Free Theorems in Languages with Real-World Programming Features

Dissertation
zur
Erlangung des Doktorgrades (Dr.rer.nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Daniel Seidel

aus
Marienberg

Bonn 2013

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

Erstgutachter: Jun.-Prof. Dr. Janis Voigtländer, Bonn
Zweitgutachter: Prof. Dr. Manfred Schmidt-Schauß, Frankfurt
Tag der Promotion: 9. Oktober 2013
Erscheinungsjahr: 2013

Abstract

Free theorems, type-based assertions about functions, have become a prominent reasoning tool in functional programming languages. But their correct application requires a lot of care. Restrictions arise due to features present in implemented such languages, but not in the language free theorems were originally investigated in.

This thesis advances the formal theory behind free theorems w.r.t. the application of such theorems in non-strict functional languages such as Haskell. In particular, the impact of general recursion and forced strict evaluation is investigated. As formal ground, we employ different lambda calculi equipped with a denotational semantics.

For a language with general recursion, we develop and implement a counterexample generator that tells if and why restrictions on a certain free theorem arise due to general recursion. If a restriction is necessary, the generator provides a counterexample to the unrestricted free theorem. If not, the generator terminates without returning a counterexample. Thus, we may on the one hand enhance the understanding of restrictions and on the other hand point to cases where restrictions are superfluous.

For a language with a strictness primitive, we develop a refined type system that allows to localize the impact of forced strict evaluation. Refined typing results in stronger free theorems and therefore increases the value of the theorems. Moreover, we provide a generator for such stronger theorems.

Lastly, we broaden the view on the kind of assertions free theorems provide. For a very simple, strict evaluated, calculus, we enrich free theorems by (runtime) efficiency assertions. We apply the theory to several toy examples. Finally, we investigate the performance gain of the foldr/build program transformation. The latter investigation exemplifies the main application of our theory: Free theorems may not only ensure semantic correctness of program transformations, they may also ensure that a program transformation speeds up a program.

Überblick

Freie Theoreme sind typbasierte Aussagen über Funktionen. Sie dienen als beliebtes Hilfsmittel für gleichungsbasiertes Schließen in funktionalen Sprachen. Jedoch erfordert ihre korrekte Verwendung viel Sorgfalt. Bestimmte Sprachkonstrukte in praxisorientierten Programmiersprachen beschränken freie Theoreme. Anfängliche theoretische Arbeiten diskutieren diese Einschränkungen nicht oder nur teilweise, da sie nur einen reduzierten Sprachumfang betrachten.

In dieser Arbeit wird die Theorie freier Theoreme weiterentwickelt. Im Vordergrund steht die Verbesserung der Anwendbarkeit solcher Theoreme in praxisorientierten, „nicht-strikt" auswertenden, funktionalen Programmiersprachen, wie Haskell. Dazu ist eine Erweiterung des formalen Fundaments notwendig. Insbesondere werden die Auswirkungen von allgemeiner Rekursion und selektiv strikter Auswertung untersucht. Als Ausgangspunkt für die Untersuchungen dient jeweils ein mit einer denotationellen Semantik ausgestattetes Lambda-Kalkül.

Im Falle allgemeiner Rekursion wird ein Gegenbeispielgenerator entwickelt und implementiert. Ziel ist es zu zeigen ob und warum allgemeine Rekursion bestimmte Einschränkungen verursacht. Wird die Notwendigkeit einer Einschränkung festgestellt, liefert der Generator ein Gegenbeispiel zum unbeschränkten Theorem. Sonst terminiert er ohne ein Beispiel zu liefern. Auf der einen Seite erhöht der Generator somit das Verständnis für Beschränkungen. Auf der anderen Seite deutet er an, dass Beschränkungen teils überflüssig sind.

Bezüglich selektiv strikter Auswertung wird in dieser Arbeit ein verfeinertes Typsystem entwickelt, das den Einfluss solcher vom Programmierer erzwungener Auswertung auf freie Theoreme lokal begrenzt. Verfeinerte Typen ermöglichen stärkere, und somit für die Anwendung wertvollere, freie Theoreme. Durch einen online verfügbaren Generator stehen die Theoreme faktisch aufwandsfrei zur Verfügung.

Abschließend wird der Blick auf die Art von Aussagen, die freie Theoreme liefern können, erweitert. Für ein sehr einfaches, strikt auswertendes, Kalkül werden freie Theoreme mit Aussagen über Programmeffizienz bzgl. der Laufzeit angereichert. Die Anwendbarkeit der Theorie wird an einigen sehr einfachen Beispielen verifiziert. Danach wird die Auswirkung der foldr/build-Programmtransformation auf die Programmlaufzeit betrachtet. Diese Betrachtung steckt das Anwendungsziel ab: Freie Theoreme sollen nicht nur die semantische Korrektheit von Programmtransformationen verifizieren, sie sollen außerdem zeigen, wann Transformationen die Performanz eines Programms erhöhen.

Contents

Figures

Acknowledgments

I am deeply grateful for all the help I received while I was working on my thesis. Firstly I want to express my gratitude to my supervisor Janis Voigtländer. During the last five years he was the principal supporter of my work. He never lost patience when writing took longer than expected, was always a good contact for questions and discussion, and did the best to provide me continuously with new work contracts. Secondly I want to thank Manfred Schmidt-Schauß who accepted to read and review this book.

My research was mainly funded by the Deutsche Forschungsgemeinschaft (DFG). For seven month I also received a studentship from the Rheinische Friedrich-Wilhelms-Universität Bonn. Thanks for the financial support.

During my time as researcher I won a good friend, Jan Christiansen. We had fruitful discussions, joint research projects, and a lot of fun. Thanks for the time we spent together and for your serious support of my thesis! I hope our friendship persists.

I am also much obliged to Jens Behley and Jenny Balfer, who made work feel like home. For requests for coffee, technical questions, personal counsel and for each impression that had to be shared: Your office was the right place to go.

Special thanks go to my parents Reinhard and Birgit Seidel. To this day they show lively interest on how I proceed and they always provide a good place to stay when I come “home”.

Moreover, I want to thank God for his guidance and all the gifts he gave to me, the ways he opened up and the shelter I found in him. I never thought that happiness lies in Bonn.

Finally, I devote this work to my grandfather Heinz Seidel. He passed away peacefully last summer at the age of ninety-seven. It was a joy spending time with him. Our conversations were always a pleasant mixture of topics of current events, touching stories of his life and funny poems. I am grateful for all the things I could learn from him and the mental guidance he provided. He always asked if I finished the thesis. Yes grandfather, now I did.

Part I

Introduction and Background

Chapter 1

Introduction

1.1 The Power of Types

Types in programming languages are a long, successful story. Pierce (2002) defines a type system as follows: *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."* This definition is by purpose very general. Type systems are beneficial in many respects: We can statically detect errors, achieve abstraction via well defined interfaces and use type annotations as program documentation. Moreover, the original reason types were introduced in programming languages (Fortran in the 1950s) was to enhance efficiency. The distinction between integers and floating point numbers allowed for specialized efficient representation of different number types.

The reason for these many benefits is the restriction a type puts on the possible semantic meanings of a phrase in a program: Type systems are good to prove *"the absence of certain program behaviors"*. Via typing rules phrases are separated into typeable ones — which are again separated by the type they are typeable to — and untypeable ones. We are only interested in typeable phrases. Thus we can think of the typing rules as a mechanism to sort out "senseless" phrases, like the addition of a number and an empty list, $1+[\,]$, by making them untypeable. But we can also use typing rules to eliminate possibly meaningful but unwanted phrases, e.g. lists with entries of different type, as $[1,[\,]]$. Furthermore, for typeable phrases the type already provides information about the phrase. Looking at functional programming languages with a type system supporting parametric polymorphism, such as ML (Milner et al., 1997), Clean (Plasmeijer and van Eekelen, 2002) or Haskell (Haskell, 2010), this holds in a very nontrivial way.

In functional programming languages one treats a program as the combination of *mathematical functions*. This treatment in particular implies the distinguishing feature of *referential transparency*, i.e., the result of a function does only depend on the function's arguments. Hence, at least in the *pure core* of functional languages, *side effects* (such as state, I/O, mutable data structures) are not allowed.

referential transparency

side effects

Parametric polymorphism means that a function acts on inputs of several types or even any possible type in the same, uniform way.[1] The simplest such polymorphic function is the identity function, id. It takes a value of arbitrary type and — whatever type it was — returns its argument unchanged. That genericity in the type can be expressed via type variables as parameter types in a function's type signature and a way to replace the variables by concrete types, i.e., instantiate the polymorphic function. Thus, we can annotate id with a type writing $id :: \alpha \to \alpha$, where "::" is read as "has type" and α is a type variable. We leave type instantiation implicit for the moment. Given id, obviously every function $g :: \tau_1 \to \tau_2$ with arbitrary argument type τ_1 and result type τ_2 satisfies $id \circ g = g \circ id$, where $\circ$ is function composition. Curiously — leaving nontermination aside — id is (seen as semantic function) *the only* function of type $\alpha \to \alpha$. Since neither a value belongs to every type nor an operation is defined for every type, a function of type $\alpha \to \alpha$ must return its input unchanged.[2] Therefore, we can also state that for every function $f :: \alpha \to \alpha$, all types τ_1, τ_2 and all functions $g :: \tau_1 \to \tau_2$ the equality $f \circ g = g \circ f$ holds. But not enough, even for more complex types we get similar statements. Consider the type $[\tau]$, the type of lists with elements of type τ, and the function $map :: (\alpha \to \beta) \to [\alpha] \to [\beta]$[3] that takes a function and a list, and applies the function to every single element in the list. In Haskell, map could be implemented via pattern matching as

parametric polymorphism

$$\begin{aligned}
&map :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&map\ g\ [\,] \qquad\quad = [\,] \\
&map\ g\ (x : xs) = (g\ x) : (map\ g\ xs)
\end{aligned}$$

Having map and a list type, by free theorems we get that for every function $f :: [\alpha] \to [\alpha]$, all types τ_1, τ_2 and functions $g :: \tau_1 \to \tau_2$ it holds that $f \circ (map\ g) = (map\ g) \circ f$.

The presented statements about functions arise as specializations of *free theorems* (Wadler, 1989). Free theorems rely on the theory of relational parametricity (Reynolds, 1983) and were originally explored in the polymorphic lambda calculus (Girard, 1972; Reynolds, 1974). Since the pure core of typed functional programming languages[4] resembles the polymorphic lambda calculus, free

ad hoc polymorphism

[1]Note that parametric polymorphism is different from *ad hoc polymorphism* where the function's behavior is dependent on the concrete type instantiation (Strachey, 2000, Section 3.6.4).

[2]Contrarily, ad hoc polymorphism would of course allow for different functions as well because the function's behavior can rely on the input's type.

[3]The $\to$ is right-associative, i.e., $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is equivalent to $(\alpha \to \beta) \to ([\alpha] \to [\beta])$.

[4]There are also untyped functional programming languages, such as the Lisp (McCarthy, 1960)

theorems can be applied to derive statements about programs in functional programming languages as well. But a requirement is that we can distinguish by type if we consider a pure function or some function using an impure feature that forbids reasoning via free theorems, or at least complicates it. And that's where we experience problems in real world programming languages.

1.2 Real World Problems

In this thesis we concentrate on how free theorems fare in implemented functional programming languages. In particular, we consider features present in the functional programming language Haskell, and not observable from types. Moreover, we try to increase the information free theorems provide. We extend them to incorporate relative assertions about costs of program parts that yield the same results. In Haskell the following features, whose usage is not observable from the type, extend the polymorphic lambda calculus. First, Haskell allows for general recursion and, therefore, nontermination. Second, Haskell provides a way to influence the evaluation strategy. By default Haskell evaluates non-strict (i.e., it evaluates expressions only if necessary to go on with the overall computation). But we can force Haskell to evaluate expressions at an arbitrarily chosen point, even if not needed to proceed with the overall computation: The polymorphically typed primitive **seq** $:: \alpha \to \beta \to \beta$ evaluates its first argument and returns the second argument only if this evaluation succeeds, i.e., is non-erroneous and terminating. Both features are very useful: general recursion really extends the expressiveness and selective strict evaluation can improve efficiency if employed properly. Unfortunately, both features also constrain the power of free theorems. They enforce side conditions.

Let us briefly validate the need of restrictions on an example from above. We assert the following statement as verified by a free theorem:

$$\begin{array}{c}\forall f :: [\alpha] \to [\alpha], \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, xs :: [\tau_1]. \\ f\ (map\ g\ xs) = map\ g\ (f\ xs)\end{array} \tag{1.1}$$

By general recursion, embodied via the fixpoint primitive **fix** $:: (\alpha \to \alpha) \to \alpha$, we can generate a nonterminating expression (short, $\bot$) of arbitrary type when taking the fixpoint of the identity function. If now, assuming the base type Nat of natural numbers is present, in (1.1) we choose $f = \lambda xs.[\bot]$, i.e., the constant function to a list with only $\bot$ as entry, and we furthermore choose $\tau_1 = \tau_2 = Nat$ and $g = \lambda x.42$, i.e., the constant function to 42, then the theorem states for every $xs :: [Nat]$ that $[\bot]$ is equal to $[42]$. In detail, for the left-hand

with its most known dialects Common Lisp (Graham, 1997) and Scheme (Dybvig, 2003). For them, the untyped lambda calculus resembles the core of the language.

side of the theorem's statement we have

$$\begin{aligned}
& f\ (map\ g\ xs) \\
& \quad = \quad \{ \text{ insert definition of } f \ \} \\
& (\lambda xs.[\bot])\ (map\ g\ xs) \\
& \quad = \quad \{ \text{ apply } \lambda xs.[\bot] \text{ to } map\ g\ xs \ \} \\
& [\bot]
\end{aligned}$$

And for the right-hand side we have

$$\begin{aligned}
& map\ g\ (f\ xs) \\
& \quad = \quad \{ \text{ insert definition of } f \ \} \\
& map\ g\ ((\lambda xs.[\bot])\ xs) \\
& \quad = \quad \{ \text{ apply } \lambda xs.[\bot] \text{ to } xs \ \} \\
& map\ g\ [\bot] \\
& \quad = \quad \{ \text{ insert definition of } g \ \} \\
& map\ (\lambda x.42)\ [\bot] \\
& \quad = \quad \{ \text{ apply definition of } map\text{, i.e., map } g \text{ into the list } \} \\
& [(\lambda x.42)\ \bot] \\
& \quad = \quad \{ \text{ apply } \lambda x.42 \text{ to } \bot \ \} \\
& [42]
\end{aligned}$$

The statement is obviously wrong. To regain a valid theorem we need to restrict g to range only over *strict* functions, i.e., we require $g\ x = \bot$ if $x = \bot$. This restriction was already recognized by Wadler (1989, Section 7). If **seq** can be employed in f the theorem can break again. Consider for instance the partially defined function $f = \lambda[x].\mathbf{seq}\ x\ [x]$. It is only defined for one element lists as input and returns its input if the element in the list is not $\bot$, if so it returns $\bot$. If moreover we choose types $\tau_1 = \tau_2 = Nat$ and $g = \lambda x.\bot$, the theorem yields for every $xs = [y]$ with $y \neq \bot$ that $\bot$ is equal to $[\bot]$. In detail, for the left-hand side of the theorem we have

$$\begin{aligned}
& f\ (map\ g\ xs) \\
& \quad = \quad \{ \text{ insert definitions } f \text{ and } g \text{ and take } xs = [y] \ \} \\
& (\lambda[x].\mathbf{seq}\ x\ [x])\ (map\ (\lambda x.\bot)\ [y]) \\
& \quad = \quad \{ \text{ map } \lambda x.\bot \text{ into } [y] \text{ and apply it to } y \ \} \\
& (\lambda[x].\mathbf{seq}\ x\ [x])\ [\bot] \\
& \quad = \quad \{ \text{ apply } \lambda[x].\mathbf{seq}\ x\ [x] \text{ to } [\bot] \ \} \\
& \mathbf{seq}\ \bot\ [\bot] \\
& \quad = \quad \{ \text{ definition of } \mathbf{seq} \ \} \\
& \bot
\end{aligned}$$

And for the right-hand side we have

$$\begin{aligned}
& \mathit{map}\ g\ (f\ xs) \\
=\ & \quad \{ \text{ insert definitions } f \text{ and } g \text{ and take } xs = [y] \ \} \\
& \mathit{map}\ (\lambda x.\bot)\ ((\lambda[x].\mathbf{seq}\ x\ [x])\ [y]) \\
=\ & \quad \{ \text{ apply } \lambda[x].\mathbf{seq}\ x\ [x] \text{ to } [y] \text{ with } y \neq \bot \ \} \\
& \mathit{map}\ (\lambda x.\bot)\ [y] \\
=\ & \quad \{ \text{ apply definition of } \mathit{map} \ \} \\
& [(\lambda x.\bot)\ y] \\
=\ & \quad \{ \text{ apply } \lambda x.\bot \text{ to } y \ \} \\
& [\bot]
\end{aligned}$$

The statement is again wrong. Counterexamples to the free theorem, as the just given one, that arise through the use of **seq** are ruled out by restricting g to be total, i.e., $g\ x = \bot$ only if $x = \bot$, as proven by Johann and Voigtländer (2004).

What becomes apparent from the above — even though artificial — examples is that to safely use free theorems the effects of programming features must be explored diligently. The investigation of a new feature, say $\mathcal{F}$, can take place in four phases.

First, an intuition about what problems $\mathcal{F}$ causes w.r.t. free theorems is desirable. Therefore, we look for counterexamples, as the ones above. Hence, the first phase will be to *manually search for counterexamples* to the "naive" versions of free theorems that do not consider any influence of $\mathcal{F}$. These counterexamples hopefully yield necessary extra conditions to regain correct free theorems in the extended setting.

Second, a *formal proof* has to be made ensuring that the discovered conditions are sufficient to regain free theorems. We need, as elaborated later, a proof of the parametricity theorem w.r.t. a logical relation that is restricted according to the investigated setting. The parametricity theorem, first stated by Reynolds (1983), is the basis to derive free theorems.

At the end of phase two we are able to derive correct free theorems even in the presence of the new feature $\mathcal{F}$. Now we could announce success because free theorems are available in the extended calculus. But the price is still high. The new conditions on the theorems may often be superfluous for special instances of a theorem. Hence, we move on to phase three to remedy that situation.

In the third phase we *localize the influence of* $\mathcal{F}$ by a refined type system. If it is ensured that a polymorphic function f we want to apply the free theorem to does not use the new feature $\mathcal{F}$, the newly introduced restrictions may be unnecessary. Hence, we would benefit, in the sense of stronger free theorems, if functions having currently the same type but differing in the use of $\mathcal{F}$ could yield differently restricted free theorems. Since free theorems only rely on the type of a function, the use of $\mathcal{F}$ must be reflected in the type somehow. Therefore, the only way is a refined type system where the use of $\mathcal{F}$ leaves a mark in the type and thus functions employing $\mathcal{F}$ can be distinguished from the ones that do not and allow for more liberal free theorems.

In a last phase, the focus is on the *automatic generation of counterexamples* to the naive free theorems that are not adjusted to $\mathcal{F}$. This automatic search, if implemented as a tool, can help to properly understand effects arising from the new features, in particular, it may help people applying free theorems. Furthermore, there are types where the restrictions introduced due to $\mathcal{F}$ are never necessary, even if the refined type tells us so. If the automatic counterexample search is complete and returns without a counterexample, such types are identified.

An overview of the current development in the theory of free theorems w.r.t. additional language features, as well as of applications of the theorems, is given in Chapter 3.

1.3 The Contributions of the Thesis

In this thesis we will push the development of the theory of free theorems further w.r.t. different language features:

- In Chapter 4 we consider the feature of general recursion and develop an algorithm to automatically generate counterexamples to free theorems that lack a strictness restriction — a restriction that arises if a calculus is enriched with general recursion.

- In Chapter 5 we localize the impact of selective strictness on free theorems. Therefor, we develop a refined type system to track whether a subterm of a term is forced to be strictly evaluated by a strictness primitive or not. If not, totality restrictions in the free theorem's assertion may safely be dropped.

The last chapter highlights a completely different aspect of free theorems, orthogonal to the described adaptation to new language features:

- In Chapter 6 we investigate quantitative aspects of free theorems. Free theorems state program equivalences and are a successfully applied reasoning tool, in particular for the verification of program transformations (see Section 3.2). For program transformations it is beneficial to know which one of two semantically equivalent programs is more efficient, e.g. needs fewer evaluation steps. We enrich the theory of parametricity thus, that a free theorem tells not only that two program parts are semantically equivalent, but also how the two parts differ in their (runtime) cost behavior.

1.4 The Structure of the Thesis

The thesis is structured in three parts. The first part contains this introduction. Furthermore, a formal introduction, Chapter 2, is included. It is meant to make the thesis self-contained. Chapter 2 does not only introduce the theory behind free theorems and the adaptations necessary when we extend (a sublanguage of) the polymorphic lambda calculus by general recursion and forced strict evaluation, it also explains most of the notation used in the thesis. Chapter 3 surveys related work.

