\ x_1'' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)\ \texttt{in}\ ((\lambda x_2.x_1'')\ y'')))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y', y'' = \texttt{False},$$
$$x_1'' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)$$
$$\texttt{in}\ ((\lambda x_2.x_1'')\ y''))$$

$$\xrightarrow{n,lbeta} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y', y'' = \texttt{False},$$
$$x_1'' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots)$$
$$\texttt{in}\ (\texttt{letrec}\ x_2'' = y''\ \texttt{in}\ x_1''))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ fun = (\lambda y.(((\lambda x_1.(\lambda x_2.x_1))\ (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots))\ y)),$$
$$y' = \texttt{False}, x_1' = \texttt{'b'}, x_2' = y', y'' = \texttt{False},$$
$$x_1'' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots(\texttt{'b'} \to \texttt{'b'})\ldots), x_2'' = y''$$
$$\texttt{in}\ x_1'')$$

$\xrightarrow{n,IOr,?}$ Ein zweites IO-Paar wird benötigt!

A.10 Berechnung zu Lemma 4.37

Zu Zeigen: Wenn $P = \emptyset$, dann gilt $[\![t]\!]{\Downarrow}(P)$ und $[\![s]\!]{\Uparrow}(P)$.

$$[\![s]\!] \equiv (\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1)\ldots(p_n \to p_n)\ (p_{n+1} \to \bot)\ldots(p_N \to \bot)),$$
$$f = (\lambda x.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w))\ldots(p_N \to (\lambda w.w)))))$$
$$\texttt{in}\ (\texttt{letrec}\ v' = (f\ \texttt{True})\ \texttt{in}\ (\texttt{case}\ v'\ (p_1 \to \texttt{'a'})\ldots(p_N \to \texttt{'a'}))))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1)\ldots(p_n \to p_n)\ (p_{n+1} \to \bot)\ldots(p_N \to \bot)),$$
$$f = (\lambda x.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w))\ldots(p_N \to (\lambda w.w))))),$$
$$v' = (f\ \texttt{True})$$
$$\texttt{in}\ (\texttt{case}\ v'\ (p_1 \to \texttt{'a'})\ldots(p_N \to \texttt{'a'})))$$

$\xrightarrow{n,cp}$ (letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w)) \ldots (p_N \to (\lambda w.w)))))$,
$v' = ((\lambda x.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w)) \ldots (p_N \to (\lambda w.w))))))\ \texttt{True})$
in $(\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))$)

$\xrightarrow{n,lbeta}$ (letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w)) \ldots (p_N \to (\lambda w.w)))))$,
$v' = (\texttt{letrec}\ x' = \texttt{True}\ \texttt{in}$
$(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w)) \ldots (p_N \to (\lambda w.w)))))$
in $(\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))$)

$\xrightarrow{n,(lll)^*}$ (letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to (\lambda w.w)) \ldots (p_N \to (\lambda w.w)))))$,
$v' = (\texttt{case}\ z'\ (p_1 \to (\lambda w.w)) \ldots (p_N \to (\lambda w.w)))$,
$x' = \texttt{True}, z' = z$;
in $(\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))$)

$\xrightarrow{n,IOr,?}$ Ein IO-Paar wird benötigt!

$[\![t]\!] \equiv$ (letrec
$z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$
in $(\texttt{letrec}\ v' = (f\ \texttt{True})\ \texttt{in}\ (\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'})))$)

$\xrightarrow{n,llet}$ (letrec
$z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$v' = (f\ \texttt{True})$
in $(\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))$)

$\xrightarrow{n,cp}$ (letrec
$z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$v' = ((\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}$
$(\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))\ \texttt{True})$
in $(\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))$)

$\xrightarrow{n,lbeta}$ (letrec
$z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$v' = (\texttt{letrec}\ x = \texttt{True}\ \texttt{in}$
$(\lambda y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y))))))$
in $(\texttt{case}\ v'\ (p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'}))$)

$\xrightarrow{n,lll}$ (letrec
$z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$v' = (\lambda y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$x = \texttt{True}$
in (case v' $(p_1 \to \texttt{'a'}) \ldots (p_N \to \texttt{'a'})$))

$\xrightarrow{n,case}$ (letrec
$z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$v' = (\lambda y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$x = \texttt{True}$
in (letrec {} in 'a'))

$\xrightarrow{n,lll}$ (letrec
$z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))$,
$f = (\lambda x\ y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$v' = (\lambda y.(\texttt{letrec}\ z' = z\ \texttt{in}\ (\texttt{case}\ z'\ (p_1 \to ((\lambda w.w)\ y)) \ldots (p_N \to ((\lambda w.w)\ y)))))$,
$x = \texttt{True}$
in 'a')

A.11 Berechnung zu Beispiel 4.45

Sei $P = \{\mathcal{B}, \texttt{'c'}\}$, dann gilt $[\![t]\!]\Downarrow(P)$ und $[\![s]\!]\Uparrow(P)$.

$[\![s]\!] \equiv$ (letrec $f = \lambda x.$(letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)$ in z)
in (letrec $y' = (f\ \texttt{'a'})$ in (case y' $(pat_1 \to (f\ \texttt{'b'})) \ldots (pat_N \to (f\ \texttt{'b'}))$)))

$\xrightarrow{n,llet}$ (letrec $f = \lambda x.$(letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)$ in z), $y' = (f\ \texttt{'a'})$
in (case y' $(pat_1 \to (f\ \texttt{'b'})) \ldots (pat_N \to (f\ \texttt{'b'}))$))

$\xrightarrow{n,cp}$ (letrec $f = \lambda x.$(letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)$ in z),
$y' = (\lambda x'.$(letrec $z' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)$ in z') 'a'
in (case y' $(pat_1 \to (f\ \texttt{'b'})) \ldots (pat_N \to (f\ \texttt{'b'}))$))

$\xrightarrow{n,(lll)^*}$ (letrec $f = \lambda x.$(letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)$ in z),
$x' = \texttt{'a'}, z' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots), y' = z'$
in (case y' $(pat_1 \to (f\ \texttt{'b'})) \ldots (pat_N \to (f\ \texttt{'b'}))$))

$\xrightarrow{n,IOr,(\mathcal{B},\texttt{'c'})}$ (letrec $f = \lambda x.$(letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)$ in z),
$x' = \texttt{'a'}, z' = (\texttt{case}\ (\texttt{'c'})\ \ldots), y' = z'$
in (case y' $(pat_1 \to (f\ \texttt{'b'})) \ldots (pat_N \to (f\ \texttt{'b'}))$))

$\xrightarrow{n,case}$ (letrec $f = \lambda x.$(letrec $z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)$ in z),
$x' = \texttt{'a'}, z' = (\texttt{letrec}\ \{\}\ \texttt{in}\ \texttt{'c'}), y' = z'$
in (case y' $(pat_1 \to (f\ \texttt{'b'})) \ldots (pat_N \to (f\ \texttt{'b'}))$))

$$\xrightarrow{n,llet} (\texttt{letrec}\ f = \lambda x.(\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)\ \texttt{in}\ z), x' = \texttt{'a'}, z' = \texttt{'c'}, y' = z'\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,case} (\texttt{letrec}\ f = \lambda x.(\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)\ \texttt{in}\ z), x' = \texttt{'a'}, z' = \texttt{'c'}, y' = z'\ \texttt{in}\ (\texttt{letrec}\ \{\}\ \texttt{in}\ (f\ \texttt{'b'})))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ f = \lambda x.(\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)\ \texttt{in}\ z), x' = \texttt{'a'}, z' = \texttt{'c'}, y' = z'\ \texttt{in}\ (f\ \texttt{'b'}))$$

$$\xrightarrow{n,cp} (\texttt{letrec}\ f = \lambda x.(\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)\ \texttt{in}\ z), x' = \texttt{'a'}, z' = \texttt{'c'}, y' = z'\ \texttt{in}\ (\lambda x''.(\texttt{letrec}\ z'' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)\ \texttt{in}\ z')\ \texttt{'b'}))$$

$$\xrightarrow{n,(lll)^*} (\texttt{letrec}\ f = \lambda x.(\texttt{letrec}\ z = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)\ \texttt{in}\ z), x' = \texttt{'a'}, z' = \texttt{'c'}, y' = z', x'' = \texttt{'b'}, z'' = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ \ldots)\ \texttt{in}\ z'')$$

$\xrightarrow{n,IOr,?}$ Ein zweites IO-Paar wird benötigt!

$$[\![t]\!] \equiv (\texttt{letrec}\ f = (\texttt{letrec}\ z\ = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ldots)\ \texttt{in}\ \lambda x.z)\ \texttt{in}\ (\texttt{letrec}\ y' = (f\ \texttt{'a'})\ \texttt{in}\ (case\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'})))))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ f = (\texttt{letrec}\ z\ = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ldots)\ \texttt{in}\ \lambda x.z), y' = (f\ \texttt{'a'})\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ f = (\lambda x.z), z\ = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ldots)y' = (f\ \texttt{'a'})\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,cp} (\texttt{letrec}\ f = (\lambda x.z), z\ = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ldots)y' = ((\lambda x'.z)\ \texttt{'a'})\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,(lll)^*} (\texttt{letrec}\ f = (\lambda x.z), z\ = (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ldots)y' = z, x' = \texttt{'a'}\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,IOr,(\mathcal{B},\texttt{'c'})} (\texttt{letrec}\ f = (\lambda x.z), z\ = (\texttt{case 'c'}\ \ldots)y' = z, x' = \texttt{'a'}\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,case} (\texttt{letrec}\ f = (\lambda x.z), z\ = (\texttt{letrec}\ \{\}\ \texttt{in 'c'})y' = z, x' = \texttt{'a'}\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ f = (\lambda x.z), z\ = \texttt{'c'}y' = z, x' = \texttt{'a'}\ \texttt{in}\ (\texttt{case}\ y'\ (pat_1 \to (f\ \texttt{'b'}))\ldots(pat_N \to (f\ \texttt{'b'}))))$$