The second part of the thesis presents original results. All the results are already published and at the beginning of each chapter we cite the corresponding publications via footnotes.

Each chapter comes with its own summary and outlook sections. Nevertheless, we close the thesis by Part III with a conclusion for the whole work.

Chapter 2

The Formal Background of Free Theorems

In this chapter we introduce the formal background behind free theorems. For this purpose, in Section 2.1 we set up a simply typed lambda calculus (Church, 1940; Curry and Feys, 1958) with type variables, base type Nat and lists as algebraic data type. We call the calculus λ^α and will use it with various alterations as the formal language for our investigations. As semantics, we employ a denotational semantics. Furthermore, we state the abstraction or parametricity theorem (Reynolds, 1983) for λ^α and derive free theorems from it. In Section 2.2 we add general recursion to λ^α and discuss the major changes. The adjustments that are necessary if selective strictness is added are considered in Section 2.3. In Section 2.4 we explain the choice of the type system we consider and address the extension to explicit type abstraction and instantiation.

The chapter also introduces notation and provides a lot of explanation on it that is not repeated later on when similar notation is used.

2.1 Free Theorems for the Simply Typed Lambda Calculus

2.1.1 Definition of the Simply Typed Lambda Calculus

"Underlying the formal calculi which we shall develop is the concept of a function as it appears in various branches of mathematics, either under that name or under synonymous names, 'operation' or 'transformation'."
Church (1941)

A lambda calculus (also written λ-calculus) is a mathematical calculus that formalizes the concept of functions. The terms of the pure untyped lambda calculus are established only out of variables, and two ways to form terms out of others. The first way is abstraction, called λ-abstraction, and the second way application of one term to another. An abstraction is written as $\lambda x.t$ where x is a variable and t an arbitrary term. By $\lambda x.t$ the occurrences of the variable x in t are bound. We regard $\lambda x.t$ as an anonymous function that takes a parameter x that might be needed to evaluate the function body t. When we apply a term $\lambda x.t$ to a term t', written as $(\lambda x.t)\ t'$, we can evaluate the application to t with all occurrences of x in t (that are not under the scope of another binder of x, i.e., another $\lambda x.\ldots$) substituted by t'. Also $x\ y$ states a valid application, but we cannot evaluate it further.

Example 1

We can encode data types in the pure untyped lambda calculus. For example, the terms $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ are possible encodings of the Boolean values $True$ and $False$.[1]

For these encodings, the term

$$\lambda m.\lambda n.m\ n\ m$$

represents the logical and.[2] If we apply it to the representation of $True$, it yields a λ-abstraction that, if applied to another term t, simply returns t. If we apply it to the representation of $False$ it yields the representation of $False$ when applied to a second term. Thus, $\lambda m.\lambda n.m\ n\ m$ corresponds to the following Haskell implementation of and:

```
and :: Bool → Bool → Bool
and True x = x
and False x = False
```

Let us validate the correspondence for the application of and to $True$ as first argument. For the representations we get

$$\begin{aligned}
&(\lambda m.\lambda n.m\ n\ m)\ (\lambda x.\lambda y.x)\\
&=\quad \{\text{ application of } (\lambda m.\lambda n.m\ n\ m) \text{ to } (\lambda x.\lambda y.x)\ \}\\
&\lambda n.(\lambda x.\lambda y.x)\ n\ (\lambda x.\lambda y.x)\\
&=\quad \{\text{ application of } (\lambda x.\lambda y.x) \text{ to } n\ \}\\
&\lambda n.(\lambda y.n)\ (\lambda x.\lambda y.x)\\
&=\quad \{\text{ application of } (\lambda y.n) \text{ to } (\lambda x.\lambda y.x)\ \}\\
&\lambda n.n
\end{aligned}$$

which is exactly what we intended.

[1] λ-abstraction expands as far to the right as possible, e.g. $\lambda x.\lambda y.x$ is equal to $\lambda x.(\lambda y.x)$.
[2] Application is left-associative, so $\lambda m.\lambda n.m\ n\ m$ is equivalent to $\lambda m.\lambda n.(m\ n)\ m$.

τ	$::=$	α	type variable
	$\mid$	$\tau \to \tau$	function type
	$\mid$	Nat	number type
	$\mid$	$[\tau]$	list type
t	$::=$	x	variable
	$\mid$	$\lambda x :: \tau . t$	abstraction
	$\mid$	$t\ t$	application
	$\mid$	n	natural number
	$\mid$	$t + t$	addition of naturals
	$\mid$	$\textbf{case}\ t\ \textbf{of}\ \{0 \to t; \_ \to t\}$	case expression for naturals
	$\mid$	$[]_\tau$	empty list
	$\mid$	$t : t$	list constructor
	$\mid$	$\textbf{case}\ t\ \textbf{of}\ \{[] \to t; x : x \to t\}$	case expression for lists

Figure 2.1: Type and term syntax of λ^α

Concerning computational strength the untyped λ-calculus is Turing-complete.[3] For an overview of the calculus, see Barendregt (1992, Section 2) or Pierce (2002, Chapter 5).

Because we are aiming at type based reasoning, we are not interested in the untyped lambda calculus, but in a typed version. That means, besides terms, we have a set of types. These types can be assigned to terms by typing rules, as we will see later on. Here, we define a simply typed lambda calculus with type variables, natural numbers and lists. We call it λ^α. The syntax definitions are presented in a style similar to the Backus-Naur-Form (BNF). In contrast to the original BNF, we do not enclose non-terminals in triangular brackets. The distinction between terminals and non-terminals is only implicit. Non-terminals are the symbols that either appear on a left-hand side of a definition or are defined to range over a certain set of other symbols. All symbols that are neither non-terminals nor belong to the BNF metalanguage (i.e., "::=" and "|") are terminals. Type and term syntax of λ^α are given in Figure 2.1, where α ranges over a countable set of *type variables*, x over a disjoint countable set of *term variables* and n over the natural numbers.

λ^α

type variable

term variable

A variable can occur bound or unbound in a term.[4]

Definition 1 (unbound occurrence of a term variable, UV(·))

We define the *set of term variables that occur unbound* in a term t, $\mathrm{UV}(t)$, inductively by

$$\mathrm{UV}(x) = \{x\}$$

[3]Note that the simply typed lambda calculus that we introduce next is not Turing-complete because type constraints prohibit general recursion.

[4]Often, unbound occurrences of variables are also called free occurrences of variables.

$$\begin{aligned}
\mathrm{UV}(\lambda x :: \tau.t) &= \mathrm{UV}(t) \setminus \{x\} \\
\mathrm{UV}(t_1\ t_2) = \mathrm{UV}(t_1 + t_2) = \mathrm{UV}(t_1 : t_2) &= \mathrm{UV}(t_1) \cup \mathrm{UV}(t_2) \\
\mathrm{UV}(n) = \mathrm{UV}([\,]_\tau) &= \emptyset \\
\mathrm{UV}(\textbf{case}\ t\ \textbf{of}\ \{0 \to t_1; \_ \to t_2\}) &= \mathrm{UV}(t) \cup \mathrm{UV}(t_1) \cup \mathrm{UV}(t_2) \\
\mathrm{UV}(\textbf{case}\ t\ \textbf{of}\ \{[\,] \to t_1; x : xs \to t_2\}) &= \mathrm{UV}(t) \cup \mathrm{UV}(t_1) \\
&\quad \cup (\mathrm{UV}(t_2) \setminus \{x, xs\})
\end{aligned}$$

We say x *occurs unbound* in t if $x \in \mathrm{UV}(t)$.

Definition 2
(bound occurrence of a term variable, $\mathrm{BV}(\cdot)$)

We define the *set of term variables that occur bound* in a term t, $\mathrm{BV}(t)$, inductively by

$$\begin{aligned}
\mathrm{BV}(x) &= \emptyset \\
\mathrm{BV}(\lambda x :: \tau.t) &= \mathrm{BV}(t) \cup \{x\} \\
\mathrm{BV}(t_1\ t_2) = \mathrm{BV}(t_1 + t_2) = \mathrm{BV}(t_1 : t_2) &= \mathrm{BV}(t_1) \cup \mathrm{BV}(t_2) \\
\mathrm{BV}(n) = \mathrm{BV}([\,]_\tau) &= \emptyset \\
\mathrm{BV}(\textbf{case}\ t\ \textbf{of}\ \{0 \to t_1; \_ \to t_2\}) &= \mathrm{BV}(t) \cup \mathrm{BV}(t_1) \cup \mathrm{BV}(t_2) \\
\mathrm{BV}(\textbf{case}\ t\ \textbf{of}\ \{[\,] \to t_1; x : xs \to t_2\}) &= \mathrm{BV}(t) \cup \mathrm{BV}(t_1) \cup \mathrm{BV}(t_2) \cup \{x, xs\}
\end{aligned}$$

We say x *occurs bound* in t if $x \in \mathrm{BV}(t)$.

Note that a variable can at the same time occur unbound *and* bound in a term. The definitions should become quite clear by the following example.

Example 2
(unbound / bound occurrences of variables)

$\lambda x :: Nat.(x + 5)$	x occurs bound
$\lambda x :: Nat.(y + 5)$	x occurs bound and y occurs unbound
$(\lambda x :: Nat.5)\ x$	x occurs unbound and bound
$x\ (\lambda x :: Nat.(x + 5))$	x occurs unbound and bound

In general, the distinction between unbound and bound occurrences of variables is also reasonable on the type level. However, since we regard only calculi without explicit type abstraction and instantiation (more details are found in Section 2.4), type variables cannot occur bound in our calculi. To emphasize that fact we state the following definition.

Definition 3
((unbound occurrence of a) type variable, $\mathrm{UTV}(\cdot)$)

The set of type variables (that occur unbound) in a type τ is denoted by $\mathrm{UTV}(\tau)$.

The second part of the calculus' definition is a set of *typing rules*, describing how to assign types to terms. Typing rules separate terms into typeable and untypeable terms, where we are only interested in the typeable ones. We can think of the typing rules as a mechanism to sort out "senseless" terms, like $1 + [\,]$, or to eliminate possibly meaningful but unwanted terms, e.g. $[1, [\,]]$.

typing rules

Via typing rules, we define if a typing judgment is valid. A *typing judgment* has the form $\Gamma \vdash t :: \tau$, telling that the term t is typeable to type τ under the typing context Γ. A *typing context* Γ is the union of a set of type variables, Γ_T, referred to as *type context*, and a set of term variables with associated type, Γ_V, referred to as *term context*. In the term context the same variable is not allowed to appear twice, even if associated to different types.

typing judgment
typing context (Γ)
type context (Γ_T)
term context (Γ_V)

CONVENTION 1 ($\Gamma = \Gamma_T \cup \Gamma_V$)

Let Γ be a typing context. We implicitly assume that Γ is the union of the type context Γ_T and term context Γ_V, and hence use Γ_T and Γ_V freely without stating the connection to Γ.

For every typing context Γ we write $\Gamma, x :: \tau$ to denote the typing context $\Gamma \cup \{x :: \tau\}$. If Γ is empty we simply write $x :: \tau$. To guarantee consistency, we forbid to add $x :: \tau$ to a typing context Γ whenever $(x :: \tau') \in \Gamma$ for some $\tau' \neq \tau$. We can also take the union of two term or typing contexts, but only gain a new term or typing context if the contexts are compatible in the following way.

DEFINITION 4 (compatible term/typing contexts)

Two term contexts Γ_V, Γ'_V are compatible if for every $(x :: \tau) \in \Gamma_V$ and $\tau' \neq \tau$, it holds that $(x :: \tau') \notin \Gamma'_V$. The notion extends canonically to typing contexts.

Concerning the typing rules discussed below, the compatibility restriction does not impose a problem. We can rename bound occurrences of variables without altering the (still to be defined) meaning of a term. The renaming is known as α-conversion and formally introduced in Lemma 2. In the following we identify terms up to α-conversion. Furthermore, we write α, Γ to denote the typing context $\Gamma \cup \{\alpha\}$, and if Γ is empty we simply write α.

The typing rules for λ^α are given in Figure 2.2. They consist of *axioms*, i.e., typing judgments like (VAR) and (NIL) that are satisfied without any precondition, and rules of the form

axiom

$$\frac{\mathit{premise}_1 \ldots \mathit{premise}_n}{\mathit{conclusion}}$$

that tell: If all n premises are fulfilled, then the conclusion holds. For example, the rule (APP) states the following: If we have a term t_1 typeable to $\tau_1 \to \tau_2$ under typing context Γ and a term t_2 typeable to τ_1 under the same context Γ, then the term $t_1\ t_2$ (which is the application of t_1 to t_2) is typeable to τ_2 under Γ. A derivation for a typing judgment, called *type derivation*, is a tree of instances of typing rules where the root is the rule that actually yields the considered typing judgment and all leaves are axioms. As an example, we present the type derivation for $\vdash (\lambda x :: \mathit{Nat}.(x + x))\ 5 :: \mathit{Nat}$, stating that without any contextual

type derivation

$$\Gamma, x :: \tau \vdash x :: \tau \ (\textsc{Var}) \qquad \Gamma \vdash []_\tau :: [\tau] \ (\textsc{Nil}) \qquad \Gamma \vdash n :: \mathit{Nat} \ (\textsc{Nat})$$

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]} \ (\textsc{Cons}) \qquad \frac{\Gamma \vdash t_1 :: \mathit{Nat} \qquad \Gamma \vdash t_2 :: \mathit{Nat}}{\Gamma \vdash (t_1 + t_2) :: \mathit{Nat}} \ (\textsc{Sum})$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1 \to \tau_2} \ (\textsc{Abs}) \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1 \ t_2) :: \tau_2} \ (\textsc{App})$$

$$\frac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{[] \to t_1; x_1 : x_2 \to t_2\}) :: \tau} \ (\textsc{LCase})$$

$$\frac{\Gamma \vdash t :: \mathit{Nat} \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}) :: \tau} \ (\textsc{NCase})$$

Figure 2.2: Typing rules of λ^α

knowledge about types, the function application of $\lambda x :: \mathit{Nat}.(x + x)$ to 5 is typeable to Nat.

Example 3 (type derivation)

$$\frac{\dfrac{\dfrac{x :: \mathit{Nat} \vdash x :: \mathit{Nat} \qquad x :: \mathit{Nat} \vdash x :: \mathit{Nat}}{x :: \mathit{Nat} \vdash (x + x) :: \mathit{Nat}}}{\vdash (\lambda x :: \mathit{Nat}.(x + x)) :: \mathit{Nat} \to \mathit{Nat}} \qquad \vdash 5 :: \mathit{Nat}}{\vdash ((\lambda x :: \mathit{Nat}.(x + x))\ 5) :: \mathit{Nat}}$$

The reader may have noticed that we define typing contexts to contain type variables, but we do not use them at all in the typing rules. In fact, it is not necessary to keep track of type variables in the typing context, but it will come in handy later on when we state the parametricity theorem (Theorem 1) to know which type variables are present in a typing judgment. Thus, if t is typeable to τ under a typing context Γ, then all type variables appearing in a type that is associated to a term variable in Γ, in a type annotation in t, or in τ must be in Γ_T. Finally, we end up with the following definition of a valid typing judgment.

Definition 5 (valid typing judgment)

We say a typing judgment $\Gamma \vdash t :: \tau$ is *valid*[5] (or in other words t *is typeable to* τ *under the typing context* Γ) in λ^α, if

- there exists a type derivation for $\Gamma \vdash t :: \tau$ and
- all type variables that occur in a type annotation of a variable in Γ_V, a type annotation in t, or in τ are in Γ_T.

[5]Validity of a typing judgment is always w.r.t. a system of typing rules and hence specific to a calculus. Be aware of that fact later on, when we have different calculi and compare typeability.

Convention 2

When we are not interested in type variables, we write $t :: \tau$ if there exists Γ with $\Gamma_V = \emptyset$ and $\Gamma \vdash t :: \tau$ is valid.

To find a type derivation for a given term t under a typing context Γ it is essential that all term variables that occur unbound in t appear in Γ. If so, we say t is closed under Γ.

Definition 6 (closed term)

A term t is *closed under a term context* Γ_V if for every $x \in \mathrm{UV}(t)$ there exists τ such that $(x :: \tau) \in \Gamma_V$. If t is closed under the empty term context, i.e., all variables occur only bound in t, it is called *closed*.

A similar definition can be given for types. Closed types in our calculi are also called *monotypes* since they contain no type variables at all.

monotype

Definition 7 (closed type)

A type τ is *closed under a type context* (or any set of type variables) Γ_T if $\Gamma_T \supseteq \mathrm{UTV}(\tau)$. If t is closed under the empty context, it is called *closed*.

A last hint concerning type annotations is in order. In contrast to Haskell, in all λ-abstractions it is mandatory to annotate the variable whose occurrences get bound by a type. That is, we have to write $\lambda x :: \tau.t$ instead of $\lambda x.t$. Also the empty list has to be annotated by a type in λ^{α}, i.e., we write $[\,]_\tau$ instead of only $[\,]$. The different styles, whether to enforce explicit type annotations or not, are known as Church and Curry style. The choice of a language with explicit type annotations (Church style) saves us from type inference in the way that the typing rules in Figure 2.2 enjoy the *weak subformula property* w.r.t. type reconstruction. That is, we can easily deduce the type of any typeable term or recognize ill-typed terms by simply applying the typing rules backwards without having to guess some type at any stage.

Church / Curry style

weak subformula property

Ultimately, concerning the syntax, let us point out some conventions we will apply throughout this thesis to reduce the number of parentheses.

Convention 3 (operator precedence)

Convention	Example
function application is left-associative	$f\ x\ y = (f\ x)\ y$
the body of a function abstraction extends to the right as far as possible	$\lambda x.\lambda y.y\ x = \lambda x.(\lambda y.(y\ x))$
":" is right-associative	$1:2:[\,]_{Nat} = 1:(2:[\,]_{Nat})$
function application binds stronger than "+", and "+" stronger than ":"	$f\ 1{+}2{:}[\,]_{Nat} = ((f\ 1){+}2){:}[\,]_{Nat}$

$$\begin{array}{llll} [\![\alpha]\!]_\theta &= \theta(\alpha) & [\![[\tau]]\!]_\theta &= \{[\mathbf{a_1},\ldots,\mathbf{a_n}] \mid n \in \mathbb{N} \land \forall i \in \{1,\ldots,n\}.\ \mathbf{a_i} \in [\![\tau]\!]_\theta\} \\ [\![Nat]\!]_\theta &= \mathbb{N} & [\![\tau_1 \to \tau_2]\!]_\theta &= \{\mathbf{f} : [\![\tau_1]\!]_\theta \to [\![\tau_2]\!]_\theta\} \end{array}$$

Figure 2.3: Type semantics of λ^α

$$\begin{array}{ll} [\![x]\!]_\sigma &= \sigma(x) \\ [\![n]\!]_\sigma &= \mathbf{n} \\ [\![t_1 + t_2]\!]_\sigma &= [\![t_1]\!]_\sigma + [\![t_2]\!]_\sigma \\ [\![[\,]_\tau]\!]_\sigma &= [\,] \\ [\![t_1 : t_2]\!]_\sigma &= [\![t_1]\!]_\sigma : [\![t_2]\!]_\sigma \\ [\![\lambda x :: \tau.t]\!]_\sigma &= \lambda\mathbf{a}.[\![t]\!]_{\sigma[x\mapsto\mathbf{a}]} \\ [\![t_1\ t_2]\!]_\sigma &= [\![t_1]\!]_\sigma\ [\![t_2]\!]_\sigma \end{array}$$

$$[\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]_\sigma = \begin{cases} [\![t_1]\!]_\sigma & \text{if } [\![t]\!]_\sigma = \mathbf{0} \\ [\![t_2]\!]_\sigma & \text{otherwise} \end{cases}$$

$$[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1; x_1 : x_2 \to t_2\}]\!]_\sigma = \begin{cases} [\![t_1]\!]_\sigma & \text{if } [\![t]\!]_\sigma = [\,] \\ [\![t_2]\!]_{\sigma[x_1\mapsto\mathbf{a},x_2\mapsto\mathbf{b}]} & \text{if } [\![t]\!]_\sigma = \mathbf{a} : \mathbf{b} \end{cases}$$

Figure 2.4: Term semantics of λ^α

So far we have presented the complete syntax for λ^α and we can distinguish between "senseful" and "senseless" terms via typing rules. The final thing missing to complete the calculus is a semantics. We have to assign meaning to terms, at least to the ones proclaimed "senseful", i.e., the typeable ones.

axiomatic / operational / denotational semantics

In general, there are three different kinds of semantics: *axiomatic*, *operational* and *denotational*. A short overview is found in the books of Pierce (2002) and Mitchell (1996). We concentrate on denotational semantics. It assigns to every term and type directly a meaning, i.e., a mathematical object.