$$\xrightarrow{n,case} (\texttt{letrec}\ f = (\lambda x.z), z\ = \texttt{'c'}y' = z, x' = \texttt{'a'}\ \texttt{in}\ (\texttt{letrec}\ \{\}\ \texttt{in}\ (f\ \texttt{'b'})))$$

$$\xrightarrow{n,llet} (\texttt{letrec}\ f = (\lambda x.z), z\ = \texttt{'c'}y' = z, x' = \texttt{'a'}\ \texttt{in}\ (f\ \texttt{'b'}))$$

$$\xrightarrow{n,cp} (\texttt{letrec}\ f = (\lambda x.z), z\ = \texttt{'c'}y' = z, x' = \texttt{'a'}\ \texttt{in}\ ((\lambda x''.z)\ \texttt{'b'}))$$

$$\xrightarrow{n,(lll)^*} (\texttt{letrec}\ f = (\lambda x.z), z\ = \texttt{'c'}y' = z, x' = \texttt{'a'}, x'' = \texttt{'b'}\ \texttt{in}\ z$$

A.12 Berechnung zu Abschnitt 4.4.3.4

$$
\begin{aligned}
[\![s]\!] \;&= (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),\\
&\qquad\quad \texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ [\![e']\!])\\
&\qquad \texttt{in } (f\ \texttt{dictMyType})((f\ \texttt{dictMyType})\ [\![y]\!]))\\
&\xrightarrow{cp} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),\\
&\qquad\quad \texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ [\![e']\!])\\
&\qquad \texttt{in } ((\lambda dict'.(\texttt{case } dict'\ ([\![\texttt{MyClassDict}]\!]\ f_1' \to f_1')\ldots))\ \texttt{dictMyType})((f\ \texttt{dictMyType})\ [\![y]\!]))\\
&\xrightarrow{(lll)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),\\
&\qquad\quad \texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ [\![e']\!]),\\
&\qquad\quad dict' = \texttt{dictMyType}\\
&\qquad \texttt{in } ((\texttt{case } dict'\ ([\![\texttt{MyClassDict}]\!]\ f_1' \to f_1')\ldots)((f\ \texttt{dictMyType})\ [\![y]\!]))\\
&\xrightarrow{case} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),\\
&\qquad\quad \texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ g),\\
&\qquad\quad g = [\![e']\!],\\
&\qquad\quad dict' = \texttt{dictMyType}\\
&\qquad \texttt{in } ((\texttt{letrec } f_1' = g\ \texttt{in } f_1')((f\ \texttt{dictMyType})\ [\![y]\!]))\\
&\xrightarrow{(lll)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),\\
&\qquad\quad \texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ g),\\
&\qquad\quad g = [\![e']\!],\\
&\qquad\quad f_1' = g,\\
&\qquad\quad dict' = \texttt{dictMyType}\\
&\qquad \texttt{in } (f1'\ ((f\ \texttt{dictMyType})\ [\![y]\!]))\\
&\xrightarrow{cp} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),\\
&\qquad\quad \texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ g),\\
&\qquad\quad g = [\![e']\!],\\
&\qquad\quad f_1' = g,\\
&\qquad\quad dict' = \texttt{dictMyType}\\
&\qquad \texttt{in } (f1'\ (((\lambda dict''.(\texttt{case } dict''\ ([\![\texttt{MyClassDict}]\!]\ f_1'' \to f_1'')\ldots))\ \texttt{dictMyType})\ [\![y]\!]))\\
&\xrightarrow{(lll)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),\\
&\qquad\quad \texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ g),\\
&\qquad\quad g = [\![e']\!],\\
&\qquad\quad f_1' = g,\\
&\qquad\quad dict' = \texttt{dictMyType},\\
&\qquad\quad dict'' = \texttt{dictMyType}\\
&\qquad \texttt{in } (f1'\ ((\texttt{case } dict''\ ([\![\texttt{MyClassDict}]\!]\ f_1'' \to f_1'')\ldots))\ [\![y]\!]))
\end{aligned}
$$