The standard denotational semantics for λ^α is given in Figures 2.3 and 2.4 for types and terms, respectively.

type environment (θ)

$[\mathbf{a_1},\ldots,\mathbf{a_n}]$

Let us discuss the denotational semantics for types first. We interpret types as sets. Type *Nat* is interpreted as the set $\mathbb{N}$ of natural numbers including zero and function types are interpreted as the set of functions between two sets. List types are interpreted as the free monoid over the interpretation of the list's element type. The neutral element we denote by $[\,]$, and by $:$ we denote the binary operation. Thus, the semantic interpretation of a list has the form $\mathbf{a_1} : \ldots : \mathbf{a_n} : [\,]$. For type variables we can choose any set we like as interpretation. The choice we take is described by the *type environment* θ that is a partial map from type variables to sets. Two comments are in order concerning notation. First, throughout the thesis we distinguish syntax and semantics by different fonts. While f is a term (of function type), $\mathbf{f}$ is a semantic object (a mathematical function). Second, the list notation $[\mathbf{a_1},\ldots,\mathbf{a_n}]$ is a shorthand for $\mathbf{a_1} : \ldots : \mathbf{a_n} : [\,]$. A similar abbreviation is applied on the syntactic level.

The term semantics maps terms to elements of the sets that types are mapped to. All syntactic operation symbols are mapped to mathematical operations.

Thus, the syntactic "+" becomes the usual "+" operation on the naturals and the syntactic ":" becomes a semantic one. Case expressions are translated into different objects depending on the scrutinee's semantics. Syntactic numbers become natural numbers, λ-abstractions become mathematical functions and application becomes standard mathematical function application which is, as on the syntactic level, denoted by juxtaposition (i.e., by $\mathbf{f}\ \mathbf{a}$ instead of $\mathbf{f}(\mathbf{a})$). Note also that mathematical functions are written similar to λ-abstractions, only with a slightly different lambda symbol. That is, we write $\boldsymbol{\lambda}\mathbf{a}.\mathbf{t}$ instead of $\mathbf{f}\ \mathbf{a} = \mathbf{t}$ for all $\mathbf{a}$, where $\mathbf{t}$ can depend on $\mathbf{a}$. The *term environment* σ in Figure 2.4 is a partial map from term variables to semantic values. It stores the values of variables that occur unbound. In particular, it stores the values of function arguments when evaluating the function body, as well as head and tail of lists when evaluating the second alternative of a case expression over lists. For environments, we employ the following notation: We write $\emptyset$ for the empty environment, and for any environment κ, $\kappa[x_1 \mapsto \mathbf{a_1}, \ldots, x_n \mapsto \mathbf{a_n}]$ denotes κ extended by the entries $\mathbf{a_1}, \ldots, \mathbf{a_n}$ for the variables $x_1, \ldots, x_n$, respectively. If an x_i is already in the domain of κ, the entry for x_i is overwritten. When extending the empty environment, we omit $\emptyset$. To denote the domain of an environment κ, we write $dom(\kappa)$. In the sequel we employ $dom(\cdot)$ to refer to the domain of arbitrary (mathematical) functions.

term environment (σ)

$dom(\cdot)$

To make sure that the denotational semantics is well-defined, we have to guarantee that the semantics of a term of type τ is an element of the type semantics of τ. Otherwise our mathematical interpretations are not well-defined. It could for example happen that we add values that are not natural numbers. Fortunately, the just given semantics satisfies the type restriction.

Lemma 1

If $\Gamma \vdash t :: \tau$ valid in λ^α, then for all θ and σ with

- $\Gamma_T \subseteq dom(\theta)$ and
- $(x :: \tau') \in \Gamma_V \Rightarrow \sigma(x) \in [\![\tau']\!]_\theta$

we have $[\![t]\!]_\sigma \in [\![\tau]\!]_\theta$.

Proof. Induction over the type derivation. □

As the last point in this subsection, we discuss operations on terms in λ^α that do not change the semantics of the term. To state these operations, we need syntactic substitution of term variables by terms. It is defined as follows.

Definition 8 (Substitution)

Let t, t' terms and x a term variable. The substitution of x in t by t', written $t[t'/x]$, is the syntactical replacement of all unbound occurrences of x in t by t'.

capture avoiding

Be aware that substitution, as just defined, is *capture avoiding*, i.e., bound occurrences of variables — either bound via λ-abstraction or in a case expression for lists — are not substituted.

Finally, we can state two equivalences that hold in λ^α, and also in any of the following λ-calculi.

LEMMA 2

In λ^α the following equivalences hold for every appropriate σ

$$\begin{aligned} [\![\lambda x :: \tau.t]\!]_\sigma &= [\![\lambda y :: \tau.t[y/x]]\!]_\sigma \quad \text{if } y \notin \mathrm{UV}(t) \cup \mathrm{BV}(t) && (\alpha\text{-conversion}) \\ [\![(\lambda x :: \tau.t_1)\ t_2]\!]_\sigma &= [\![t_1[t_2/x]]\!]_\sigma && (\beta\text{-conversion}) \end{aligned}$$

CONVENTION 4 (≡)

In the following we write $t \equiv t'$ to denote that $[\![t]\!]_\sigma = [\![t']\!]_\sigma$ (under each appropriate term environment σ).

Intuitively clear, but worth to be formally defined, the replacement of a subterm of a term with a semantically equivalent term does not alter the semantics of the overall term. To state that result formally, we define context and subterm. Informally, a context is a term with a hole[6]. Formally, it is defined as follows.

DEFINITION 9 (context)

A *context* $\mathcal{C}[]$ is a function from terms to terms that is defined via

$$\begin{array}{rcl} \mathcal{C}[] & ::= & [] \\ & | & \lambda x :: \tau.\mathcal{C}[] \\ & | & \mathcal{C}[]\ t \\ & | & t\ \mathcal{C}[] \\ & | & \mathcal{C}[] + t \\ & | & t + \mathcal{C}[] \\ & | & \textbf{case } \mathcal{C}[] \textbf{ of } \{0 \to t; \_ \to t\} \\ & | & \textbf{case } t \textbf{ of } \{0 \to \mathcal{C}[]; \_ \to t\} \\ & | & \textbf{case } t \textbf{ of } \{0 \to t; \_ \to \mathcal{C}[]\} \\ & | & \mathcal{C}[] : t \\ & | & t : \mathcal{C}[] \\ & | & \textbf{case } \mathcal{C}[] \textbf{ of } \{[\,] \to t; x : x \to t\} \\ & | & \textbf{case } t \textbf{ of } \{[\,] \to \mathcal{C}[]; x : x \to t\} \\ & | & \textbf{case } t \textbf{ of } \{[\,] \to t; x : x \to \mathcal{C}[]\} \end{array}$$

where t ranges over the terms and x over the term variables. The application of a context $\mathcal{C}[]$ to a term t is denoted by $\mathcal{C}[t]$ and yields the context $\mathcal{C}[]$ with $[]$ replaced by t, i.e., a term.

[6]A context may also have many holes, but we only regard one hole contexts.

A term t' is a *subterm* of term t, if there exists a context $\mathcal{C}[]$, such that $\mathcal{C}[t'] = t$.

DEFINITION 10 (subterm)

For every context $\mathcal{C}[]$ and terms t, t' with $t \equiv t'$, we have $\mathcal{C}[t] \equiv \mathcal{C}[t']$.

LEMMA 3

Proof. Lemma 3 is a direct consequence of the definition of the denotational semantics. □

The next lemma considers another useful property of substitution that carries over to all calculi considered in the following.

Let $\Gamma, x{::}\tau' \vdash t{::}\tau$ and $\Gamma' \vdash t'{::}\tau'$ with $\Gamma' = \Gamma, x_1{::}\tau_1, \ldots, x_n{::}\tau_n$. Furthermore, let θ, σ and σ' be given with

- $\Gamma_T \subseteq dom(\theta)$,
- $\forall (x :: \tau) \in \Gamma_V.\ \sigma(x) \in [\![\tau]\!]_\theta$, and
- $\sigma' = \sigma[x_1 \mapsto \mathbf{a_1}, \ldots, x_n \mapsto \mathbf{a_n}]$ with $\mathbf{a_1} \in [\![\tau_1]\!]_\theta, \ldots, \mathbf{a_n} \in [\![\tau_n]\!]_\theta$,

then

$$[\![t]\!]_{\sigma[x \mapsto [\![t']\!]_{\sigma'}]} = [\![t[t'/x]]\!]_{\sigma'}$$

LEMMA 4

Proof. Induction over the structure of t. □

2.1.2 Relational Parametricity and Free Theorems

"We explore the thesis that type structure is a syntactic discipline for maintaining levels of abstraction"
Reynolds (1983)

We aim for theorems about functions only gained from the functions' type. The theory that enables such theorems is called *relational parametricity* and was developed by Reynolds (1974, 1983) out of an algebraic view on types. Different implementations of primitive data types should be exchangeable without changing the overall meaning of a program as long as we can establish homomorphisms between these implementations of the data type. For example, it does not matter how exactly integers are implemented. Consider that we have two machines that use different implementations of integers. Now, on both machines, we implement a function f taking an integer as input and yielding an integer as output. Running the function we expect that for machine dependent representations i' and i'' of the same integer i, the different implementations

$$\begin{aligned}
\Delta_{\alpha,\rho} &= \rho(\alpha) \\
\Delta_{Nat,\rho} &= id_{\mathbb{N}} \\
\Delta_{[\tau],\rho} &= \{([\mathbf{a_1},\dots,\mathbf{a_n}],[\mathbf{b_1},\dots,\mathbf{b_n}]) \mid n \in \mathbb{N} \land \forall i \in \{1,\dots,n\}.\ (\mathbf{a_i},\mathbf{b_i}) \in \Delta_{\tau,\rho}\} \\
\Delta_{\tau_1\to\tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \forall(\mathbf{a},\mathbf{b}) \in \Delta_{\tau_1,\rho}.\ (\mathbf{f}\ \mathbf{a},\mathbf{g}\ \mathbf{b}) \in \Delta_{\tau_2,\rho}\}
\end{aligned}$$

Figure 2.5: Logical relation for λ^α

of our function will yield representations o' and o'' that are representations of a unique integer o. If they do so, the two implementations of f, say f' and f'', are also different representations of the same function.

Driven by the view of data types as an abstraction over concrete implementations or representations, Reynolds (1974) formulates a concrete notion of "representation" that he extends in a later paper (Reynolds, 1983) to relations. In particular, he establishes the abstraction or parametricity theorem that generalizes the representation theorem gained in 1974. We explain the abstraction theorem driven by an intuition about representation. Consider the calculus λ^α and its type semantics. We can view the interpretation of a type variable α by θ as choosing a representation for the abstract data type α. Now, exploring α as an abstract type, α has no operation nor constant at all, it has an empty interface. Thus, a representation of type α can be completely arbitrary and we can also arbitrarily choose a mapping between two interpretations of α. We formalize that insight by a relational type interpretation. For type variables we define a *relational environment* ρ that maps type variables to relations between two different interpretations of the type variable. We can intuitively interpret "related" as "representing the same thing". With the same intuition, we can extend the relational type interpretation to arbitrary types in λ^α. The relational interpretation of Nat becomes the identity relation on $\mathbb{N}$. For functions, we use extensionality as criterion for relatedness, i.e., two functions are related if for related inputs they yield related outputs — that is what we expected for f' and f'' above. Lists are related if they have the same length and all corresponding elements are related. The formal definition of the relational interpretation, also called *logical relation*[7], is given in Figure 2.5 and, if our intuition is right, it should satisfy the following property: Consider a term t with a possibly polymorphic type. If we take two different interpretations of type variables, fixed by the type environments θ_1 and θ_2, and if we choose relations between these different interpretations, characterized by a suitable environment ρ, then for every choice of related (w.r.t. ρ) interpretations for term variables that occur unbound in t, the interpretations of t should be related as well. This property is called parametricity or abstraction theorem. We denote the set of relations between two sets S_1 and S_2 by $Rel(S_1, S_2)$ and the collection of all relations between two arbitrary sets by Rel.

relational environment (ρ)

Rel / $Rel(S_1, S_2)$

[7] A family of relations inductively defined over the type structure is called logical relation if the function lifting is defined as for $\Delta_{\cdot,\cdot}$ (Plotkin, 1973; Plotkin et al., 2000; Honsell and Sannella, 2002). We will keep the notion a bit sloppy, as mostly done in literature, and also call the family of relations logical if the function lifting is defined slightly differently.

Theorem 1 (Parametricity Theorem for λ^α, Reynolds (1983))

If $\Gamma \vdash t :: \tau$ valid in λ^α, then for every θ_1, θ_2, ρ, σ_1, σ_2 such that

- for every α occurring in Γ, $\rho(\alpha) \in Rel(\theta_1(\alpha), \theta_2(\alpha))$, and
- for every $x :: \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$,

we have $([\![t]\!]_{\sigma_1}, [\![t]\!]_{\sigma_2}) \in \Delta_{\tau,\rho}$.

A main task throughout this thesis is to restate the parametricity theorem appropriately adjusted in different language settings. Therefore, we present the proof for Theorem 1 completely, although much of it is straightforward. When we extend the λ-calculus, we only fix the proof. That is, we use the following proof as a kind of basis for all upcoming proofs of parametricity theorems.

Proof (Theorem 1). The proof is accomplished by induction on the structure of the type derivation of a term. That is, we regard only the rule at the root of a type derivation, assume all premises satisfy the parametricity theorem (by the induction hypothesis), and prove that the conclusion does as well. The induction is well-founded because every typeable term has a finite type derivation. Hence, we do a case distinction over all typing rules. For all cases the names of variables, terms and types refer to the names in the typing rules shown in Figure 2.2.

(Var)

$$
\begin{aligned}
& ([\![x]\!]_{\sigma_1}, [\![x]\!]_{\sigma_2}) \in \Delta_{\tau,\rho} \\
\Leftrightarrow \quad & \{ \text{ term semantics } \} \\
& (\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau,\rho}
\end{aligned}
$$

The last statement holds by the conditions of Theorem 1.

(Nil)

$$
\begin{aligned}
& ([\![[\,]_\tau]\!]_{\sigma_1}, [\![[\,]_\tau]\!]_{\sigma_2}) \in \Delta_{[\tau],\rho} \\
\Leftrightarrow \quad & \{ \text{ term semantics } \} \\
& ([\,], [\,]) \in \Delta_{[\tau],\rho}
\end{aligned}
$$

The last statement holds by the definition of the logical relation for list types.

(Nat)

$$
\begin{aligned}
& ([\![n]\!]_{\sigma_1}, [\![n]\!]_{\sigma_2}) \in \Delta_{Nat,\rho} \\
\Leftrightarrow \quad & \{ \text{ term semantics } \} \\
& (\mathbf{n}, \mathbf{n}) \in \Delta_{Nat,\rho}
\end{aligned}
$$

The last statement holds by the definition of the logical relation for type Nat.

(CONS)

$$\begin{aligned}
&(\llbracket t_1 : t_2 \rrbracket_{\sigma_1}, \llbracket t_1 : t_2 \rrbracket_{\sigma_2}) \in \Delta_{[\tau],\rho}\\
\Leftrightarrow\quad &\{\text{ term semantics }\}\\
&(\llbracket t_1 \rrbracket_{\sigma_1} : \llbracket t_2 \rrbracket_{\sigma_1}, \llbracket t_1 \rrbracket_{\sigma_2} : \llbracket t_2 \rrbracket_{\sigma_2}) \in \Delta_{[\tau],\rho}\\
\Leftrightarrow\quad &\{\text{ logical relation for list types }\}\\
&(\llbracket t_1 \rrbracket_{\sigma_1}, \llbracket t_1 \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho} \wedge (\llbracket t_2 \rrbracket_{\sigma_1}, \llbracket t_2 \rrbracket_{\sigma_2}) \in \Delta_{[\tau],\rho}
\end{aligned}$$

The last statement holds by the induction hypotheses.

(SUM)

$$\begin{aligned}
&(\llbracket t_1 + t_2 \rrbracket_{\sigma_1}, \llbracket t_1 + t_2 \rrbracket_{\sigma_2}) \in \Delta_{Nat,\rho}\\
\Leftrightarrow\quad &\{\text{ term semantics }\}\\
&(\llbracket t_1 \rrbracket_{\sigma_1} + \llbracket t_2 \rrbracket_{\sigma_1}, \llbracket t_1 \rrbracket_{\sigma_2} + \llbracket t_2 \rrbracket_{\sigma_2}) \in \Delta_{Nat,\rho}\\
\Leftarrow\quad &\{\text{ logical relation for } Nat \}\\
&(\llbracket t_1 \rrbracket_{\sigma_1}, \llbracket t_1 \rrbracket_{\sigma_2}) \in \Delta_{Nat,\rho} \wedge (\llbracket t_2 \rrbracket_{\sigma_1}, \llbracket t_2 \rrbracket_{\sigma_2}) \in \Delta_{Nat,\rho}
\end{aligned}$$

The last statement holds by the induction hypotheses.

(LCASE)

By the first premise in (LCASE) we have $(\llbracket t \rrbracket_{\sigma_1}, \llbracket t \rrbracket_{\sigma_2}) \in \Delta_{[\tau_1],\rho}$, i.e., by the lifting of the logical relation for lists, either

(a) $\llbracket t \rrbracket_{\sigma_1} = \llbracket t \rrbracket_{\sigma_2} = [\,]$, or

(b) $\llbracket t \rrbracket_{\sigma_1} = \mathbf{a} : \mathbf{b}$ and $\llbracket t \rrbracket_{\sigma_2} = \mathbf{c} : \mathbf{d}$ with
(i) $(\mathbf{a}, \mathbf{c}) \in \Delta_{\tau_1,\rho}$ and *(ii)* $(\mathbf{b}, \mathbf{d}) \in \Delta_{[\tau_1],\rho}$.

In case *(a)* we reason as follows.

$$\begin{aligned}
&(\llbracket \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \rightarrow t_1; x_1 : x_2 \rightarrow t_2\} \rrbracket_{\sigma_1},\\
&\ \llbracket \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \rightarrow t_1; x_1 : x_2 \rightarrow t_2\} \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho}\\
\Leftrightarrow\quad &\{\text{ term semantics under conditions of case } (a) \}\\
&(\llbracket t_1 \rrbracket_{\sigma_1}, \llbracket t_1 \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho}
\end{aligned}$$

The last statement holds by the induction hypothesis for the second premise, i.e., by the induction hypothesis for $\Gamma \vdash t_1 :: \tau$.
In case *(b)* we reason as follows.

$$\begin{aligned}
&(\llbracket \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \rightarrow t_1; x_1 : x_2 \rightarrow t_2\} \rrbracket_{\sigma_1},\\
&\ \llbracket \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \rightarrow t_1; x_1 : x_2 \rightarrow t_2\} \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho}\\
\Leftrightarrow\quad &\{\text{ term semantics under conditions of case } (b) \}\\
&(\llbracket t_2 \rrbracket_{\sigma_1[x_1 \mapsto \mathbf{a}, x_2 \mapsto \mathbf{b}]}, \llbracket t_2 \rrbracket_{\sigma_2[x_1 \mapsto \mathbf{c}, x_2 \mapsto \mathbf{d}]}) \in \Delta_{\tau,\rho}
\end{aligned}$$

The last statement holds by the induction hypothesis for the third premise. The extended Γ, σ_1 and σ_2 meet the conditions of Theorem 1 by *(i)* and *(ii)*.

(NCase)

Similar to (LCase).

(Abs)

$$\begin{aligned}
&([\![\lambda x :: \tau_1.t]\!]_{\sigma_1}, [\![\lambda x :: \tau_1.t]\!]_{\sigma_2}) \in \Delta_{\tau_1 \to \tau_2,\rho} \\
\Leftrightarrow \quad & \{ \text{ term semantics } \} \\
&(\lambda \mathbf{a}.[\![t]\!]_{\sigma_1[x \mapsto \mathbf{a}]}, \lambda \mathbf{b}.[\![t]\!]_{\sigma_2[x \mapsto \mathbf{b}]}) \in \Delta_{\tau_1 \to \tau_2,\rho} \\
\Leftrightarrow \quad & \{ \text{ logical relation for function types } \} \\
&\forall (\mathbf{a}, \mathbf{b}) \in \Delta_{\tau_1,\rho}.\ ([\![t]\!]_{\sigma_1[x \mapsto \mathbf{a}]}, [\![t]\!]_{\sigma_2[x \mapsto \mathbf{b}]}) \in \Delta_{\tau_2,\rho}
\end{aligned}$$

The last statement holds by the induction hypothesis.