$$\xrightarrow{case} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),$$
$$\texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ h),$$
$$g = [\![e']\!],$$
$$f_1' = g,$$
$$dict' = \texttt{dictMyType},$$
$$dict'' = \texttt{dictMyType},$$
$$h = g,$$
$$\texttt{in } (f1'\ ((\texttt{letrec } f_1'' = h\ \texttt{in} f_1'')\ [\![y]\!]))$$

$$\xrightarrow{(lll)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),$$
$$\texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ h),$$
$$g = [\![e']\!],$$
$$f_1' = g,$$
$$dict' = \texttt{dictMyType},$$
$$dict'' = \texttt{dictMyType},$$
$$h = g,$$
$$f_1'' = h,$$
$$\texttt{in } (f1'\ (f_1''\ [\![y]\!]))$$

$$\xrightarrow{(cpx)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),$$
$$\texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ h),$$
$$g = [\![e']\!],$$
$$f_1' = g,$$
$$dict' = \texttt{dictMyType},$$
$$dict'' = \texttt{dictMyType},$$
$$h = g,$$
$$f_1'' = h,$$
$$\texttt{in } (g\ (g\ [\![y]\!]))$$

$$\xrightarrow{(gc)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),$$
$$\texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ h),$$
$$g = [\![e']\!],$$
$$\texttt{in } (g\ (g\ [\![y]\!]))$$

$$\xleftarrow{(gc)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ ([\![\texttt{MyClassDict}]\!]\ f_1 \to f_1)\ldots)),$$
$$\texttt{dictMyType} = ([\![\texttt{MyClassDict}]\!]\ g),$$
$$g = [\![e']\!],$$
$$f_1''' = g,$$
$$f' = f_1''',$$
$$dict''' = \texttt{dictMyType}$$
$$\texttt{in } (g\ (g\ [\![y]\!]))$$

$$
\begin{aligned}
&\xleftarrow{(cpx)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ (\llbracket \texttt{MyClassDict} \rrbracket\ f_1 \to f_1)\ldots)),\\
&\qquad\qquad \texttt{dictMyType} = (\llbracket \texttt{MyClassDict} \rrbracket\ g),\\
&\qquad\qquad g = \llbracket e' \rrbracket,\\
&\qquad\qquad f_1''' = g,\\
&\qquad\qquad f' = f_1''',\\
&\qquad\qquad dict''' = \texttt{dictMyType}\\
&\qquad\ \texttt{in } (f'\ (f'\ \llbracket y \rrbracket)))\\
&\xleftarrow{(lll)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ (\llbracket \texttt{MyClassDict} \rrbracket\ f_1 \to f_1)\ldots)),\\
&\qquad\qquad \texttt{dictMyType} = (\llbracket \texttt{MyClassDict} \rrbracket\ g),\\
&\qquad\qquad g = \llbracket e' \rrbracket,\\
&\qquad\qquad f' = (\texttt{letrec } f_1''' = g\ \texttt{in } f_1'''),\\
&\qquad\qquad dict''' = \texttt{dictMyType}\\
&\qquad\ \texttt{in } (f'\ (f'\ \llbracket y \rrbracket)))\\
&\xleftarrow{case} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ (\llbracket \texttt{MyClassDict} \rrbracket\ f_1 \to f_1)\ldots)),\\
&\qquad\qquad \texttt{dictMyType} = (\llbracket \texttt{MyClassDict} \rrbracket\ \llbracket e \rrbracket),\\
&\qquad\qquad f' = (\texttt{case } dict'''\ (\llbracket \texttt{MyClassDict} \rrbracket\ f_1''' \to f_1''')\ldots)),\\
&\qquad\qquad dict''' = \texttt{dictMyType}\\
&\qquad\ \texttt{in } (f'\ (f'\ \llbracket y \rrbracket)))\\
&\xleftarrow{(lll)^*} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ (\llbracket \texttt{MyClassDict} \rrbracket\ f_1 \to f_1)\ldots)),\\
&\qquad\qquad \texttt{dictMyType} = (\llbracket \texttt{MyClassDict} \rrbracket\ \llbracket e \rrbracket),\\
&\qquad\qquad f' = ((\lambda dict'''.(\texttt{case } dict'''\ (\llbracket \texttt{MyClassDict} \rrbracket\ f_1''' \to f_1''')\ldots))\ \texttt{dictMyType})\\
&\qquad\ \texttt{in } (f'\ (f'\ \llbracket y \rrbracket)))\\
&\xleftarrow{cp} (\texttt{letrec } f = (\lambda dict.(\texttt{case } dict\ (\llbracket \texttt{MyClassDict} \rrbracket\ f_1 \to f_1)\ldots)),\\
&\qquad\qquad \texttt{dictMyType} = (\llbracket \texttt{MyClassDict} \rrbracket\ \llbracket e \rrbracket),\\
&\qquad\qquad f' = (f\ \texttt{dictMyType})\\
&\qquad\ \texttt{in } (f'\ (f'\ \llbracket y \rrbracket)))\\
&\equiv \llbracket t \rrbracket
\end{aligned}
$$