(App)

$$\begin{aligned}
&([\![t_1\ t_2]\!]_{\sigma_1}, [\![t_1\ t_2]\!]_{\sigma_2}) \in \Delta_{\tau_2,\rho} \\
\Leftrightarrow \quad & \{ \text{ term semantics } \} \\
&([\![t_1]\!]_{\sigma_1}\ [\![t_2]\!]_{\sigma_1}, [\![t_1]\!]_{\sigma_2}\ [\![t_2]\!]_{\sigma_2}) \in \Delta_{\tau_2,\rho} \\
\Leftarrow \quad & \{ \text{ choose } \mathbf{a} = [\![t_2]\!]_{\sigma_1} \text{ and } \mathbf{b} = [\![t_2]\!]_{\sigma_2} \} \\
&([\![t_2]\!]_{\sigma_1}, [\![t_2]\!]_{\sigma_2}) \in \Delta_{\tau_1,\rho} \wedge \\
&\forall (\mathbf{a}, \mathbf{b}) \in \Delta_{\tau_1,\rho}.\ ([\![t_1]\!]_{\sigma_1}\ \mathbf{a}, [\![t_1]\!]_{\sigma_2}\ \mathbf{b}) \in \Delta_{\tau_2,\rho} \\
\Leftrightarrow \quad & \{ \text{ logical relation for function types } \} \\
&([\![t_2]\!]_{\sigma_1}, [\![t_2]\!]_{\sigma_2}) \in \Delta_{\tau_1,\rho} \wedge ([\![t_1]\!]_{\sigma_1}, [\![t_1]\!]_{\sigma_2}) \in \Delta_{\tau_1 \to \tau_2,\rho}
\end{aligned}$$

The last statement holds by the induction hypotheses.

□

The way from the parametricity theorem to free theorems has been explored by Wadler (1989). As he states, his main contribution is *"to suggest that parametricity also has 'specific' applications: it says interesting things about particular functions with particular types."* Before, parametricity had only been used to get "general" assertions about implementations or models of lambda calculi (Wadler, 1989). "Specific" statements are derived by unfolding the logical relation and maybe specializing it. We exemplify the procedure by deriving statement (1.1) that was presented in the introduction (page 5).

Example 4 (derivation of a functional free theorem)

Let $\alpha \vdash f :: [\alpha] \to [\alpha]$, then Theorem 1 states

$$\forall S_1, S_2 \text{ sets}, \mathcal{R} \in Rel(S_1, S_2).\ ([\![f]\!]_\emptyset, [\![f]\!]_\emptyset) \in \Delta_{[\alpha] \to [\alpha], [\alpha \mapsto \mathcal{R}]}$$

By the definition of the logical relation for function types we obtain

$$\begin{aligned}
&\forall S_1, S_2 \text{ sets}, \mathcal{R} \in Rel(S_1, S_2), (\mathbf{x}, \mathbf{y}) \in \Delta_{[\alpha], [\alpha \mapsto \mathcal{R}]}. \\
&\qquad ([\![f]\!]_\emptyset\ \mathbf{x}, [\![f]\!]_\emptyset\ \mathbf{y}) \in \Delta_{[\alpha], [\alpha \mapsto \mathcal{R}]}
\end{aligned}$$

and specializing S_1 to $[\![\tau_1]\!]_\emptyset$, S_2 to $[\![\tau_2]\!]_\emptyset$, and $\mathcal{R}$ to (the graph of) $[\![g]\!]_\emptyset$ with $g :: \tau_1 \to \tau_2$, we get

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, (\mathbf{x}, \mathbf{y}) \in \Delta_{[\alpha],[\alpha \mapsto [\![g]\!]_\emptyset]}.\\ ([\![f]\!]_\emptyset\ \mathbf{x}, [\![f]\!]_\emptyset\ \mathbf{y}) \in \Delta_{[\alpha],[\alpha \mapsto [\![g]\!]_\emptyset]}$$

Unfolding the list lifting (plus term semantics definition) we obtain

$$\begin{aligned}\Delta_{[\alpha],[\alpha \mapsto [\![g]\!]_\emptyset]} &= \{([\mathbf{x_1}, \ldots, \mathbf{x_n}], [[\![g]\!]_\emptyset\ \mathbf{x_1}, \ldots, [\![g]\!]_\emptyset\ \mathbf{x_n}]) \mid \\ &\qquad n \in \mathbb{N} \wedge \forall i \in \{1, \ldots, n\}.\ \mathbf{x_i} \in [\![\tau_1]\!]_\emptyset\} \\ &= \{(\mathbf{xs}, [\![map\ g]\!]_\emptyset\ \mathbf{xs}) \mid \mathbf{xs} \in [\![[\tau_1]]\!]\}\end{aligned}$$

If we apply that equivalence twice (plus term semantics definition) we get

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, xs :: [\tau_1].\ [\![map\ g\ (f\ xs)]\!]_\emptyset = [\![f\ (map\ g\ xs)]\!]_\emptyset$$

which is statement (1.1) from the introduction where it was left implicit that we mean semantic equivalence.

We can derive similar theorems for every possible (polymorphic) type. Wadler (1989) was optimistic about the applications of such theorems. And as the many applications briefly mentioned in Section 3.2 attest, he was right. However, mostly these theorems are not used in a language that resembles the polymorphic lambda calculus, nor our even simpler language λ^α. Concerning programming languages, the feature necessary to achieve Turing-completeness is general recursion. Unfortunately, neither λ^α nor the polymorphic lambda calculus provide general recursion. Adding it to λ^α (and also to the polymorphic lambda calculus) has a serious impact on the semantics and weakens free theorems. Nevertheless, we have to deal with general recursion to handle real world programming languages, and, thus, the next section describes how we can add it to λ^α and what happens then to parametricity.

2.2 Adding General Recursion

In Subsection 2.2.1, we explore how to integrate general recursion into λ^α, setting up the calculus $\lambda^\alpha_{\mathbf{fix}}$. Subsection 2.2.2 describes the necessary adjustments in the relational type interpretation to reestablish a parametricity theorem.

2.2.1 Changes to the Calculus

General recursion is not expressible in λ^α and needs to be explicitly added via a primitive. We extend the term syntax by **fix** t and the typing rules given in Figure 2.2 by

$$\frac{\Gamma \vdash t :: \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ t :: \tau}\ (\textsc{Fix})$$

$\lambda^\alpha_{\mathbf{fix}}$

We call the resulting calculus $\lambda^\alpha_{\mathbf{fix}}$. For later reference, the extensions to syntax and typing rules are given in Figures 2.6 and 2.7. As the semantics of **fix** t

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ t & ::= & \ldots & \\ & | & \mathbf{fix}\ t & \text{fixpoint primitive} \end{array}$$

Figure 2.6: Type and term syntax of $\lambda^{\alpha}_{\mathbf{fix}}$, extended from Figure 2.1

$$\frac{\Gamma \vdash t :: \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ t :: \tau}\ (\textsc{Fix})$$

Figure 2.7: Typing rules of $\lambda^{\alpha}_{\mathbf{fix}}$, extended from Figure 2.2

we choose the fixpoint of the function $\mathbf{t} = [\![t]\!]_\sigma \in [\![\tau \to \tau]\!]_\theta$.[8] The idea behind the calculation of the fixpoint is as follows. A recursively defined function $\mathbf{f} \in [\![\tau]\!]_\theta$ is a function that calls itself. We can abstract over the recursive call and express $\mathbf{f}$ as an application of $\mathbf{t}$ to $\mathbf{f}$. That means $\mathbf{f}$ is a fixpoint of $\mathbf{t}$, i.e., $\mathbf{t}\ \mathbf{f} = \mathbf{f}$. However, we still do not know $\mathbf{f}$ in a non self-referring way and hence on the first sight the equivalence $\mathbf{t}\ \mathbf{f} = \mathbf{f}$ seems useless. But, it really tells us something interesting, namely that $\mathbf{f}$ is a fixpoint of $\mathbf{t}$. That is, when we start with a function different from $\mathbf{f}$ as input to $\mathbf{t}$ and apply $\mathbf{t}$ often enough to that input, we may get $\mathbf{f}$. Thus, we may define the semantics of $\mathbf{fix}\ t$ as follows: Choose an appropriate initial input to $\mathbf{t}$ and apply $\mathbf{t}$ over and over again till a fixpoint is reached. But, immediately two (though related) questions arise.

1. What to take as the initial input to $\mathbf{t}$?
2. What fixpoint shall we select if there exists more than one?

To formally answer these questions major changes to the denotational semantics are necessary. The required formal background is the subject of domain theory. Abramsky and Jung (1994) provide an overview of the topic. Here, we discuss only briefly the important results.

The fundamental semantic change is to treat a type no longer as a set, but as a pointed complete partial order (pcpo). Knowledge about a value, or definedness, is modeled semantically by the order in the respective pcpo, called *definedness order* $\sqsubseteq$.

definedness order

Definition 11 **((pointed) complete partial order, (p)cpo)**

Let S a set and $\sqsubseteq$ a partial order on S. We say the partially ordered set, short poset, $D = (S, \sqsubseteq)$ is *complete* (a cpo) if for every chain[9] $C \subseteq D$ there exists a supremum s in D, denoted as $s = \bigsqcup_{c \in C} c$. D is *pointed* if there exists a least element. We abbreviate pointed complete partial order as pcpo.

[8]In most cases, though other uses are possible, τ will be a function type $\tau_1 \to \tau_2$, i.e., we define a recursive function $\mathbf{f} \in [\![\tau_1 \to \tau_2]\!]_\theta$ via the fixpoint $\mathbf{f} = [\![\mathbf{fix}\ t]\!]_\theta$.

[9]A chain is a totally ordered subset of a partially ordered set. We write $C \subseteq D$ if $D = (S, \sqsubseteq)$ and $C = (S', \sqsubseteq|_{S'})$ with $S' \subseteq S$.

Convention 5 ($\bot$)

For every pcpo we denote the least element by $\bot$.

When interpreting types as pcpos, we can provide an appropriate definition for the semantics of $\mathbf{fix}\ t$.[10]

$$[\![\mathbf{fix}\ t]\!]_\sigma = \bigsqcup_{n \geqslant 0} (([\![t]\!]_\sigma)^n \perp) \tag{2.1}$$

Say, we want to define a function f recursively. Then this is achieved by an appropriate t whose least fixpoint is f. Starting with the completely undefined function $\bot$ as argument to t, with every application of t the function becomes more (at least not less) defined and finally the supremum yields the least fixpoint of t and hence our recursive function. That the supremum $\bigsqcup_{n \geqslant 0} ([\![t]\!]_\sigma)^n \perp$ always exists and characterizes the least fixpoint of t is guaranteed by our choice of pcpos as structure for type interpretation (Abramsky and Jung, 1994).

A new semantic equivalence arising from the interpretation of $\mathbf{fix}\ t$ as a fixpoint of t is the following

Lemma 5 (unrolling)

For every $t :: \tau \to \tau$,

$$\mathbf{fix}\ t \equiv t\ (\mathbf{fix}\ t) \tag{unrolling}$$

Example 5 (definition of map)

Let us exemplify the usage of **fix** by defining a recursive function (via the semantics of) $map :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ that applies a function (its first argument) to each element in a list (given as second argument):

$$\begin{aligned} map = \lambda f :: \alpha \to \beta.& \\ &\mathbf{fix}\ (\lambda m :: [\alpha] \to [\beta].\lambda xs :: [\alpha]. \\ &\quad \mathbf{case}\ xs\ \mathbf{of}\ \{[\,] \to [\,]_\beta; y : ys \to f\ y : m\ ys\}) \end{aligned}$$

Being precise, the recursive function is not directly (the semantic interpretation of) map, but (the interpretation of) $map\ f$ for some given f.[11] What happens on the semantic level when the fixpoint is evaluated can be simulated via equational reasoning with the rules from Lemmas 2, 3 and 5. During the reasoning we abbreviate $\lambda m :: [\alpha] \to [\beta].\lambda xs :: [\alpha].\mathbf{case}\ xs\ \mathbf{of}\ \{[\,] \to [\,]_\beta; y : ys \to f\ y : m\ ys\}$ by t.

$$\begin{aligned} & map\ f \\ \equiv\ & \{\text{ definition of } map \text{ and } \beta\text{-conversion }\} \\ & \mathbf{fix}\ t \end{aligned}$$

[10]The exponent n means that $[\![t]\!]_\sigma$ is applied n times, e.g. $(([\![t]\!]_\sigma)^3 \perp) = [\![t]\!]_\sigma\ ([\![t]\!]_\sigma\ ([\![t]\!]_\sigma \perp))$.

$\equiv$ { unrolling and definition of t }

$(\lambda m :: [\alpha] \to [\beta].\lambda xs :: [\alpha].$
$\quad \textbf{case } xs \textbf{ of } \{[\,] \to [\,]_\beta; y : ys \to f\ y : m\ ys\})\ (\textbf{fix } t)$

$\equiv$ { β-conversion }

$\lambda xs :: [\alpha].\textbf{case } xs \textbf{ of } \{[\,] \to [\,]_\beta; y : ys \to f\ y : (\textbf{fix } t)\ ys\}$

$\equiv$ { unroll **fix** t again, α- and β-conversion }

$\lambda xs :: [\alpha].\textbf{case } xs \textbf{ of } \{$
$\quad [\,] \to [\,]_\beta;$
$\quad y : ys \to f\ y : \textbf{case } ys \textbf{ of } \{[\,] \to [\,]_\beta; z : zs \to f\ z : (\textbf{fix } t)\ zs\}\}$

$\equiv$ { repeat last step }

...

We unroll the fixpoint more and more and that way obtain a more and more explicit description of $map\ f$. In the beginning we only know that it is defined via the fixpoint, i.e., via itself. After one unrolling step, we know that for the empty list as input it yields the empty list as output — independently of **fix** t. One more unrolling step and we know how it is defined for singleton lists independently of **fix** t. Hence, step by step we gain a description of a function that maps f into arbitrary long lists and that is not defined in terms of itself anymore. That function is the fixpoint **fix** t.

EXAMPLE 6 (application of map)

If we apply map[12] to concrete terms, say $\lambda x :: Nat.x + 1$ and a list $1 : 2 : [\,]_{Nat}$ evaluation (stated as a sequence of equational reasoning steps and thus more in an operational way, but in line with explicitly employing our denotational semantics) proceeds as follows, where we abbreviate $\lambda m :: [Nat] \to [Nat].\lambda xs :: [Nat].\textbf{case } xs \textbf{ of } \{[\,] \to [\,]_{Nat}; y : ys \to (\lambda x.x + 1)\ y : m\ ys\})$ by t.

$map\ (\lambda x.x + 1)\ (1 : 2 : [\,]_{Nat})$

$\equiv$ { definition of map, β-conversion and definition of t }

$\textbf{fix } t\ (1 : 2 : [\,]_{Nat})$

$\equiv$ { unrolling }

$(\lambda m :: [Nat] \to [Nat].\lambda xs :: [Nat].$
$\quad \textbf{case } xs \textbf{ of } \{[\,] \to [\,]_{Nat}; y : ys \to (\lambda x.x + 1)\ y : m\ ys\})$
$\quad\quad (\textbf{fix } t)\ (1 : 2 : [\,]_{Nat})$

$\equiv$ { β-conversion (twice) }

$\textbf{case } 1 : 2 : [\,]_{Nat} \textbf{ of } \{[\,] \to [\,]_{Nat}; y : ys \to (\lambda x.x + 1)\ y : (\textbf{fix } t)\ ys\}$

$\equiv$ { evaluate **case** expression }

$(\lambda x.x + 1)\ 1 : (\textbf{fix } t)\ (2 : [\,]_{Nat})$

$\equiv$ { β-conversion }

$2 : (\textbf{fix } t)\ (2 : [\,]_{Nat})$

[11]It would also be possible to directly write map as $map' = \textbf{fix } t$ for an appropriate t.

$$
\begin{aligned}
&\equiv \quad \{ \text{ unrolling } \}\\
&2 : (\lambda m :: [Nat] \to [Nat].\lambda xs :: [Nat].\\
&\qquad \mathbf{case}\ xs\ \mathbf{of}\ \{[\,] \to [\,]_{Nat}; y : ys \to (\lambda x.(x+1))\ y : m\ ys\})\\
&\qquad\quad (\mathbf{fix}\ t)\ (2 : [\,]_{Nat})\\
&\equiv \quad \{ \ \beta\text{-conversion (twice) } \}\\
&2 : \mathbf{case}\ 2 : [\,]_{Nat}\ \mathbf{of}\ \{[\,] \to [\,]_{Nat}; y : ys \to (\lambda x.x+1)\ y : (\mathbf{fix}\ t)\ ys\}\\
&\equiv \quad \{ \text{ evaluate } \mathbf{case} \text{ expression } \}\\
&2 : (\lambda x.x+1)\ 2 : (\mathbf{fix}\ t)\ [\,]_{Nat}\\
&\equiv \quad \{ \ \beta\text{-conversion } \}\\
&2 : 3 : (\mathbf{fix}\ t)\ [\,]_{Nat}\\
&\equiv \quad \{ \text{ unrolling } \}\\
&2 : 3 : (\lambda m :: [Nat] \to [Nat].\lambda xs :: [Nat].\\
&\qquad \mathbf{case}\ xs\ \mathbf{of}\ \{[\,] \to [\,]_{Nat}; y : ys \to (\lambda x.x+1)\ y : m\ ys\})\\
&\qquad\quad (\mathbf{fix}\ t)\ [\,]_{Nat}\\
&\equiv \quad \{ \ \beta\text{-conversion (twice) } \}\\
&2 : 3 : \mathbf{case}\ [\,]_{Nat}\ \mathbf{of}\ \{[\,] \to [\,]_{Nat}; y : ys \to (\lambda x.x+1)\ y : (\mathbf{fix}\ t)\ ys\}\\
&\equiv \quad \{ \text{ evaluate } \mathbf{case} \text{ expression } \}\\
&2 : 3 : [\,]_{Nat}
\end{aligned}
$$

Note in particular, that there is no need to evaluate the fixpoint completely. Evaluation stops as soon as the current approximation can handle the concrete list that is given as argument.

$(\cdot)_\perp$

lfp

As already said, to allow for the just given fixpoint interpretation, we have to adjust the type semantics, in particular to switch from sets to pcpos. The adjusted type semantics, with θ now mapping from type variables to pcpos, is presented in Figure 2.8. There, $\mathbb{N}$ is regarded as discretely ordered and $(\cdot)_\perp$ takes a partially ordered set, adds a least element $\perp$, and otherwise leaves the ordering unchanged. The least fixpoint of a function $\mathbf{f}$ is denoted by $\mathit{lfp}(\mathbf{f})$. Note that, in contrast to the list semantics in λ^α, the definition via the least fixpoint provides for infinite lists. These can arise as semantics of a recursive definition of a list. It is worth noting that indeed the least fixpoint suffices to capture infinite lists. The set without infinite lists, which would be the least fixpoint w.r.t. sets, is not a pcpo. The function space is restricted to monotone and continuous functions[13], ordered point-wise. The restrictions guarantee the function space to constitute a pcpo. Lifting is not necessary since there already is a least function, i.e., the one that takes every input to the undefined value.

[12]We changed the type of map to $(Nat \to Nat) \to [Nat] \to [Nat]$ because otherwise we encounter type problems. How these problems are solved without changing map's type in the first place is explained in Section 2.4.

[13]The restriction is equivalent to Scott-continuity (cf. Abramsky and Jung (1994, Definition 2.1.17)). Furthermore, we silently employed continuity of functions already for $[\![t]\!]_\sigma$ when explaining the definition in (2.1) on page 28.

$$\begin{array}{llll} [\![\alpha]\!]^{\mathbf{fix}}_\theta & = \theta(\alpha) & [\![[\tau]]\!]^{\mathbf{fix}}_\theta & = \mathit{lfp}(\lambda S.\ (\{[\,]\} \cup \{\mathbf{a} : \mathbf{b} \mid \mathbf{a} \in [\![\tau]\!]^{\mathbf{fix}}_\theta, \mathbf{b} \in S\})_\perp) \\ [\![Nat]\!]^{\mathbf{fix}}_\theta & = \mathbb{N}_\perp & [\![\tau_1 \to \tau_2]\!]^{\mathbf{fix}}_\theta & = \{\mathbf{f} : [\![\tau_1]\!]^{\mathbf{fix}}_\theta \to [\![\tau_2]\!]^{\mathbf{fix}}_\theta\} \end{array}$$

Figure 2.8: Type semantics of $\lambda^\alpha_{\mathbf{fix}}$

$$\begin{aligned} [\![x]\!]^{\mathbf{fix}}_\sigma &= \sigma(x) \\ [\![n]\!]^{\mathbf{fix}}_\sigma &= \mathbf{n} \\ [\![t_1 + t_2]\!]^{\mathbf{fix}}_\sigma &= \\ &\begin{cases} [\![t_1]\!]^{\mathbf{fix}}_\sigma + [\![t_2]\!]^{\mathbf{fix}}_\sigma & \text{if } [\![t_1]\!]^{\mathbf{fix}}_\sigma \neq \perp \\ & \wedge [\![t_2]\!]^{\mathbf{fix}}_\sigma \neq \perp \\ \perp & \text{otherwise} \end{cases} \\ [\![[\,]_\tau]\!]^{\mathbf{fix}}_\sigma &= [\,] \\ [\![t_1 : t_2]\!]^{\mathbf{fix}}_\sigma &= [\![t_1]\!]^{\mathbf{fix}}_\sigma : [\![t_2]\!]^{\mathbf{fix}}_\sigma \\ [\![\lambda x :: \tau.t]\!]^{\mathbf{fix}}_\sigma &= \lambda\mathbf{a}.[\![t]\!]^{\mathbf{fix}}_{\sigma[x \mapsto \mathbf{a}]} \\ [\![t_1\ t_2]\!]^{\mathbf{fix}}_\sigma &= [\![t_1]\!]^{\mathbf{fix}}_\sigma\ [\![t_2]\!]^{\mathbf{fix}}_\sigma \end{aligned}$$

$$\begin{aligned} &[\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{\mathbf{fix}}_\sigma = \\ &\qquad\begin{cases} [\![t_1]\!]^{\mathbf{fix}}_\sigma & \text{if } [\![t]\!]^{\mathbf{fix}}_\sigma = \mathbf{0} \\ [\![t_2]\!]^{\mathbf{fix}}_\sigma & \text{if } [\![t]\!]^{\mathbf{fix}}_\sigma \in \mathbb{N} \setminus \{\mathbf{0}\} \\ \perp & \text{if } [\![t]\!]^{\mathbf{fix}}_\sigma = \perp \end{cases} \\ &[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1; x_1 : x_2 \to t_2\}]\!]^{\mathbf{fix}}_\sigma = \\ &\qquad\begin{cases} [\![t_1]\!]^{\mathbf{fix}}_\sigma & \text{if } [\![t]\!]^{\mathbf{fix}}_\sigma = [\,] \\ [\![t_2]\!]^{\mathbf{fix}}_{\sigma[x_1 \mapsto \mathbf{a}, x_2 \mapsto \mathbf{b}]} & \text{if } [\![t]\!]^{\mathbf{fix}}_\sigma = \mathbf{a} : \mathbf{b} \\ \perp & \text{if } [\![t]\!]^{\mathbf{fix}}_\sigma = \perp \end{cases} \\ &[\![\mathbf{fix}\ t]\!]^{\mathbf{fix}}_\sigma = \bigsqcup_{n \geqslant 0} (([\![t]\!]^{\mathbf{fix}}_\sigma)^n \perp) \end{aligned}$$

Figure 2.9: Term semantics of $\lambda^\alpha_{\mathbf{fix}}$

Concerning the term semantics, besides adding the semantic interpretation for **fix** t, the semantics of case expressions and of addition has to be altered: The scrutinee of any case expression can evaluate to $\perp$, as for example the expression **fix** $(\lambda x :: \tau.x)$ will do. In this case the semantics of the whole case expression is $\perp$. Regarding addition, the summands may evaluate to $\perp$. If one of them does so, the whole sum evaluates to $\perp$. Figure 2.9 shows the complete term semantics. To distinguish the semantics of λ^α from the one of $\lambda^\alpha_{\mathbf{fix}}$ we add the superscript **fix** to the semantic function, for types and also for terms.[14]

Finally, we need to ensure that the altered semantics is still well-defined, i.e., that the semantics of each term of type τ is an element of the semantics of τ.

Lemma 6

If $\Gamma \vdash t :: \tau$ valid in $\lambda^\alpha_{\mathbf{fix}}$, then for all θ and σ with

- $\Gamma_\mathcal{T} \subseteq dom(\theta)$ and
- $(x :: \tau') \in \Gamma_\mathcal{V} \Rightarrow \sigma(x) \in [\![\tau']\!]^{\mathbf{fix}}_\theta$

we have $[\![t]\!]^{\mathbf{fix}}_\sigma \in [\![\tau]\!]^{\mathbf{fix}}_\theta$.

Proof. Induction over the type derivation. Special care has to be taken concerning continuity. □

[14] In statement (2.1) we omitted the superscript of the semantic function to avoid confusion.

2.2.2 Changes to the Parametricity Results

Since the expressiveness of $\lambda^{\alpha}_{\mathbf{fix}}$ has increased compared to λ^{α} we expect also changes to the parametricity results gained in Section 2.1.2. In particular that for every type we have a least element in the interpretation, the completely undefined value, gives rise to changes, as was already observed by Wadler (1989). If we reconsider our intuition about parametricity, the main idea was to regard a completely polymorphic type (i.e., a type variable) as an abstract data type with an empty interface. That intuition does not hold anymore. The completely undefined value $\perp$ is present in the semantic interpretation of each type and for each type τ the term $\mathbf{fix}\ (\lambda x :: \tau.x)$, in the following abbreviated as $\perp_\tau$, is interpreted as $\perp$. Hence, there is a constant present for every type and consequently the completely polymorphic type has a non-empty interface. Intuitively, the logical relation now has to respect the interface. It has to relate $\perp$ from one interpretation to $\perp$ from the other, i.e., has to be *strict*. This restriction of the logical relation helps us to reestablish the parametricity theorem. However, another restriction is necessary. If two chains of values are related element-wise, then also the suprema of these chains should be related. That property is called *continuity*. The necessity of continuity arises from the interpretation of fixpoints. The semantics of $\mathbf{fix}\ t$ in two different environments is a pair of suprema of element-wise related chains, and of course we want the two interpretations of $\mathbf{fix}\ t$ to be related.

$\perp_\tau$

The adjusted logical relation is given in Figure 2.10. As just discussed, ρ maps to *strict* and *continuous* relations between pcpos.

Definition 12 (strict relation)

A relation $\mathcal{R}$ between two pcpos is *strict* if $(\perp, \perp) \in \mathcal{R}$.

Definition 13 (continuous relation)

A relation $\mathcal{R}$ between two cpos $(D_1, \sqsubseteq_1)$, $(D_2, \sqsubseteq_2)$ is *continuous* if for all chains $C_1 \subseteq D_1$ and $C_2 \subseteq D_2$ whose elements are pairwise related by $\mathcal{R}$ the suprema of these chains are also related by $\mathcal{R}$.

$Rel^{\perp} / Rel^{\perp}(D_1, D_2)$

Without the continuity and the strictness restriction, the parametricity theorem for $\lambda^{\alpha}_{\mathbf{fix}}$ fails. In the following, we denote by $Rel^{\perp}(D_1, D_2)$ the set of strict and continuous relations between the pcpos D_1 and D_2 and by $Rel^{\perp}$ the collection of all such relations between two arbitrary pcpos. The second main adjustment to the logical relation is the lifting for lists. We have to allow for infinite lists. The adjustment is similar to the one for the standard type interpretation.

With the altered logical relation we restate the parametricity theorem. To prove it, it is essential that strictness and continuity propagate through the whole logical relation.

$$\begin{aligned}
\Delta^{\mathbf{fix}}_{\alpha,\rho} &= \rho(\alpha)\\
\Delta^{\mathbf{fix}}_{Nat,\rho} &= id_{\mathbb{N}_\bot}\\
\Delta^{\mathbf{fix}}_{[\tau],\rho} &= lfp(\lambda R.\ \{(\bot,\bot),([\,],[\,])\} \cup \{(\mathbf{a} : \mathbf{as}, \mathbf{b} : \mathbf{bs}) \mid (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{fix}}_{\tau,\rho} \wedge (\mathbf{as},\mathbf{bs}) \in R\})\\
\Delta^{\mathbf{fix}}_{\tau_1\to\tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \forall (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{fix}}_{\tau_1,\rho}.\ (\mathbf{f}\ \mathbf{a}, \mathbf{g}\ \mathbf{b}) \in \Delta^{\mathbf{fix}}_{\tau_2,\rho}\}
\end{aligned}$$

Figure 2.10: Logical relation for $\lambda^{\alpha}_{\mathbf{fix}}$

LEMMA 7

If ρ maps into $Rel^\bot$, then $\Delta^{\mathbf{fix}}_{\tau,\rho} \in Rel^\bot$ for all τ closed under $dom(\rho)$.

Proof. The proof proceeds by structural induction on the type τ. □

THEOREM 2 (Parametricity Theorem for $\lambda^{\alpha}_{\mathbf{fix}}$, Wadler (1989))

If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}}$, then for every θ_1, θ_2, ρ, σ_1, σ_2 such that

- for every α occurring in Γ, $\rho(\alpha) \in Rel^\bot(\theta_1(\alpha), \theta_2(\alpha))$, and
- for every $x :: \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathbf{fix}}_{\tau',\rho}$,

we have $([\![t]\!]^{\mathsf{fix}}_{\sigma_1}, [\![t]\!]^{\mathsf{fix}}_{\sigma_2}) \in \Delta^{\mathbf{fix}}_{\tau,\rho}$.

Proof. We only point out differences to the proof of Theorem 1. Of course, we have (FIX) as an extra case:

(FIX)

$$\begin{aligned}
&([\![\mathbf{fix}\ t]\!]^{\mathsf{fix}}_{\sigma_1}, [\![\mathbf{fix}\ t]\!]^{\mathsf{fix}}_{\sigma_2}) \in \Delta^{\mathbf{fix}}_{\tau,\rho}\\
\Leftrightarrow\quad &\{\text{ term semantics }\}\\
&(\textstyle\bigsqcup_{n\geqslant 0}\ (([\![t]\!]^{\mathsf{fix}}_{\sigma_1})^n\ \bot), \bigsqcup_{n\geqslant 0}\ (([\![t]\!]^{\mathsf{fix}}_{\sigma_2})^n\ \bot)) \in \Delta^{\mathbf{fix}}_{\tau,\rho}\\
\Leftarrow\quad &\{\text{ continuity of } \Delta^{\mathsf{fix}}_{\tau,\rho} \text{ (Lemma 7) }\}\\
&\forall n \in \mathbb{N}.\ (([\![t]\!]^{\mathsf{fix}}_{\sigma_1})^n\ \bot, ([\![t]\!]^{\mathsf{fix}}_{\sigma_2})^n\ \bot) \in \Delta^{\mathbf{fix}}_{\tau,\rho}\\
\Leftarrow\quad &\{\text{ strictness of } \Delta^{\mathsf{fix}}_{\tau,\rho} \text{ (Lemma 7) and logical relation for functions }\}\\
&\forall (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{fix}}_{\tau,\rho}.\ ([\![t]\!]^{\mathsf{fix}}_{\sigma_1}\ \mathbf{a}, [\![t]\!]^{\mathsf{fix}}_{\sigma_2}\ \mathbf{b}) \in \Delta^{\mathbf{fix}}_{\tau,\rho}\\
\Leftrightarrow\quad &\{\text{ logical relation for functions }\}\\
&([\![t]\!]^{\mathsf{fix}}_{\sigma_1}, [\![t]\!]^{\mathsf{fix}}_{\sigma_2}) \in \Delta^{\mathbf{fix}}_{\tau\to\tau,\rho}
\end{aligned}$$

Additionally for (LCASE) and (NCASE) changes are required. Both times an extra subcase with $[\![t]\!]^{\mathsf{fix}}_{\sigma_1} = [\![t]\!]^{\mathsf{fix}}_{\sigma_2} = \bot$ arises. It is proved immediately by strictness of the logical relation (Lemma 7). Similarly, an extra subcase arises for (SUM). All other cases remain unchanged, except that $\Delta_{\cdot,\cdot}$ is replaced by $\Delta^{\mathbf{fix}}_{\cdot,\cdot}$. □

Before we close the section, we examine how the changes on the parametricity theorem affect free theorems. Therefore we regard Example 4 again, but now w.r.t. Theorem 2.

Example 7 (derivation of a functional free theorem)

Let $\alpha \vdash f :: [\alpha] \to [\alpha]$, then Theorem 2 states

$$\forall S_1, S_2 \text{ sets}, \mathcal{R} \in \mathit{Rel}^{\perp}(S_1, S_2).\ ([\![f]\!]_\emptyset, [\![f]\!]_\emptyset) \in \Delta^{\mathbf{fix}}_{[\alpha]\to[\alpha],[\alpha\mapsto\mathcal{R}]}$$

By the definition of the logical relation for function types we obtain

$$\begin{aligned}&\forall S_1, S_2 \text{ sets}, \mathcal{R} \in \mathit{Rel}^{\perp}(S_1, S_2), (\mathbf{x}, \mathbf{y}) \in \Delta^{\mathbf{fix}}_{[\alpha],[\alpha\mapsto\mathcal{R}]}.\\ &\qquad ([\![f]\!]_\emptyset\ \mathbf{x}, [\![f]\!]_\emptyset\ \mathbf{y}) \in \Delta^{\mathbf{fix}}_{[\alpha],[\alpha\mapsto\mathcal{R}]}\end{aligned}$$

and specializing S_1 to $[\![\tau_1]\!]_\emptyset$, S_2 to $[\![\tau_2]\!]_\emptyset$, and $\mathcal{R}$ to (the graph of) a strict function $[\![g]\!]_\emptyset$ with $g :: \tau_1 \to \tau_2$, we get

$$\begin{aligned}&\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2 \text{ strict}, (\mathbf{x}, \mathbf{y}) \in \Delta^{\mathbf{fix}}_{[\alpha],[\alpha\mapsto[\![g]\!]_\emptyset]}.\\ &\qquad ([\![f]\!]_\emptyset\ \mathbf{x}, [\![f]\!]_\emptyset\ \mathbf{y}) \in \Delta^{\mathbf{fix}}_{[\alpha],[\alpha\mapsto[\![g]\!]_\emptyset]}\end{aligned}$$

Note that $[\![g]\!]_\emptyset$ really has to be strict. Otherwise its graph is not in $\mathit{Rel}^{\perp}$. Continuity is guaranteed anyway, because each term of function type is semantically interpreted as a continuous function (see Lemma 6 and the semantic interpretation of function types).

Unfolding the list lifting (plus term semantics definition) we obtain

$$\begin{aligned}\Delta^{\mathbf{fix}}_{[\alpha],[\alpha\mapsto[\![g]\!]_\emptyset]} &= \mathit{lfp}(\lambda R.\ \{(\perp, \perp), ([\,], [\,])\} \cup \{(\mathbf{x} : \mathbf{xs}, [\![g]\!]_\emptyset\ \mathbf{x} : \mathbf{ys}) \mid\\ &\qquad (\mathbf{xs}, \mathbf{ys}) \in R\})\\ &= \{(\mathbf{xs}, [\![\mathit{map}\ g]\!]_\emptyset\ \mathbf{xs}) \mid \mathbf{xs} \in [\![[\tau_1]]\!]\}\end{aligned}$$

If we apply that equivalence twice (plus term semantics definition) we get

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2 \text{ strict}, \mathit{xs} :: [\tau_1].\ [\![\mathit{map}\ g\ (f\ \mathit{xs})]\!]_\emptyset = [\![f\ (\mathit{map}\ g\ \mathit{xs})]\!]_\emptyset$$

which, besides the additional strictness restriction, is statement (1.1) from the introduction where it was left implicit that we mean semantic equivalence.

2.3 Adding Selective Strictness

In Subsection 2.3.1 we explain the difference between strict and non-strict evaluation and show how we can enrich the calculus $\lambda^{\alpha}_{\mathbf{fix}}$ with a strict let expression. The enriched calculus is called $\lambda^{\alpha}_{\mathbf{seq}}$. The adjustments to the theory of parametricity that are caused by the possibility of strict evaluation are described in Subsection 2.3.2.

For motivation why selective strictness is a useful feature we refer to Hudak et al. (2007, Section 10.3) and the introduction of Chapter 5 in this thesis.

2.3.1 Changes to the Calculus

non-strict evaluation

For all calculi up to now, the given semantics corresponds to *non-strict evaluation*. The general description for non-strict evaluation is that terms are only evaluated when necessary to go on with the overall computation. This property is not found directly in the denotational semantics, because it is too high level to capture non-strict evaluation the way just described. The description is fitting for a rewrite system, in which terms are rewritten to normal forms that then are considered as their semantics. In the denotational semantics, non-strict evaluation is realized by treating the undefined value $\perp$ the same way as all other values. In particular, our semantics allows for functions that get $\perp$ as argument and do not return $\perp$, but yield more defined values. Furthermore, the semantics allows for partially defined lists. In a strict calculus both would not be allowed. *Strict evaluation* means that arguments to functions and constructors are completely evaluated before the function result or the constructed data structure is returned. If the evaluation fails, as is indicated by $\perp$, the result of the whole application is undefined, i.e., $\perp$.

strict evaluation

Haskell, in general evaluating non-strict, supplies the programmer with the possibility to force strict evaluation. It provides the primitive $\mathbf{seq} :: \alpha \to \beta \to \beta$ that forces its first argument to be evaluated to weak head normal form[15] and, only if successful, returns the second argument, otherwise the result is undefined. In pseudo code, we can specify **seq** as follows:

```
$\mathbf{seq} :: \alpha \to \beta \to \beta$
$\mathbf{seq}\ \perp\ \_ = \perp$
$\mathbf{seq}\ \_\ \ x = x$
```

Haskell also provides other ways to force strictness, e.g. via strict function application $! or strict constructors. However, all these constructs can be simulated by **seq** and **seq** itself can be simulated by a strict let expression as we add it to the calculus $\lambda^{\alpha}_{\mathbf{fix}}$.[16] The syntax of the let expression is shown in Figure 2.11. The typing rule added to $\lambda^{\alpha}_{\mathbf{fix}}$ due to the strict let expression is given in Figure 2.12. The intention behind the expression $\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2$ is to test if the term t_1 assigned to the variable x is $\perp$, and if so interpret the whole let expression as undefined, but otherwise employ the value assigned to x in the evaluation of t_2. We call the extended calculus $\lambda^{\alpha}_{\mathbf{seq}}$. The following example should clarify the intended semantics of a strict let expression.

$\lambda^{\alpha}_{\mathbf{seq}}$

[15]The notion weak head normal form means to rewrite a term until it is a constant of a base type, a constructor (with possibly unevaluated arguments) or a term of the form $\lambda x.t$. In the denotational semantics evaluation to weak head normal form corresponds to a check if a value is completely undefined, i.e., is $\perp$.

[16]We can define $\mathbf{seq}\ t_1\ t_2 = \mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2$ where x is fresh.

$$\begin{array}{lll} \tau & ::= & \ldots \\ t & ::= & \ldots \\ & | & \mathbf{let!}\ x = t\ \mathbf{in}\ t \qquad \text{strict let expression} \end{array}$$

Figure 2.11: Type and term syntax of $\lambda^{\alpha}_{\mathbf{seq}}$, extended from Figure 2.6

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}\ (\text{SLet})$$

Figure 2.12: Typing rules of $\lambda^{\alpha}_{\mathbf{seq}}$, extended from Figure 2.7

$$\begin{array}{llll} [\![\alpha]\!]^{\mathsf{seq}}_{\theta} & = \theta(\alpha) & [\![[\tau]]\!]^{\mathsf{seq}}_{\theta} & = \mathit{lfp}(\lambda S.\ (\{[\,]\} \cup \{\mathbf{a} : \mathbf{b} \mid \mathbf{a} \in [\![\tau]\!]^{\mathsf{seq}}_{\theta}, \mathbf{b} \in S\})_{\bot}) \\ [\![\mathit{Nat}]\!]^{\mathsf{seq}}_{\theta} & = \mathbb{N}_{\bot} & [\![\tau_1 \to \tau_2]\!]^{\mathsf{seq}}_{\theta} & = (\{\mathbf{f} : [\![\tau_1]\!]^{\mathsf{seq}}_{\theta} \to [\![\tau_2]\!]^{\mathsf{seq}}_{\theta}\})_{\bot} \end{array}$$

Figure 2.13: Type semantics of $\lambda^{\alpha}_{\mathbf{seq}}$

Example 8
(strict let expression)

We aim for the following behavior of the strict let expression:

$$[\![\mathbf{let!}\ x = 0\ \mathbf{in}\ 5]\!]^{\mathsf{seq}}_{\emptyset} = \mathbf{5}$$
$$[\![\mathbf{let!}\ x = \bot_{\mathit{Nat} \to \mathit{Nat}}\ \mathbf{in}\ 5]\!]^{\mathsf{seq}}_{\emptyset} = \bot$$
$$[\![\mathbf{let!}\ x = (\lambda y :: \mathit{Nat}.\bot_{\mathit{Nat}})\ \mathbf{in}\ 5]\!]^{\mathsf{seq}}_{\emptyset} = \mathbf{5}$$

The first line should be clear. In the second line, the undefined value assigned to x forces the overall expression to be undefined, even though x is not required to evaluate 5. To allow for the result of the last line, our semantic interpretation of functions needs to be changed. The constant function to $\bot$ that is assigned to x behaves differently from $\bot$ in the function space. Up to now the terms $\lambda x :: \mathit{Nat}.\bot_{\mathit{Nat}}$ and $\bot_{\mathit{Nat} \to \mathit{Nat}}$ were mapped to the same semantic function, but $\lambda x :: \mathit{Nat}.\bot_{\mathit{Nat}}$ is a weak head normal form, while $\bot_{\mathit{Nat} \to \mathit{Nat}}$ is not and cannot be evaluated to weak head normal form.