Anhang B

Einige Programme

B.1 Beispielprogramm aus Kapitel 2

Im Folgenden sind drei Varianten des Programms aus Beispiel 2.2 angegeben. Die Programme unterscheiden sich darin, dass mittels Pragmas (siehe Abschnitt 2.3.3) Inlining für die Bindungen angewendet werden soll oder nicht. Das dritte Programm benutzt keine Pragmas und überlässt die Entscheidung dem Compiler. Im Anschluss daran präsentieren wir die Ergebnisse der einzelnen Programme, wobei zusätzlich auch mit HasFuse compiliert wurde.

Explizite INLINING-Pragmas für alle Bindungen

```
*********************************************************************
Datei: explWithInl.lhs
*********************************************************************

> module Main(main) where
> import System.IO
> import System.IO.Unsafe

> data Pack a = C a

> {-# fglasgow-exts #-}
> {-# INLINE unpack #-}
> unpack (C x) = x

> {-# INLINE a #-}
> a = unsafePerformIO getChar

> {-# INLINE b #-}
> b x = unsafePerformIO getChar

> {-# INLINE c #-}
> c = C (unsafePerformIO getChar)

> main = do
```

```
>         putChar a
>         putChar a
>         putChar (b True)
>         putChar (b True)
>         putChar (unpack c)
>         putChar (unpack c)
>         putChar '\n'
```

Explizite NOINLINING-Pragmas für alle Bindungen

```
**********************************************************************
Datei: explWithNoInl.lhs
**********************************************************************

> module Main(main) where
> import System.IO
> import System.IO.Unsafe

> data Pack a = C a

> {-# fglasgow-exts #-}
> {-# NOINLINE unpack #-}
> unpack (C x) = x

> {-# NOINLINE a #-}
> a = unsafePerformIO getChar

> {-# NOINLINE b #-}
> b x = unsafePerformIO getChar

> {-# NOINLINE c #-}
> c = C (unsafePerformIO getChar)

> main = do
>         putChar a
>         putChar a
>         putChar (b True)
>         putChar (b True)
>         putChar (unpack c)
>         putChar (unpack c)
>         putChar '\n'
```

Ohne Pragmas

```
**********************************************************************
Datei: explWithOutInl.lhs
**********************************************************************

> module Main(main) where
> import System.IO
```

```
> import System.IO.Unsafe

> data Pack a = C a
> unpack (C x) = x

> a = unsafePerformIO getChar
> b x = unsafePerformIO getChar
> c = C (unsafePerformIO getChar)

> main = do
>          putChar a
>          putChar a
>          putChar (b True)
>          putChar (b True)
>          putChar (unpack c)
>          putChar (unpack c)
>          putChar '\n'
```

B.1.1 Ergebnisse

Tabelle B.1 zeigt eine ausführliche Übersicht der Ergebnisse der Ausführung. Das

Compiler	no-cse Flag	Pragma	Optimierungsstufe 0	1	2
			aa bb cc	aa bb cc	aa bb cc
GHC	nein	INLINE	11 23 45	11 11 11	11 11 11
GHC	nein	NOINLINE	11 23 44	11 23 44	11 11 11
GHC	nein	–	11 23 44	11 11 11	11 11 11
GHC	ja	INLINE	11 23 45	11 23 45	11 23 45
GHC	ja	NOINLINE	11 23 44	11 11 11	11 11 11
GHC	ja	–	11 23 44	11 22 33	11 22 33
HasFuse	nein	INLINE	11 23 45	11 23 45	11 23 45
HasFuse	nein	NOINLINE	11 23 44	11 23 44	11 23 44
HasFuse	nein	–	11 23 44	11 23 44	11 23 44

Tabelle B.1: Sämtliche Ergebnisse der Ausführung des Programms

für die FUNDIO-Semantik korrekte Ergebnis ist 11 23 44. Der modifizierte Compiler liefert nur dann ein falsches Ergebnis, wenn INLINING-Pragmas benutzt werden, aber solche Pragmas sollten nur dann benutzt werden, wenn der Programmierer dafür garantiert, dass Inlining für die entsprechende rechte Seite der Bindung bzgl. der FUNDIO-Semantik ein korrekte Programmtransformation ist. Dies ist für dieses Beispiel nicht gegeben, daher ist das falsche Ergebnis ein Benutzerfehler und kein Compilerfehler.