As shown in Example 8 the terms $\lambda x :: \tau_1.\bot_{\tau_2}$ and $\bot_{\tau_1 \to \tau_2}$ are distinguishable w.r.t. their evaluation behavior when we introduce forced strict evaluation as intended. Hence, the terms must have different semantic interpretations. In $\lambda^{\alpha}_{\mathbf{fix}}$ both are indistinguishable and interpreted as the least value of the function space, i.e., the constant function to the least value of the result type. We only denoted this function by $\bot$, but it was a real function, e.g. $\lambda\mathbf{x}.\bot$ in $[\![\mathit{Nat} \to \mathit{Nat}]\!]^{\mathsf{fix}}_{\emptyset}$. Regarding strict evaluation, we lift the function space to distinguish the constant function to the least value of result type and a newly added, even less defined, value. The adjusted type semantics is shown in Figure 2.13. The function space remains as in $\lambda^{\alpha}_{\mathbf{fix}}$ ordered point-wise and restricted to monotone and continuous functions. It is only enriched with a new least element $\bot$.

altered semantics:

$$[\![t_1\ t_2]\!]^{\mathbf{seq}}_\sigma = [\![t_1]\!]^{\mathbf{seq}}_\sigma \ \$\ [\![t_2]\!]^{\mathbf{seq}}_\sigma$$

$$[\![\mathbf{fix}\ t]\!]^{\mathbf{seq}}_\sigma = \bigsqcup_{n\geqslant 0}\ (([\![t]\!]^{\mathbf{seq}}_\sigma\ \$)^n\ \bot)$$

extension of the semantics:

$$[\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]^{\mathbf{seq}}_\sigma = \begin{cases} [\![t_2]\!]^{\mathbf{seq}}_{\sigma[x\mapsto \mathbf{a}]} & \text{if } [\![t_1]\!]^{\mathbf{seq}}_\sigma = \mathbf{a} \neq \bot \\ \bot & \text{if } [\![t_1]\!]^{\mathbf{seq}}_\sigma = \bot \end{cases}$$

For all unmentioned cases only replace $[\![\cdot]\!]^{\mathbf{fix}}_\cdot$ by $[\![\cdot]\!]^{\mathbf{seq}}_\cdot$.

Figure 2.14: Term semantics of $\lambda^\alpha_{\mathbf{seq}}$, changed and extended from Figure 2.9

The term semantics of $\lambda^\alpha_{\mathbf{seq}}$ also differs from the one of $\lambda^\alpha_{\mathbf{fix}}$. In particular, function application has changed. The artificial least element added to the function space is not a mathematical function at all, hence we cannot apply it to a value. The problem is solved by a new operator for function application, $\$$.

Definition 14 (Operator $\$$)

Let D_1, D_2 pcpos, $\mathbf{f} \in (D_1 \to D_2)_\bot$ and $\mathbf{x} \in D_1$. We define

$$\mathbf{f}\ \$\ \mathbf{x} = \begin{cases} \mathbf{f}\ \mathbf{x} & \text{if } \mathbf{f} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

The semantic changes for terms compared to $\lambda^\alpha_{\mathbf{fix}}$ are given in Figure 2.14.

Again, we need to ensure that the altered semantics is well-defined.

Lemma 8

If $\Gamma \vdash t :: \tau$ valid in $\lambda^\alpha_{\mathbf{seq}}$, then for all θ and σ with

- $\Gamma_T \subseteq dom(\theta)$ and
- $(x :: \tau') \in \Gamma_V \Rightarrow \sigma(x) \in [\![\tau']\!]^{\mathbf{seq}}_\theta$

we have $[\![t]\!]^{\mathbf{seq}}_\sigma \in [\![\tau]\!]^{\mathbf{seq}}_\theta$.

Proof. Induction over the type derivation. Special care has to be taken concerning continuity. □

2.3.2 Changes to the Parametricity Results

By the new feature added to the calculus, we can test for every type if a term evaluates to $\bot$. Hence, stressing our view on polymorphic types as abstract data types, the completely polymorphic type provides a new operation in its interface: a definedness check. In contrast to the introduction of **fix**, we can now check a property of a value of polymorphic type instead of producing a value. As a consequence a new restriction, called *bottom-reflectingness*, is required to

$$\Delta^{\mathbf{seq}}_{\tau_1\to\tau_2,\rho} = \{(\mathbf{f},\mathbf{g}) \mid \mathbf{f} = \bot \text{ iff } \mathbf{g} = \bot, \forall(\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{seq}}_{\tau_1,\rho}.\ (\mathbf{f}\ \$\ \mathbf{a}, \mathbf{g}\ \$\ \mathbf{b}) \in \Delta^{\mathbf{seq}}_{\tau_2,\rho}\}$$

$\boxed{\text{For all unmentioned cases only replace } \Delta^{\mathbf{fix}}_{\cdot,\cdot} \text{ by } \Delta^{\mathbf{seq}}_{\cdot,\cdot}.}$

Figure 2.15: Logical relation for $\lambda^{\alpha}_{\mathbf{seq}}$, altered from Figure 2.10

guarantee parametricity results. The intuition behind the additional restriction is quite simple: To guarantee the parametricity theorem, the logical relation has to respect the interface of the polymorphic type when relating values. Since now a polymorphic function can check if its polymorphic input is undefined or not, the logical relation may relate $\bot$ to $\bot$, but never to a non-$\bot$ value.

Definition 15 (bottom-reflectingness)

A relation $\mathcal{R}$ between two pcpos is *bottom-reflecting* if

$$(\mathbf{x},\mathbf{y}) \in \mathcal{R} \Rightarrow (\mathbf{x} = \bot \Leftrightarrow \mathbf{y} = \bot)$$

Unfortunately, it is not sufficient to require type variables to be mapped to bottom-reflecting (and strict and continuous) relations to guarantee the logical relation, as defined in Figure 2.10, to be bottom-reflecting on every type. The lifting for function types must be adjusted as well (and in a more fundamental way than by inserting only the new function application symbol \$). The relation $\Delta^{\mathbf{fix}}_{\tau_1\to\tau_2,\rho}$ relates all functions that yield related results for related arguments. With $[\![\tau_1 \to \tau_2]\!]^{\mathbf{seq}}_{\theta}$ as function space, in particular $\lambda x.\bot$ and $\bot$ would be related, i.e., $\Delta^{\mathbf{fix}}_{\tau_1\to\tau_2,\rho}$ is not bottom-reflecting w.r.t. the type semantics of $\lambda^{\alpha}_{\mathbf{seq}}$ and we have to explicitly enforce bottom-reflectingness of the relation for function types. The appropriately adjusted logical relation for $\lambda^{\alpha}_{\mathbf{seq}}$ is presented in Figure 2.15.

With the new definition the logical relation is bottom-reflecting, strict and continuous, if ρ maps to bottom-reflecting, strict and continuous relations. For two pcpos D_1 and D_2 we denote the bottom-reflecting, strict and continuous relations between D_1 and D_2 by $Rel^{\top}(D_1, D_2)$, and the collection of $Rel^{\top}(D_1, D_2)$ over all pcpos D_1 and D_2 by $Rel^{\top}$. That the lemma below holds only with the extra bottom-reflectingness condition on the logical relation's lifting for function types was observed by Johann and Voigtländer (2004).

$Rel^{\top}$/$Rel^{\top}(D_1, D_2)$

Lemma 9

If ρ maps into $Rel^{\top}$, then $\Delta^{\mathbf{seq}}_{\tau,\rho} \in Rel^{\top}$ for all τ closed under $dom(\rho)$.

Proof. Induction on the structure of τ. □

The new logical relation allows to prove the following parametricity theorem.

If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{seq}}$, then for every θ_1, θ_2, ρ, σ_1, σ_2 such that

- for every α occurring in Γ, $\rho(\alpha) \in Rel^{\top}(\theta_1(\alpha), \theta_2(\alpha))$, and
- for every $x :: \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathbf{seq}}_{\tau',\rho}$,

we have $([\![t]\!]^{\mathbf{seq}}_{\sigma_1}, [\![t]\!]^{\mathbf{seq}}_{\sigma_2}) \in \Delta^{\mathbf{seq}}_{\tau,\rho}$.

Theorem 3 (Parametricity Theorem for $\lambda^{\alpha}_{\mathbf{seq}}$, Johann and Voigtländer (2004))

Proof. We only point out differences to the proof of Theorem 2. Of course, we have (SLet) as extra case:

(SLet)

By the first premise of (SLet) we have $([\![t_1]\!]^{\mathbf{seq}}_{\sigma_1}, [\![t_1]\!]^{\mathbf{seq}}_{\sigma_2}) \in \Delta^{\mathbf{seq}}_{\tau_1,\rho}$ and by $\Delta^{\mathbf{seq}}_{\tau_1,\rho}$ bottom-reflecting (Lemma 9) we have either

(a) $[\![t_1]\!]^{\mathbf{seq}}_{\sigma_1} = [\![t_1]\!]^{\mathbf{seq}}_{\sigma_2} = \bot$, or

(b) $[\![t_1]\!]^{\mathbf{seq}}_{\sigma_1} = \mathbf{a} \neq \bot$ and $[\![t_1]\!]^{\mathbf{seq}}_{\sigma_2} = \mathbf{b} \neq \bot$.

In case *(a)* we reason as follows.

$$\begin{array}{l} ([\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]^{\mathbf{seq}}_{\sigma_1}, [\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]^{\mathbf{seq}}_{\sigma_2}) \in \Delta^{\mathbf{seq}}_{\tau,\rho} \\ \Leftrightarrow \quad \{ \text{ term semantics and } (a) \} \\ (\bot, \bot) \in \Delta^{\mathbf{seq}}_{\tau,\rho} \end{array}$$

The last statement holds by $\Delta^{\mathbf{seq}}_{\tau,\rho}$ strict (Lemma 9).
In case *(b)* we reason as follows.

$$\begin{array}{l} ([\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]^{\mathbf{seq}}_{\sigma_1}, [\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]^{\mathbf{seq}}_{\sigma_2}) \in \Delta^{\mathbf{seq}}_{\tau,\rho} \\ \Leftrightarrow \quad \{ \text{ term semantics and } (b) \} \\ ([\![t_2]\!]^{\mathbf{seq}}_{\sigma_1[x\mapsto\mathbf{a}]}, [\![t_2]\!]^{\mathbf{seq}}_{\sigma_2[x\mapsto\mathbf{b}]}) \in \Delta^{\mathbf{seq}}_{\tau,\rho} \end{array}$$

The last statement holds by the induction hypothesis of the second premise and $(\mathbf{a}, \mathbf{b}) \in \Delta^{\mathbf{seq}}_{\tau_1,\rho}$ by *(b)*.

The remaining cases are nearly identical as in the proof of Theorem 2. Only cases involving function types require tiny changes. All these changes are straightforward and we omit stating them here explicitly. □

The changes in the parametricity theorem of course also change free theorems. In particular, the bottom-reflectingness restriction on relations has to carry over to the functions that are instantiated for the relations, i.e., these functions have to yield a non-$\bot$ value for each non-$\bot$ input. Functions that satisfy this property are called *total*. We again exemplify the changes to free theorems by an adaptation of Example 4, now using Theorem 3.

total function

Example 9 (derivation of a functional free theorem)

Let $\alpha \vdash f :: [\alpha] \to [\alpha]$, then Theorem 3 states

$$\forall S_1, S_2 \text{ sets}, \mathcal{R} \in Rel^\top(S_1, S_2).\ (\llbracket f \rrbracket_\emptyset, \llbracket f \rrbracket_\emptyset) \in \Delta^{\mathbf{seq}}_{[\alpha]\to[\alpha],[\alpha\mapsto\mathcal{R}]}$$

By the definition of the logical relation for function types we obtain

$$\begin{aligned}&\forall S_1, S_2 \text{ sets}, \mathcal{R} \in Rel^\top(S_1, S_2), (\mathbf{x}, \mathbf{y}) \in \Delta^{\mathbf{seq}}_{[\alpha],[\alpha\mapsto\mathcal{R}]}.\\ &\quad (\llbracket f \rrbracket_\emptyset\ \mathbf{x}, \llbracket f \rrbracket_\emptyset\ \mathbf{y}) \in \Delta^{\mathbf{seq}}_{[\alpha],[\alpha\mapsto\mathcal{R}]}\end{aligned}$$

and specializing S_1 to $\llbracket \tau_1 \rrbracket_\emptyset$, S_2 to $\llbracket \tau_2 \rrbracket_\emptyset$, and $\mathcal{R}$ to (the graph of) a strict and total function $\llbracket g \rrbracket_\emptyset$ with $g :: \tau_1 \to \tau_2$, we get

$$\begin{aligned}&\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2 \text{ strict and total}, (\mathbf{x}, \mathbf{y}) \in \Delta^{\mathbf{seq}}_{[\alpha],[\alpha\mapsto\llbracket g \rrbracket_\emptyset]}.\\ &\quad (\llbracket f \rrbracket_\emptyset\ \mathbf{x}, \llbracket f \rrbracket_\emptyset\ \mathbf{y}) \in \Delta^{\mathbf{seq}}_{[\alpha],[\alpha\mapsto\llbracket g \rrbracket_\emptyset]}\end{aligned}$$

Note that $\llbracket g \rrbracket_\emptyset$ really has to be strict and total. Otherwise its graph is not in $Rel^\top$, in particular not strict and bottom-reflecting.

Unfolding the list lifting (plus term semantics definition) we obtain

$$\begin{aligned}\Delta^{\mathbf{seq}}_{[\alpha],[\alpha\mapsto\llbracket g \rrbracket_\emptyset]} &= lfp(\lambda R.\ \{(\bot,\bot), ([\,],[\,])\} \cup \{(\mathbf{x} : \mathbf{xs}, \llbracket g \rrbracket_\emptyset\ \mathbf{x} : \mathbf{ys}) \mid\\ &\qquad (\mathbf{xs}, \mathbf{ys}) \in R\})\\ &= \{(\mathbf{xs}, \llbracket map\ g \rrbracket_\emptyset\ \mathbf{xs}) \mid \mathbf{xs} \in \llbracket [\tau_1] \rrbracket\}\end{aligned}$$

If we apply that equivalence twice (plus term semantics definition) we get

$$\begin{aligned}&\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2 \text{ strict and total}, xs :: [\tau_1].\\ &\quad \llbracket map\ g\ (f\ xs) \rrbracket_\emptyset = \llbracket f\ (map\ g\ xs) \rrbracket_\emptyset\end{aligned}$$

which, besides the additional strictness and totality restriction, is statement (1.1) from the introduction where it was left implicit that we mean semantic equivalence.

2.4 Explicit Type Abstraction and Instantiation

The calculi presented so far do not allow for explicit type abstraction and instantiation as the polymorphic λ-calculus investigated by Wadler (1989) does. Adding abstraction and instantiation syntax for terms and $\forall$-quantifiers to the type syntax would be possible for some of our investigations. However, for this thesis, we refrain from that generalization because it is either straightforward or not possible for the respective problems we investigate.[17] Hence, we only point out when an extension of the calculus is possible and when not.

[17] Just as side note, a maybe not so obvious problem: In Section 2.1 the extension to explicit type abstraction and instantiations would render the interpretation of types as sets impossible (Reynolds, 1984).

Nevertheless, the handling of type instantiation is essential to correctly state free theorems the way we want. For example, consider statement (1.1) from the introduction and the various versions of it that are derived in the Examples 4, 7 and 9. Intuitively, everything works fine. But a technical detail is swept under the carpet: Function f has the polymorphic type $[\alpha] \to [\alpha]$ and we apply it to arguments of monotypes (types $[\tau_1]$ and $[\tau_2]$). Hence the applications of f are not typeable in our calculi. Hence, regarding the semantics of those applications is formally incorrect. But: "morally" it is correct. Function f can be typed differently, in particular the type variable α can be substituted by any other type without affecting the semantics of f. The semantic interpretation of terms in the calculi do not rely on type information.

To get statements that are formally correct, we might try to type f differently right from the beginning — but then we lose the parametricity results that rely on polymorphism. But, we also achieve formally correct statements if we introduce a mechanism for substituting type variables by *monotypes*, i.e., types without type variables.[18] The mechanism is introduced by the next definition and its correctness is verified by the following lemma.

monotype

Definition 16 (type substitution)

Let τ type and τ' monotype, t term and α type variable. Then $\tau\,[\tau'/\alpha]$ and $t[\tau'/\alpha]$ denote the *substitution*, i.e., syntactical replacement, of the type variable α by τ' in τ and in (the type annotations of) t, respectively.
The definition extends to typing contexts the following way. For a typing context Γ the typing context $\Gamma[\tau'/\alpha]$ includes the same type variables as Γ except for α, and $(x :: \tau) \in \Gamma \Leftrightarrow (x :: \tau[\tau'/\alpha]) \in \Gamma[\tau'/\alpha]$.

The substitution of type variables satisfies the following lemma.

Lemma 10

Let $\Gamma \vdash t :: \tau$ valid, α type variable, τ' monotype and $dom(\theta) \supseteq \Gamma_{\mathcal{T}} \setminus \{\alpha\}$. We have

- $[\![\tau]\!]^{\cdot}_{\theta[\alpha\mapsto[\![\tau']\!]^{\cdot}_{\emptyset}]} = [\![\tau[\tau'/\alpha]]\!]^{\cdot}_{\theta}$
- $([\![t]\!]^{\cdot}_{\sigma} \in [\![\tau]\!]^{\cdot}_{\theta[\alpha\mapsto[\![\tau']\!]^{\cdot}_{\emptyset}]}) \Rightarrow ([\![t]\!]^{\cdot}_{\sigma} = [\![t[\tau'/\alpha]]\!]^{\cdot}_{\sigma})$

with $[\![\cdot]\!]^{\cdot}_{\cdot}$, either $[\![\cdot]\!]_{\cdot}$, $[\![\cdot]\!]^{\mathsf{fix}}_{\cdot}$ or $[\![\cdot]\!]^{\mathsf{seq}}_{\cdot}$.

Proof. The first assertion is (for each calculus) proved by induction on the structure of τ while the second assertion is (again for each calculus) proved by induction on the structure of t. □

[18] The requirement of monotypes is not necessary, but sufficient for our purposes.

The well-typed replacement of (1.1) from the introduction is

$$\begin{aligned}&\forall f :: [\alpha] \to [\alpha], \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, xs :: \tau_1.\\ &\qquad f[\tau_2/\alpha]\ (map[\tau_1/\alpha, \tau_2/\beta]\ g\ xs) \equiv map[\tau_1/\alpha, \tau_2/\beta]\ g\ (f[\tau_1/\alpha]\ xs)\end{aligned}$$

Via the substitution of type variables, we cover polymorphism already by the simply typed λ-calculus. Of course, we do not cover arbitrary type quantification as the polymorphic λ-calculus regarded by Reynolds (1983) and Wadler (1989), but at least so called prenex or rank-1 polymorphism (where type quantifiers are, or can be brought, always in front of a term's type and, as a consequence, all arguments to terms of function type must already be instantiated to monotypes), as also standard in Haskell 98 (Peyton Jones, 2003) and Haskell 2010 (Haskell, 2010)[19].

In the remaining chapters, we often omit the explicit instantiation of types because the concrete instantiations are usually clear from the context.

[19]GHC 7.6.3 (and also earlier versions) provide extensions to arbitrary-rank polymorphism, the most general extension is activated by the flag `-XRankNTypes` (GHC Team, 2013, Section 7.12.5).

Chapter 3

State of the Art

We give an overview of the formal development and the applications of the theory of parametricity and free theorems. The original results in the PhD thesis (presented in Part II) are excluded.

In the last few years the theory of parametricity has been adjusted to various settings and applied there. Hence, there are plenty of papers and we do not claim the provided overview to be complete. Nevertheless, we want to broaden our view further than to the work directly relevant to the investigations that are original in this thesis. Thus, we look at results that show development and application of the theory of parametricity in manifold settings.

In Section 3.1 we review the theoretical developments of free theorems, while in Section 3.2 we talk about applications.

3.1 Theoretical Developments of Free Theorems

We structure this section by the technical machinery that is employed in the referenced publications. Closest related to the research original in this thesis are works based on a denotational semantics. In particular, works that consider a non-strict language are of interest. All referenced works that are based on a denotational semantics do consider a non-strict setting.[1] Among those references, we consider the seminal publications on the whole topic of relational parametricity. These papers are discussed in Subsection 3.1.1. To broaden our view, we also regard results based on operational semantics (Subsection 3.1.2), as well as on pure type systems (Subsection 3.1.3).

[1]If the evaluation strategy is of interest at all, which is not the case if all programs terminate and if no other error or failure can occur, e.g. in System F.

Not all of the referenced works focus directly on free theorems, especially when it comes to the investigation of strict languages in an operational setting. The link to free theorems is the logical relation and a corresponding parametricity theorem that directly allows to establish free theorems. But most of the works aim for a logical relation not to derive free theorems, but to define an alternative notion of equivalence, easier to handle than the standard (and easy to define) notions of contextual or observational equivalence. In such references, free theorems are more a byproduct than the focus and the logical relation is proved to satisfy an even stronger property than the parametricity theorem we establish: It holds not only that the semantics of each term is related to itself, but also that only semantically equivalent terms are related. The second direction would be desirable in general but it is at least difficult, if not impossible, to establish it in a denotational semantics. For an explanation, see for example (Voigtländer and Johann, 2007), where an operational semantics is employed mainly to get grip on the second direction of the above assertion.