B.2 Beispiele zu einzelnen Transformationen

Im Folgenden werden einige Programm angegeben, die mit dem nicht modifizierten GHC nicht $[\![\cdot]\!]$-korrekt compiliert werden.

case-Eta-Expansion

Das folgende Programm wird aufgrund der nicht $[\![\cdot]\!]$-korrekten case-Eta-Expansion falsch transformiert:

```
*****************************************************************
Datei: etacase.lhs
*****************************************************************

> module Main(main) where
> import System.IO.Unsafe

> z = unsafePerformIO (putStr "Dieser Text sollte erscheinen!\n")
> f = \x -> z `seq` (\y -> y)

> main = (f True) `seq` return ()
```

Der Operator `seq a b` wird in die Kernsprache des GHC als `case` a `of` y `->` b übersetzt. Deshalb ist das korrekte Verhalten des Programms bezüglich der FUNDIO-Semantik derart, dass das Programm erst nach Ausgabe des Texts beendet wird.

Das mit dem GHC erstellte Programm, weist das korrekte Verhalten nur dann auf, wenn die Eta-Expansion explizit abgeschaltet wird:

```
ghc -O0 -o etacase etacase.lhs
./etacase

ghc -fno-do-lambda-eta-expansion -O0 -o etacase etacase.lhs
./etacase
Dieser Text sollte erscheinen!
```

HasFuse compiliert den Code korrekt:

```
ghc-inplace -O0 -o etacase etacase.lhs
 _   _           _____
| | | | __ _ ___|  ___|   _ ___  ___  A modified version of GHC, version 5.04.3
| |_| |/ _` / __| |_ | | | / __|/ _ \ Type --hasfuse for details
|  _  | (_| \__ \  _|| |_| \__ \  __/ This software comes with
|_| |_|\__,_|___/_|   \__,_|___/\___| ABSOLUTELY NO WARRANTY!

./etacase
Dieser Text sollte erscheinen!
```

Dies gilt auch, wenn die Optimierungsstufe 1 oder 2 benutzt wird.

full-laziness

Folgendes Programm zeigt ein nicht $[\![\cdot]\!]$-korrektes Verhalten, wenn die „full-laziness“-Transformation durchgeführt wird.

```
****************************************************************
Datei: fulllazy.lhs
****************************************************************

> module Main(main) where
> import System.IO.Unsafe(unsafePerformIO)

> main = let f = \xs -> let z = unsafePerformIO getChar
>                       in z
>        in
>         do
>           putStr ( (f 1):" ist das Ergebnis des Aufrufs von (f 1).\n" )
>           putStr ( (f 2):" ist das Ergebnis des Aufrufs von (f 2).\n" )
```

Wird das Programm mit dem GHC in Optimierungsstufe 1 aufgerufen, so wird nur ein Zeichen von der Standardeingabe gelesen:

```
ghc -O1 -o fulllazy fulllazy.lhs
./fulllazy
AB
A ist das Ergebnis des Aufrufs von (f 1).
A ist das Ergebnis des Aufrufs von (f 2).
```

Bei Compilierung mit HasFuse werden zwei Zeichen von der Standardeingabe gelesen.

```
ghc-inplace -O1 -o fulllazy fulllazy.lhs
 _   _           _____
| | | | __ _ ___|  ___|   _ ___  ___  A modified version of GHC, version 5.04.3
| |_| |/ _‘ / __| |_ | | | / __|/ _ \ Type --hasfuse for details
|  _  | (_| \__ \  _|| |_| \__ \  __/ This software comes with
|_| |_|\__,_|___/_|   \__,_|___/\___| ABSOLUTELY NO WARRANTY!

./fulllazy
AB
A ist das Ergebnis des Aufrufs von (f 1).
B ist das Ergebnis des Aufrufs von (f 2).
```

Dies ist auch dann der Fall, wenn in Optimierungsstufe 0 oder 2 compiliert wird.

B.3 Ein Programm mit merkwürdigem Verhalten

Das folgende (nichtterminierende) Programm sollte bezüglich der FUNDIO-Normalordnung einmal den Text „test“ ausgeben und anschließend wird die Bindung für `f` durch `f = f` aktualisiert. Danach sollte das Programm in eine Endlosschleife gehen, ohne weitere Ein-/Ausgaben zu tätigen:

```
*****************************************************************
Datei: infiniteLoop.lhs
*****************************************************************

> module Main(main,f) where
> import System.IO.Unsafe

> main = seq f (return ())

> f = case unsafePerformIO (print "test") of
>       () -> f
```

Allerdings weist das mit dem GHC oder HasFuse übersetzte Programm ein anderes Verhalten auf: Der Text „test“ wird endlos oft ausgegeben. Dieses Verhalten ist bezüglich der kontextuellen Gleichheit nicht falsch, denn nichtterminierende Programme mit unterschiedlichem IO-Verhalten werden durch diese nicht unterschieden.

Noch merkwürdiger ist jedoch, dass sich das Programm wie erwartet verhält, wenn `f` nicht exportiert wird. Untersuchungen der Debug-Ausgaben weisen daraufhin, dass das merkwürdige Verhalten im Back-End des Compiler entsteht, die entsprechende Position konnte jedoch nicht aufgespürt werden.

Literaturverzeichnis

[ABB+99] Augustsson, Lennart ; Barton, Dave ; Boutel, Brian ; Burton, Warren ; Fasel, Joseph ; Hammond, Kevin ; Hinze, Ralf ; Hudak, Paul ; Johnsson, Thomas ; Jones, Mark ; Launchbury, John ; Meijer, Erik ; Peterson, John ; Reid, Alastair ; Runciman, Colin ; Wadler, Philip ; Peyton Jones, Simon (Hrsg.) ; Hughes, John (Hrsg.). *Report on the Programming Language Haskell 98 A Non-strict, Purely Functional Language*. 1999

[Apt] Apt, Andrew T. *An External Representation for the GHC Core Language (DRAFT for GHC5.02)*

[AS98] Ariola, Zena M. ; Sabry, Amr: Correctness of monadic state: An imperative call-by-need calculus. In: *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, 1998, S. 62–74

[Ave95] Avenhaus, Jürgen: *Reduktionssysteme*. Springer, 1995

[Bar84] Barendregt, H.P.: *Studies in Logics and the Foundations of Mathematics*. Bd. 103: *The Lambda Calculus, Its Syntax and Semantics, Revised Edition*. North Holland, Amsterdam, The Netherlands, 1984

[BGP] Baker-Finch, Clem ; Glynn, Kevin ; Peyton Jones, Simon. *Constructed Product Result Analysis for Haskell*. http://research.microsoft.com/Users/simonpj/Papers/cpr/

[BW88] Bird, Richard ; Wadler, Phil: *Introduction to Functional Programming*. Prentice-Hall, 1988

[CFM+02] Chakravarty, Manuel M. T. ; Finne, Sigbjorn ; Marlow, Simon ; Peyton Jones, Simon ; Seward, Julian ; Thomas, Reuben. *The Glasgow Haskell Compiler (GHC) Commentary v0.13*. http://www.cse.unsw.edu.au/∼chak/haskell/ghc/comm. 2002

[Cha03] Chakravarty(editor), Manuel. *The Haskell 98 Foreign Function Interface 1.0 An Addendum to the Haskell 98 Report (Release Candidate 11)*. http://www.cse.unsw.edu.au/∼chak/haskell/ffi/. 2003

[Chi98] Chitil, Olaf: Common Subexpressions Are Uncommon in Lazy Functional Languages. In: Clack, Chris (Hrsg.) ; Hammond, Kevin (Hrsg.) ; Davie, Antony J. T. (Hrsg.): *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 10-12, 1997, Selected Papers* Bd. 1467, Springer, 1998, S. 53–71

[Gil96] Gill, Andrew: *Cheap Deforestation for Non-strict Functional Languages*, Glasgow University, Department of Computing Science, Diss., 1996

[HS89] Hudak, P. ; Sundaresh, R.S.: On the Expressiveness of Purely Functional I/O Systems / Yale University. New Haven, Connecticut, USA, 1989. – Forschungsbericht

[Hug89] Hughes, J.: Why Functional Programming Matters. In: *Computer Journal* 32 (1989), Nr. 2, S. 98–107

[Jon94] Jones, Mark P.: Dictionary-free Overloading by Partial Evaluation. In: *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, 1994, S. 107–117

[Kut00] Kutzner, Arne: *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*, J.W.Goethe-Universität Frankfurt, Diss., 2000

[LLC99] Launchbury, John ; Lewis, Jeffrey R. ; Cook, Byron: On embedding a microarchitectural design language within Haskell. In: *Proceedings of the ACM SIGPLAN international conference on functional programming (ICFP '99)* Bd. 34(9). New York, NY, USA : ACM Press, 1999. – ISBN 1–58113–111–9, S. 60–69

[LP95] Launchbury, John ; Peyton Jones, Simon L.: State in Haskell. In: *Lisp and Symbolic Computation* 8 (1995), Nr. 4, S. 293–341

[MG01] Marlow, Simon ; Gill, Andy. *Happy User Guide.* http://haskell.org/happy. 2001

[Par93] Partain, W. *The nofib Benchmark Suite of Haskell Programs.* 1993

[Pey92] Peyton Jones, Simon L.: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. In: *Journal of Functional Programming* 2 (1992), Nr. 2, S. 127–202

[Pey01] Peyton Jones, Simon: Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Ralf Steinbruggen Tony Hoare, Manfred B. (Hrsg.): *Engieering theories of software construction.*, IOS-Press, 2001, S. 47–96