3.1.1 Results Building on a Standard Denotational Semantics

The formalization of parametricity goes back to John C. Reynolds. Reynolds (1974) considers the polymorphic λ-calculus (i.e., a calculus similar to the one presented in Section 2.1 but with explicit type abstraction and instantiation and without any lists and Nat) as illustrative language for his work. He explores a way to formalize representation independence. That is, as already described in Section 2.1.2, that the concrete implementation of a data type is irrelevant as long as it satisfies certain conditions on its interface. This should hold for base types as well as for user defined types, where we can see the implementation as an "inner" region that is not accessible in a program, and the interface as the "outer" region, that is accessible and the only connection to the "inner" part. Representation independence now means that the "inner" part of the data type might change without affecting the overall program as long as the behavior "visible" via the interface remains static. To formalize when two type interpretations are interchangeable he regards a denotational semantics and defines a set of representations between two interpretations (each a tuple of functions, from one interpretation to the other and back) that guarantee the visible behavior to remain static when switching from one representation to the other. As we lift the relation between base types for composed types like the function type, he lifts the set of representations. In the end, he gains the so-called representation theorem, a theorem that differs from the parametricity theorem (as for example Theorem 1) mainly in the way that it quantifies over a set of representations instead of relations. Interestingly, the idea of abstraction via types does in the first place not directly focus on polymorphic types in the programming language itself. Instead, the initial idea is that even each base or monomorphic user defined type leaves space for different concrete implementations up to a certain point, i.e., up to changing the "external behavior". Thus each type is regarded as an abstract type. Despite that fact, it is worth noting that Reynolds (1974) invented the λ-calculus with explicit abstraction and instantiation of types. Since, independently and with substantially different motivation, Girard (1972) developed the same calculus, it is not only known as polymorphic λ-calculus or System F, but also as Girard-Reynolds calculus. Another name is second order λ-calculus because of the connection to second order intuitionistic logic.

Reynolds (1983) extended his results on representation independence, finally stating the abstraction theorem, or, as we introduced it (Theorem 1, Section 1), parametricity theorem. In Section 6 he proves that the switch from representations to relations is a generalization and in Section 8 he focuses on the characterization of polymorphic functions via the abstraction theorem, finding the following essential link:

"The abstraction theorem guarantees that, in an environment in which all polymorphic functions are parametric, the meaning of any ordinary expression will be parametric."
Reynolds (1983, Section 8)

Following the development from Reynolds' papers to free theorems, we directly reach the eponymous paper for free theorems: "Theorems for Free!" (Wadler, 1989). Wadler's basic insight was that Reynolds' abstraction theorem allows the derivation of useful statements about polymorphic functions, independent of the concrete function definition, only relying on the function's type. So Wadler calls his main contribution to recognize that parametricity[2] *"has 'specific' applications: it says interesting things about particular functions with particular types."* Independently, de Bruin (1989) observed the same kind of applications.

Wadler (1989) investigated the polymorphic λ-calculus and also explained how type classes (Wadler and Blott, 1989) can be handled. In the last section (Section 7) of his paper he additionally considered the restrictions that arise when a fixpoint primitive, like the function **fix** in Section 2.2, is added. Such kind of restrictions that arise through various programming features are exactly what we deal with in this thesis.

While Wadler (1989) considers the influence of general recursion only globally (as we do in Section 2.2), Launchbury and Paterson (1996) localized the influence by the distinction between pointed and unpointed types. Thereby, they potentially reduce the arising restrictions on free theorems, depending on the concrete use of general recursion.

Effects on parametricity results due to a strictness primitive that allows the programmer to influence the evaluation strategy have been noticed already when the strictness primitive **seq** was added to Haskell (version 1.3) (Peterson et al., 1996). To make the necessity of additional restrictions observable by the type, the type class Eval was introduced. It should track (problematic) uses of **seq**. In their paper about the history of Haskell, Hudak et al. (2007, Section 10.3)

[2]Wadler mentions that the notion "parametricity" is taken from Bainbridge et al. (1990) and Freyd et al. (1988).

explain how and why **seq** was introduced into Haskell, and how and why its type changed. The type class Eval is not part of the Haskell 98 standard (Peyton Jones, 2003) (and also of later standards) anymore. It was removed because it complicated program optimization: Adding **seq** usually caused many changes in type signatures. Besides that practical disadvantage, the type class was also not suitable to appropriately restrict free theorems w.r.t. forced strict evaluation. Appropriate restrictions were for the first time presented by Johann and Voigtländer (2004). The authors set up an inequational logical relation, stating only that one side of a free theorem is less or equally defined than the other side. One-sided assertions require less extra-conditions and can be combined to establish equational statements. To achieve inequational theorems, the chosen relations differ from the ones given in Section 2.3 of this thesis. A key property enforced for the relations is left-closedness: A binary relation $\mathcal{R}$ between two posets is left-closed iff $(\mathbf{x}, \mathbf{y}) \in \mathcal{R}$ implies for all $\mathbf{x}' \sqsubseteq \mathbf{y}$ that $(\mathbf{x}', \mathbf{y}) \in \mathcal{R}$. Concerning equational statements, that are easily derivable from the inequational ones, the theory matches the results presented in Section 2.3. The paper also considers the influence of **seq** on program transformations proposed for Haskell and relying on free theorems. A more comprehensive version of the conference paper of 2004 has been published in Fundamenta Informaticae in 2006 (Johann and Voigtländer, 2006).

Besides the just explained features, several others have been explored concerning their influence on parametricity and hence on free theorems. For lazy languages, research mainly focuses on the exact modeling of features present in the programming language Haskell. Thus, Stenger and Voigtländer (2009) consider free theorems when errors have to be preserved according to Haskell's imprecise error semantics (Peyton Jones et al., 1999). They extend a calculus with selective strict evaluation, like the one presented in Section 2.3, such that different errors can be raised. According to the imprecise error semantics, errors can accumulate. The main difference compared to (Johann and Voigtländer, 2004) is the replacement of strictness by a notion of error-strictness and the replacement of bottom-reflection by a suitable notion of error-reflection. To guarantee these properties for the logical relation, also slight adjustments of the relational liftings are necessary.

Not a new theoretical development, but a good tool when applying free theorems is provided by Böhme (2007). He developed a generator for free theorems. It enables the automatic generation of equational or inequational free theorems in different language settings (with/without general recursion and/or selective strict evaluation). The generator is available at http://www-ps.iai.uni-bonn.de/ft .

Type constructor classes were considered informally by Voigtländer (2009b). The results are w.r.t. a calculus without selective strictness and differentiated error handling. They are accentuated by a lot of examples.

As far as we know, previously to the results presented in this thesis, and the corresponding publications, there have been no attempts to equip free theorems with information about evaluation costs.

3.1.2 Results Building on an Operational Semantics

Results for Non-Strict Languages

Pitts (2000) considered the polymorphic λ-calculus extended with a fixpoint primitive and lists. In his work he establishes a syntactic logical relation, i.e., a relation between terms, that he shows to coincide with observational equivalence (his Theorem 4.15). Furthermore, he clarifies the relation to other definitions of equivalence, in particular contextual equivalence, Kleene equivalence and ciu equivalence (closed instantiations of uses). Specifying the evaluation of a program via evaluation frames nested in each other via a frame stack and giving rules how terms under a certain stack are to be evaluated, his key technical contribution is the notion of $\top\top$-closure. In Pitts' work the notion is central to restrict the logical relation in the appropriate way to handle fixpoints. As also apparent in the denotational setting presented in Section 2.2, the logical relation must satisfy certain properties. While in the denotational setting we enforced strictness and continuity, together called admissibility, on each type, Pitts (2000) uses $\top\top$-closedness to guarantee admissibility. In contrast to our denotational investigations, Pitts (2000) establishes a stronger result: He proves not only that any two observationally equivalent terms are related, he also proves that *only such terms* are related (while we only prove that terms with the same denotational semantics are related). Thus, his result is even stronger than the parametricity theorems we establish, and of course can, as Pitts already states, be employed to establish free theorems.

Based on Pitts' work Voigtländer and Johann (2007) transfer their results from Johann and Voigtländer (2004, 2006) to an operational setting. They reach a stronger result than in the denotational setting in the way that they prove coincidence between observational equivalence and relatedness via the logical relation. The addition of selective strictness to the calculus used by Pitts (2000) leads (as expected from the denotational results) to a further restriction on the logical relation: It has to be convergence-preserving. Generalizing Pitts' work, Voigtländer and Johann (2007) start their investigations with a notion of observational approximation instead of equivalence and build up relations that lead to approximation statements.

How to gain formal parametricity results in an operational setting with general recursion, selective strictness and different errors is presented by Johann and Voigtländer (2009). The logical relation is adjustable via a parameter to fit for different, language specific, error handlings.

Results for Strict Languages

The works concerning strict languages consider various features, mainly inspired by the ones available in strict functional programming languages like ML and its different dialects. The main focus is on state and control effects. As already mentioned, the works do not directly aim for free theorems in the different settings, rather the main interest is to establish an easy to handle notion of "semantic equivalence". But of course the achieved logical relations allow to establish free theorems.

Before we point to several recent works, we clarify two problems arising from different language extensions, that lead to special kinds of logical relations. The first problem is well-foundedness of the logical relation's definition in the presence of general recursive types. Relational interpretations become cyclic and therefore difficult to define inductively. *Step-indexed logical relations* are an appropriate solution to this problem. The idea of step-indexing stems from Appel and McAllester (2001) to get recursive types under control and is motivated as follows. We can observe if a term keeps a property if it is evaluated step-wise. The step-index indicates how far we track its behavior. For a binary step-indexed logical relation this means that two terms related by the k-indexed relation "look" related for up to k evaluation steps. Obviously, they are related if so for all $k \in \mathbb{N}$ and thus step-indexing provides a clear and well-founded induction scheme.

The second problem awaiting a solution is state. Clearly, state is not an internal property of a term but an external configuration, a term's evaluation may depend on — and consequently also the equivalence of different terms may depend on. Thus, one has to consider all relevant states terms are evaluated in. Particularly interesting are state invariants, arising from local states. Keeping those invariants is essential to identify semantically equivalent terms. As an example[3], consider the terms

$$
\begin{aligned}
e_1 &= \textbf{let}\ x = \mathit{ref}\ 1\ \textbf{in}\ (\lambda f \to (f\ (); !x)) \\
e_2 &= \lambda f \to (f\ (); 1)
\end{aligned}
$$

where x is a reference cell, filled with 1 by the call $\mathit{ref}\ \ 1$, ; is the sequence operator and by the prefix operator ! the value stored in a reference cell is accessed. To show that both terms of type $(() \to ()) \to \mathit{Nat}$ are equivalent, it is essential that f cannot alter the content of the reference cell x, i.e., that we assume the "x maps to 1" as an invariant.

A work by Dreyer et al. (2012)[4] considers an ML-like language with iso-recursive types, abstract types, general references (state) and call/cc (control). A fully abstract step-indexed Kripke logical relation is defined, and can moreover be adjusted to restrictions of the full language, namely to a language with first-order state only and/or a language without call/cc. The technical main

[3]The example is taken from Dreyer et al. (2012).
[4]A much shorter conference version appeared at ICFP 2010 (Dreyer et al., 2010a).

contribution of the paper is the use of state transition systems that express the evolution of state over time.

Beside the state transition systems for modeling and the very systematic consideration that allows enabling or disabling features such as higher order state or control, the other ideas presented by Dreyer et al. (2012) were already present earlier in papers. In particular, Dreyer et al. (2012) build on the work of Ahmed et al. (2009), who were the first to consider a language that combines existential types and local state. In their work, the use of a step-indexed Kripke logical relation is the new key technical feature. A remarkable advantage of their new technical treatment, combining a possible world model (Kripke relations) with step-indexing, manifests in the ability to handle arbitrary mutable references, particularly, higher-order references. Preceding works (that we will not discuss here) either completely exclude higher-order references (Benton and Leperchey, 2005; Pitts and Stark, 1998; Reddy and Yang, 2003), or enforce a highly stylized form of local parameters that store relations have to be expressible in (Bohr and Birkedal, 2006).

The idea of step-indexing provides for the design of appropriate logical relations in many settings and is an easy to understand concept. But it also results in annoying proof-blurring index handling, that is only owed to the technique. Dreyer et al. (2010b) set out to remedy the technical deficiencies by still building on step-indexing to set up the logical relation, but eliminating step-index caused overhead in proofs employing the relation designed. They propose a logic, called LADR, for equational reasoning about higher-order programs. As features, they consider existential type abstraction, general recursive types, and higher-order mutable state. The work directly builds on the works of Ahmed et al. (2009) and Dreyer et al. (2009), where the first one provides (up to some minor changes) the logical relation considered, and the second work forms the basis for building up a logic to reason about the logical relation — without being concerned about step-indices.

Another direction in research about parametricity sets out to investigate parametricity in dynamically typed languages, or mixed statically and dynamically typed languages.

Ahmed et al. (2011) formulate a strict-evaluating polymorphic lambda calculus where relational parametricity is enforced by a kind of dynamic sealing. The focus is on combining typed and untyped program parts, where “untyped” is represented by “typeable to the dynamic type”, a type everything is typeable to. Instead of type instantiation, the calculus keeps type bindings locally and programs with type mismatches quit the evaluation by blaming the type mismatch. The fact that type errors are blamed combined with sealing by explicitly kept type bindings, restores relational parametricity, as is shown by examples. A result about relational parametricity is not explicitly stated, but announced as forthcoming and characterized as an adaptation of a result by Matthews and Ahmed (2008).

Matthews and Ahmed (2008) regard a multi-language system: They combine System F (they call ML) and an untyped call-by-value λ-calculus (they call Scheme). Via boundaries, one calculus embeds the other and vice versa. To support parametric polymorphism, they dynamically seal values that are polymorphic in ML when they cross the boundary to Scheme. ML then unseals the value calculated by Scheme when it crosses back the boundary, or, if unsealing fails, returns an error. In the end, the role of ML is reduced in the way that only the ML type system is employed to make up contracts that pure Scheme programs comply with.

Starting with a single, nonparametric language — System F extended by dynamic type generation and a type cast — Neis et al. (2009) set up logical relations that allow to derive parametricity results even in the nonparametric setting. The work is closely related to Matthews and Ahmed (2008), but technically dynamic type generation takes over the role of dynamic sealing. A new concept are polarized logical relations. The idea is to distinguish if a term's behavior or its usage is parametric. The distinction allows to handle subtle cases, for example when two implementations of an abstract data type should be proved equivalent (Neis et al., 2009, Section 8).

Notice that Neis et al. (2009) consider a calculus where the evaluation strategy is not of interest, hence we could have referred to their work also as a work for lazy languages.

3.1.3 Formalization of Parametricity in Pure Type Systems

An interesting series of papers about parametricity has been published within the scope of Jean-Philippe Bernardy's PhD thesis (Bernardy, 2011). He (and his co-authors) employ pure type systems (PTSs) (Barendregt, 1992, Section 5.2) not only as a framework to express typed lambda calculi in a unified way, but also to formalize parametricity statements and their proofs in such PTSs. The formalization becomes possible by the propositions-as-types (and proofs-as-terms) interpretation. In essence, Bernardy formalizes a λ-calculus in a (source) PTS and defines a corresponding (target) PTS where all terms[5] of the source PTS have a relational image. The target PTS is viewed as a logic in which the parametricity statements valid in the source PTS are formalized. To formalize parametricity results in a logic is not a new idea (e.g. Plotkin and Abadi (1993)), but the uniform formalization of the calculus and the corresponding parametricity results via PTSs is original. Moreover, the framework characteristics of the approach is remarkable. Bernardy (2011) can handle different λ-calculi, such as the simply typed λ-calculus, System F, or the calculus of constructions (i.e., a calculus with dependent types), simply by parametrizing his theory. Also the enrichment of a calculus with data types, the consideration of type classes and (at the same time) of type constructors is possible.

[5]Since PTSs blur the distinction between terms and types, by terms we also mean types here.

We have not classified Bernardy's work by a strategy of evaluation (i.e., strict or non-strict). What Bernardy fails to handle are calculi that allow for nontermination (for example via **fix**) or for strictness annotations (like **seq**). Consequently, in the calculi he can handle, the strategy of evaluation is not of interest concerning parametricity results.

3.2 Applications of Free Theorems

Parametricity has various applications. In contrast to the theoretical developments, here we only concentrate on applications of free theorems.

An essential use of free theorems is to develop program transformation rules and prove their correctness. The most classical such rule is the $foldr$ / $build$ rule, also known as short-cut deforestation or short-cut fusion, described by Gill et al. (1993). The rule is used to eliminate the creation and consumption of intermediate lists that arise when a function yielding a list is composed with a function taking a list as argument. The rule is implemented in the GHC (Glasgow Haskell Compiler) as an optimization rule. Its correctness in sublanguages of Haskell is proved via free theorems. But also its incorrectness in "real" Haskell due to the possible use of **seq** is shown. There are manifolds of generalizations and variations of similar rules that allow to remove intermediate data structures. For example, Johann (2002) proves and generalizes the $foldr$ / $build$-rule in a setting with general recursion (but without selective strictness). In a similar setting Johann (2005) provides a principled approach how to prove the correctness of free theorem based program transformations, always w.r.t. the operational semantics given by Pitts (2000). Other fusion laws relying on free theorems are presented by Svenningsson (2002) (handling accumulating parameters in consumers and zip-like functions), Fernandes et al. (2007) (handling circular programs) and Voigtländer (2002, 2008a,c,d) (removing concatenate, map and reverse; proving correctness of the $destroy$ / $build$-rule; considering and improving different fusion rules taking selective strict evaluation into account; a program transformation concerning free monads).

Also in other areas free theorems come in helpful. In analogy to Knuth's 0-1-principle (Knuth, 1973), Voigtländer (2008b) proves that algorithms for parallel prefix computation can be tested correct by only checking them for a three-valued type, whereas Voigtländer (2009a) uses free theorems for the automatic generation of a (provably well-behaved) put-function w.r.t. a given get-function in a database like setting where get provides a view out of the source database and put reembeds the possibly manipulated view. Moreover, the characterization of polymorphic functions via a monomorphic instance as done by Bernardy et al. (2010) (and as also part of the PhD thesis of Bernardy (2011)) relies heavily on parametricity results. Bernardy transforms the type of the polymorphic function to test into a canonical testing type, splitting the arguments in the ones that might be employed to construct elements of unknown type (represented as a type variable) and those able to observe elements of the unknown type, as well as a result type. All three parts are represented via a functor working on the type variable, and in particular for every type instantiated for α the constructive part of the arguments forms an (F-)algebra $F\ \alpha \rightarrow \alpha$. For testing, the initial algebra with the type component μF is chosen and hence the polymorphic type is fixed. With the new framework the results of Knuth (1973) and Voigtländer (2008b) are recapitulated and by example it is also argued on the ability to restrict inputs such that they satisfy certain non-trivial properties, e.g. restrict functions to be associative. More explicit than Bernardy et al. (2010), Christiansen and Seidel (2011) apply free theorems to find an appropriate set of monomorphic inputs to test polymorphic functions for minimal strictness. The difference between the work of Bernardy et al. (2010) and the paper of Christiansen and Seidel (2011) is that the latter considers nontermination and selective strictness, whereas the first does not.

As another field of application, free theorems can be employed to prove that the functions between the abstract, container-like, representation of lists given by Bundy and Richardson (1999) correspond to the polymorphic functions on the usual list representation. The container-like representation splits up a list into shape and content, and functions on that representation never touch an element of the content. The correspondence is also gained via category theoretic reasoning by Prince et al. (2008), which shows the close connection between the notion of naturality in category theory and the notion of parametric polymorphism. Seidel and Voigtländer (2012) extend the container-like representation from Bundy and Richardson (1999) to allow for element tests on the container structure. They apply free theorems for polymorphic functions with type class constraints to prove that functions on the extended container-like representation coincide with the respective type class restricted polymorphic functions.

As a last application, we want to mention the work of Oliveira et al. (2010). They employ the results of Voigtländer (2009b) to prove properties of so called "harmless advice" in their model of advice. By their model they provide for Haskell the mechanism of advise that is widely used in aspect-oriented programming.

Part II

New Results

The main focus of this thesis is the investigation of the interplay of free theorems and programming features of real world functional programming languages.

In essence, a typed functional programming language can be considered as a typed λ-calculus with possibly some extra features and a lot of syntactic sugar. So can Haskell (Peyton Jones, 2003), as an example for a real world, general purpose programming language.

A second focus of this thesis are quantitative aspects of free theorems. We consider in which way free theorems can be enriched by assertions about efficiency. This is done in a very simple setting and will leave many directions for exploration.

As formal setting we employ appropriate extensions of the simply typed λ-calculus with type variables, equipped with a denotational semantics in the style already introduced in Chapter 2.