[PHH+93] Peyton Jones, Simon L. ; Hall, Cordelia V. ; Hammond, Kevin ; Partain, Will ; Wadler, Philip: The Glasgow Haskell Compiler: a Technical Overview. In: *UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele*, 1993

[PL91] Peyton Jones, Simon L. ; Launchbury, J.: Unboxed Values as First Class Citizens in a Non-strict Functional Language. In: Hughes, J. (Hrsg.): *Proceedings of the Conference on Functional Programming and Computer Architecture*. Cambridge, Massachussets, USA : Springer-Verlag LNCS523, 26–28 August 1991, S. 636–666

[PM02] Peyton Jones, Simon ; Marlow, Simon: Secrets of the Glasgow Haskell compiler inliner. In: *Journal of Functional Programming* 12 (2002), Nr. 4&5, S. 393–434

[PME99] Peyton Jones, Simon ; Marlow, Simon ; Elliot, Conal: Stretching the storage manager: weak pointers and stable names in Haskell. In: *Proc. 11th International Workshop on the Implementation of Functional Languages*. The Netherlands : Springer-Verlag, September 7–10 1999 (LNCS)

[PP93] Peyton Jones, Simon L. ; Partain, W.: Measuring the effectiveness of a simple strictness analyser. In: O'Donnell, J. T. (Hrsg.): *Glasgow Workshop on Functional Programming 1993*, Springer-Verlag, 5–7 July 1993

[PPRS00] Pareja, C. ; Peña, R. ; Rubio, F. ; Segura, C.: Optimizing Eden by Transformation. In: Gilmore, Stephen (Hrsg.): *Trends in Functional Programming (Volume 2) . Proceedings of 2nd Scottish Functional Programming Workshop, SFP'00* Bd. 2, Intellect, 2000, S. 13–26

[PPS96] Peyton Jones, Simon ; Partain, Will ; Santos, André: Let-floating: moving bindings to give faster programs. In: *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ACM Press, 1996. – ISBN 0–89791–770–7, S. 1–12

[PS94] Peyton Jones, S. ; Santos, A.: Compilation by Transformation in the Glasgow Haskell Compiler. In: Hammond, K. (Hrsg.) ; Turner, D. N. (Hrsg.) ; Sansom, P. M. (Hrsg.): *Glasgow Workshop on Functional Programming*. Berlin, Heidelberg : Springer, 1994, S. 184–204

[PS98] Peyton Jones, Simon L. ; Santos, André L. M.: A transformation-based optimiser for Haskell. In: *Science of Computer Programming* 32 (1998), Nr. 1–3, S. 3–47

[PS00] Peña, R. ; Segura, C. *Two Non-Determinism Analyses in Eden*. Technical Report 108-00. 2000

[San95] Santos, André: *Compilation by Transformation in Non-Strict Functional Languages*, Glasgow University, Department of Computing Science, Diss., 1995

[Sch00] Schmidt-Schauss, Manfred. *Skript zur Vorlesung „Funktionale Programmierung I“ im Sommersemester 2000.* http://www.ki.informatik.uni-frankfurt.de. 2000

[Sch01] Schmidt-Schauss, Manfred. *Skript zur Vorlesung „Einführung in funktionale Programmiersprachen 2“ im Sommersemester 2001.* http://www.ki.informatik.uni-frankfurt.de. 2001

[Sch02] Schmidt-Schauss, Manfred. *Skript zur Vorlesung „Praktische Informatik I“ im Wintersemester 2001/2002.* http://www.ki.informatik.uni-frankfurt.de. 2002

[Sch03a] Schmidt-Schauss, Manfred. *FUNDIO: A Lambda-Calculus with a letrec, case, Constructors, and an IO-Interface: Approaching a theory of unsafePerformIO.* Draft from 22.06.03. 2003

[Sch03b] Schmidt-Schauss, Manfred. *Skript zur Vorlesung „Funktionale Programmierung“ im Sommersemester 2003.* http://www.ki.informatik.uni-frankfurt.de. 2003

[The03] The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 5.04.* http://haskell.cs.yale.edu/ghc/docs/5.04.3/. 2003

[Wad90] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. In: *Theoretical Computer Science* 73 (1990), Nr. 2, S. 231–248

[Wad92] Wadler, P.: Comprehending monads. In: *Mathematical Structures in Computer Science* 2 (1992), S. 461–493

[WB89] Wadler, P. ; Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, 1989. – ISBN 0–89791–294–2, S. 60–76

[WP99] Wansbrough, Keith ; Peyton Jones, Simon: Once upon a polymorphic type. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, 1999. – ISBN 1–58113–095–3, S. 15–28

Index

Symbole

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X