We do not consider operational semantics. Though it is common, it clutters things more than we might benefit. Nevertheless, free theorems can be developed over an operational semantics as well, and we think that at least for further exploration of quantitative aspects a switch to an operational semantics might have benefits in the sense of more realistic runtime or space assertions. For the development of free theorems w.r.t. operational semantics, see the works of Pitts (2000) or Voigtländer and Johann (2007) for example.

Chapter 4

Exemplifying the Necessity of Strictness Conditions

[1] Concerning general recursion, effects on relational parametricity and hence free theorems have already been explored in detail. It has been shown that the logical relation is to be built up over strict, and not arbitrary, relations for type variables and the restriction has also been partly relaxed by localization of the influence of general recursion via a refined type system (cf. Section 3.1). An interesting point that remains to be investigated is the automatic generation of counterexamples to (insufficiently restricted) free theorems.

As motivation for the investigation we regard that strictness conditions are always sufficient, but not always necessary, even if lessened by the refined type system that localizes the influence of general recursion. For example, consider the type $[\alpha] \to Nat$. The free theorem, specialized to functions and aware of general recursion, yields that for all $f :: [\alpha] \to Nat$, all types τ_1, τ_2 and every *strict* function $g :: \tau_1 \to \tau_2$ the equality $f \circ (map\ g) \equiv f$ holds. But, even when we neglect the strictness condition on g we will not find f, τ_1, τ_2 and (non-strict) g such that the theorem breaks. So, aware of the necessity of strictness conditions in general (cf. statement (1.1) from the introduction) and at the same time experiencing the conditions unnecessary for some types, the question arises: For which types are certain strictness conditions superfluous? And also: Why not for others? An appropriate answer to these questions is provided by an algorithm that, given a type and information about which strictness conditions to check, either generates an instance of a free theorem that is wrong (ultimately caused by the violation of one of the considered strictness conditions), or returns without such a counterexample, stating that the considered strictness conditions are superfluous.

[1]Some of the results have been published in (Seidel and Voigtländer, 2010). Some are only found in (Seidel and Voigtländer, 2009b)

Because the instantiation of a free theorem includes several choices of functions and types (cf. statement (1.1) on page 5), an algorithm for counterexample generation needs to perform several tasks. The first and maybe most interesting one is to find a term of the (polymorphic) type we investigate, such that the term gives rise to a counterexample caused by the violation of a strictness restriction. To find a respective term we have to answer two questions:

- How to find a term of a given type in general?
- How to decide if a term might give rise to a counterexample?

Let us tackle the second question first. Assume we can find a term of a given type. How do we know that it gives rise to a counterexample? The refined type system presented by Launchbury and Paterson (1996) provides a way to identify terms that will definitely *not* give rise to a counterexample. The refined type system tracks the use of **fix** and thus every possibility to break the free theorem that lacks strictness conditions. Particularly interesting for us, from the refined type we can read off which strictness conditions can safely be dropped in a free theorem. That means, we can look for terms that are only typeable to refined types that do not allow to drop all strictness conditions. These are good candidates for counterexamples, while for other terms the unrestricted free theorems definitely hold. We present the refined type system in Section 4.2.

Consider now the first question: How to find a term of a given type in general? By brute force, we could try to employ QuickCheck (Claessen and Hughes, 2000) for random or SmallCheck (Runciman et al., 2008) for exhaustive term generation. But, because we already have quite specific type constraints on the terms we look for, and moreover whole counterexamples consist of several terms that are specially related to each other, a successful QuickCheck search would require rather elaborated generators to prevent an exploding number of test cases necessary to find a counterexample. Since elaborated generators only wrap an advanced search strategy, we can also throw away the QuickCheck idea and define a concrete term search directly.[2] The basic idea to find a concrete term directly is to apply the typing rules backwards (i.e., from bottom to top as done for type inference as well, but now without knowing the term in advance). Unfortunately, with the standard set of typing rules (also for the refined type system of Launchbury and Paterson (1996)) the approach fails. We observe the problem regarding the rule (App) (cf. Figure 2.2 on page 16):

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\ (\textsc{App})$$

Here, applying (App) backwards to derive a term for given type and context, we need to invent a type τ_1 and there is no hint how to do so. Moreover, we can repeatedly apply (App) the often we want and hence have no guarantee for termination for any term search algorithm using (App). Consequently, the standard typing rules are not suitable for term generation.

[2]However, we used QuickCheck (Claessen and Hughes, 2000) to test the implementation of our counterexample generator.

To remedy that situation, results from intuitionistic logic will help. The Curry-Howard isomorphism (Curry and Feys, 1958; Howard, 1980) states a close relationship between intuitionistic logic and the typed λ-calculus. In particular, types correspond to formulas and provability of a formula to type inhabitation. Loosely speaking, if we cut out the terms in the typing rules, we end up with the natural deduction rules for intuitionistic logic. There are several corresponding logic/calculi pairs, related via the Curry-Howard isomorphism[3]. We are interested in the pair of propositional intuitionistic logic and the simply typed λ-calculus (with type variables). For propositional intuitionistic logic Dyckhoff (1992) provides a sequent calculus for proof search, called LJT[4], that is guaranteed to terminate. Decorating the rules with appropriate λ-terms and regarding formulas as types, we obtain a term generator. Furthermore, Corbineau (2004) extends Dyckhoff's work to include inductive constructions and therefore, on the λ-calculus side, allows for algebraic data types.

Exploiting the work of Corbineau (2004) we end up with a term generator for the simply typed lambda calculus with type variables, various algebraic data types and base types. Similarly, Augustsson (2009) implemented a term generator for (a subset of) Haskell. To proceed to the generation of terms suitable for counterexamples, several adaptations to the "translated" rules are necessary. First, we have to incorporate general recursion. Of course, our intention is not to add the typing rule (FIX) here. It would — going back to the logical side — make the logic inconsistent, for we could derive every statement. That is, seen in the calculus, the term **fix** id_τ is an inhabitant of every type τ. Hence, we need to carefully restrict the introduction of **fix** during term generation. We present the translation of the original rule system of Corbineau (2004) to a term generator for a λ-calculus without **fix** in Section 4.3. In Section 4.4 we describe the necessary adaptations to the original term generator to yield terms that are potentially suitable for a counterexample to a free theorem with missing strictness conditions. The final algorithm we call $\mathrm{TermFind}$.

$\mathrm{TermFind}$

Regarding Statement (1.1) from the introduction (page 5) $\mathrm{TermFind}$ constructs the term $\lambda x :: [\alpha].[\bot_\alpha]$ that we (up to α-conversion) already chose in the introduction. But we are still missing instances for τ_1, τ_2, g and xs. To generate these instances, we define a second algorithm, called ExFind. It mainly employs the rules of $\mathrm{TermFind}$ and extends them with extra constructions. The most challenging task is to construct the function arguments. The construction takes place on the fly during the generation of f, but (not for statement (1.1) from the introduction, but in general) it causes serious difficulties when we try to extend one of the $\mathrm{TermFind}$-rules. Therefore we simplify this rule in ExFind, unfortunately losing completeness. Nevertheless, for many types ExFind still produces counterexamples and moreover the problematic cases can be characterized. The way from $\mathrm{TermFind}$ to complete counterexample generation via the algorithm ExFind, and the problems, are discussed in Section 4.5.

ExFind

[3]Depending on the logic/calculus pair one also speaks of Girard-Reynolds isomorphism, or simply types-as-formulae isomorphism.

[4]The calculus is adopted from Gentzen's LJ calculus.

τ	$::=$	$\ldots$	
	$\mid$	$()$	unit type
	$\mid$	(τ, τ)	product type
	$\mid$	$\mathit{Either}\ \tau\ \tau$	sum type
t	$::=$	$\ldots$	
	$\mid$	$()$	empty tuple
	$\mid$	$\mathbf{case}\ t\ \mathbf{of}\ \{() \to t\}$	case expression for the unit type
	$\mid$	(t, t)	tuple
	$\mid$	$\mathbf{case}\ t\ \mathbf{of}\ \{(x, x) \to t\}$	case expression for products
	$\mid$	$\mathbf{Left}_\tau\ t$	left value of a sum
	$\mid$	$\mathbf{Right}_\tau\ t$	right value of a sum
	$\mid$	$\mathbf{case}\ t\ \mathbf{of}\ \{\mathbf{Left}\ x \to t; \mathbf{Right}\ x \to t\}$	case expression for sums

Figure 4.1: Type and term syntax of $\lambda^{\alpha}_{\mathbf{fix}+}$ and $\lambda^{\alpha}_{\mathbf{fix}^*}$, extended from Figure 2.6

To get an impression of the results ExFind provides: For Statement (1.1) it yields the already mentioned term $\lambda x :: [\alpha].[\bot_\alpha]$, instantiates τ_1 and τ_2 to the unit type $()$, specializes g to $\lambda x :: ().()$ and xs to $[()]$. The concrete output presented by its implementation's web interface is shown in Figure 4.14 (page 112).

Section 4.6 summarizes the results and Section 4.7 discusses further research directions.

4.1 The Calculus

For counterexample search, we enrich the calculus $\lambda^{\alpha}_{\mathbf{fix}}$, introduced in Section 2.2, by a sum, a product and a unit type. The extended calculus covers thus the most basic data types. In this section we point out only the changes compared to $\lambda^{\alpha}_{\mathbf{fix}}$. Since there are no alterations to $\lambda^{\alpha}_{\mathbf{fix}}$, only extensions, we keep the name of the semantic function and the name of the logical relation. We call the extended calculus $\lambda^{\alpha}_{\mathbf{fix}+}$. Additional syntax of $\lambda^{\alpha}_{\mathbf{fix}+}$ compared to $\lambda^{\alpha}_{\mathbf{fix}}$ is given in Figure 4.1. Note the type annotations at **Left** and **Right** constructors. They state the type of the complementary alternative, e.g. $\mathbf{Left}_{()}\ 0$ has type $\mathit{Either}\ \mathit{Nat}\ ()$. $\lambda^{\alpha}_{\mathbf{fix}+}$

Concerning typing, we do not explicitly state the additional rules. The ones from Figures 2.2 and 2.7 remain unchanged and for the new term constructs it is straightforward to add rules. Moreover, the rules can be reconstructed by removing all Pointed restrictions from the refined typing rules (Figure 4.6) that we discuss in Section 4.2. Additional definitions of type and term semantics compared to $\lambda^{\alpha}_{\mathbf{fix}}$ are shown in Figures 4.2 and 4.3. The additional liftings of the logical relation are given in Figure 4.4.

$$
\begin{aligned}
[\![()]\!]^{\mathsf{fix}}_{\theta} &= \{()\}_{\bot} \\
[\![(\tau_1,\tau_2)]\!]^{\mathsf{fix}}_{\theta} &= \{(\mathbf{a},\mathbf{b}) \mid \mathbf{a} \in [\![\tau_1]\!]^{\mathsf{fix}}_{\theta}, \mathbf{b} \in [\![\tau_2]\!]^{\mathsf{fix}}_{\theta}\}_{\bot} \\
[\![\mathit{Either}\ \tau_1\ \tau_2]\!]^{\mathsf{fix}}_{\theta} &= (\{\mathrm{Left}\ \mathbf{a} \mid \mathbf{a} \in [\![\tau_1]\!]^{\mathsf{fix}}_{\theta}\} \cup \{\mathrm{Right}\ \mathbf{a} \mid \mathbf{a} \in [\![\tau_2]\!]^{\mathsf{fix}}_{\theta}\})_{\bot}
\end{aligned}
$$

Figure 4.2: Type semantics of $\lambda^{\alpha}_{\mathbf{fix}^+}$ and $\lambda^{\alpha}_{\mathbf{fix}^*}$, extended from Figure 2.8

$$
\begin{aligned}
[\![()]\!]^{\mathsf{fix}}_{\sigma} &= () \\
[\![(t_1,t_2)]\!]^{\mathsf{fix}}_{\sigma} &= ([\![t_1]\!]^{\mathsf{fix}}_{\sigma}, [\![t_2]\!]^{\mathsf{fix}}_{\sigma}) \\
[\![\mathbf{Left}_{\tau}\ t]\!]^{\mathsf{fix}}_{\sigma} &= \mathrm{Left}\ [\![t]\!]^{\mathsf{fix}}_{\sigma} \\
[\![\mathbf{Right}_{\tau}\ t]\!]^{\mathsf{fix}}_{\sigma} &= \mathrm{Right}\ [\![t]\!]^{\mathsf{fix}}_{\sigma}
\end{aligned}
$$

$$
[\![\mathbf{case}\ t\ \mathbf{of}\ \{\,() \to t_1\,\}]\!]^{\mathsf{fix}}_{\sigma} = \begin{cases} [\![t_1]\!]^{\mathsf{fix}}_{\sigma} & \text{if } [\![t]\!]^{\mathsf{fix}}_{\sigma} = () \\ \bot & \text{if } [\![t]\!]^{\mathsf{fix}}_{\sigma} = \bot \end{cases}
$$

$$
[\![\mathbf{case}\ t\ \mathbf{of}\ \{\,(x_1,x_2) \to t_1\,\}]\!]^{\mathsf{fix}}_{\sigma} = \begin{cases} [\![t_1]\!]^{\mathsf{fix}}_{\sigma[x_1\mapsto\mathbf{a},x_2\mapsto\mathbf{b}]} & \text{if } [\![t]\!]^{\mathsf{fix}}_{\sigma} = (\mathbf{a},\mathbf{b}) \\ \bot & \text{if } [\![t]\!]^{\mathsf{fix}}_{\sigma} = \bot \end{cases}
$$

$$
[\![\mathbf{case}\ t\ \mathbf{of}\ \{\,\mathbf{Left} \to t_1; \mathbf{Right} \to t_2\,\}]\!]^{\mathsf{fix}}_{\sigma} = \begin{cases} [\![t_1]\!]^{\mathsf{fix}}_{\sigma[x_1\mapsto\mathbf{a}]} & \text{if } [\![t]\!]^{\mathsf{fix}}_{\sigma} = \mathrm{Left}\ \mathbf{a} \\ [\![t_2]\!]^{\mathsf{fix}}_{\sigma[x_2\mapsto\mathbf{a}]} & \text{if } [\![t]\!]^{\mathsf{fix}}_{\sigma} = \mathrm{Right}\ \mathbf{a} \\ \bot & \text{if } [\![t]\!]^{\mathsf{fix}}_{\sigma} = \bot \end{cases}
$$

Figure 4.3: Term semantics of $\lambda^{\alpha}_{\mathbf{fix}^+}$ and $\lambda^{\alpha}_{\mathbf{fix}^*}$, extended from Figure 2.9

$$
\begin{aligned}
\Delta^{\mathbf{fix}}_{(),\rho} &= id_{(\{()\})_{\bot}} \\
\Delta^{\mathbf{fix}}_{(\tau_1,\tau_2),\rho} &= \{(\bot,\bot)\} \cup \{((\mathbf{a},\mathbf{b}),(\mathbf{c},\mathbf{d})) \mid (\mathbf{a},\mathbf{c}) \in \Delta^{\mathbf{fix}}_{\tau_1,\rho} \wedge (\mathbf{b},\mathbf{d}) \in \Delta^{\mathbf{fix}}_{\tau_2,\rho}\} \\
\Delta^{\mathbf{fix}}_{\mathit{Either}\ \tau_1\ \tau_2,\rho} &= \{(\bot,\bot)\} \cup \{(\mathrm{Left}\ \mathbf{x}, \mathrm{Left}\ \mathbf{y}) \mid (\mathbf{x},\mathbf{y}) \in \Delta^{\mathbf{fix}}_{\tau_1,\rho}\} \cup \{(\mathrm{Right}\ \mathbf{x}, \mathrm{Right}\ \mathbf{y}) \mid (\mathbf{x},\mathbf{y}) \in \Delta^{\mathbf{fix}}_{\tau_2,\rho}\}
\end{aligned}
$$

Figure 4.4: Logical relation for $\lambda^{\alpha}_{\mathbf{fix}^+}$ and $\lambda^{\alpha}_{\mathbf{fix}^*}$, extended from Figure 2.10

4.2 Refined Typing

The idea of Launchbury and Paterson (1996) is to track the occurrences of **fix** in a term via the type system. When we compare the semantics of λ^{α} (Section 2.1) and $\lambda^{\alpha}_{\mathbf{fix}}$ (Section 2.2), we observe that the introduction of **fix** causes significant semantic changes. In particular, we switch from sets to pcpos as semantic domains for the type interpretation. The alteration also influences the logical relation (cf. Figures 2.5 and 2.10) and consequently alters the parametricity theorem by forcing ρ to map to strict and continuous relations.

Closer examination of the proof of Theorem 2 reveals where the additional conditions to gain parametricity arise. Strictness of the logical relation is only forced in the proof cases for **case**-expressions, for (SUM) and for (FIX). Regarding (FIX), also continuity of the logical relation is employed. But besides in the just mentioned proof cases, no extra conditions compared to the proof of Theorem 1 are necessary.

Hence, since the proof cases resemble the structure of terms, the idea is to allow the logical relation to be nonstrict for (refined) types of which no problematic

$$\Gamma_{\mathcal{T}} \vdash () \in \mathsf{Pointed}\ (\text{CP-Brace}) \qquad \Gamma_{\mathcal{T}} \vdash \mathit{Nat} \in \mathsf{Pointed}\ (\text{CP-Nat}) \qquad \Gamma_{\mathcal{T}} \vdash [\tau] \in \mathsf{Pointed}\ (\text{CP-List})$$

$$\Gamma_{\mathcal{T}} \vdash (\tau_1, \tau_2) \in \mathsf{Pointed}\ (\text{CP-Pair}) \qquad \Gamma_{\mathcal{T}} \vdash \mathit{Either}\ \tau_1\ \tau_2 \in \mathsf{Pointed}\ (\text{CP-Either})$$

$$\frac{\alpha^* \in \Gamma_{\mathcal{T}}}{\Gamma_{\mathcal{T}} \vdash \alpha \in \mathsf{Pointed}}\ (\text{CP-Var}) \qquad \frac{\Gamma_{\mathcal{T}} \vdash \tau_2 \in \mathsf{Pointed}}{\Gamma_{\mathcal{T}} \vdash \tau_1 \to \tau_2 \in \mathsf{Pointed}}\ (\text{CP-Arrow})$$

Figure 4.5: Class membership rules for Pointed in $\lambda^{\alpha}_{\mathbf{fix}^*}$

terms are present and thus avoid strictness conditions when proving the parametricity theorem. We do not reconsider the continuity condition because the imposed restrictions are mostly unimportant in applications of free theorems[5].

Pointed

Types on which the logical relation does not have to be strict need not necessarily be interpreted as pcpos. In particular they need not to be pointed. It suffices if they are cpos. Referring to that difference, we set up a type class Pointed in the way Launchbury and Paterson (1996) do. The semantics of types in Pointed must be interpreted as pcpos and the relational interpretation must be strict. Types not in the type class may be interpreted as unpointed cpos and their relational interpretation can be nonstrict.

Convention 6 (pointed / unpointed)

Types in Pointed we call *pointed*, all other types we call *unpointed*.

Concerning type variables, we have to decide if they shall be pointed or not. If pointed, we allow them only to be interpreted by pcpos and to have a strict relational interpretation. To distinguish the two kinds of type variables, we annotate the pointed ones by * in the typing context, e.g. α, β^* means that β is forced to have a strict relational interpretation, but the standard type interpretation for α does not even have to be a pcpo and its relational interpretation can be an arbitrary continuous relation. For cpos D_1 and D_2, we denote the set of continuous relations between D_1 and D_2 by $Rel^{\infty}(D_1, D_2)$. Furthermore, we denote the collection of continuous relations over two arbitrary cpos by Rel^{∞}.

$Rel^{\infty}/Rel^{\infty}(D_1, D_2)$

If the decision for type variables is fixed, class membership in Pointed propagates through all types as given by the class membership rules in Figure 4.5. Note that the rules are dependent only on the type context $\Gamma_{\mathcal{T}}$ and not on the whole typing context Γ, and that all algebraic data types, and also Nat, are directly in Pointed without any precondition[6]. The only propagation of the choice for type variables takes place for functions via the rule (CP-Arrow). The rule corresponds to the fact that the relational interpretation of functions is only strict if the interpretation of the result type is, as visible from the function

[5]If we consider the semantics of functions of our λ-calculus as choice for the logical relation for type variables, their graphs are continuous relations anyway.

[6]It is not necessary that base types and algebraic data types are pointed by default. See the work of Launchbury and Paterson (1996) for their results about boxing and unboxing.

lifting of the logical relation (Figure 2.10), considering that the least element on function level is the constant function to $\perp$.

We prove that for all types the relational interpretation is continuous and that for pointed types this interpretation is additionally strict, i.e., that the type class restriction captures the underlying intuition.

Lemma 11 If ρ maps into Rel^{∞} then $\Delta^{\mathbf{fix}}_{\tau,\rho} \in Rel^{\infty}$ for all τ closed under $dom(\rho)$.

Proof. Induction on the structure of τ, employing the definition of the logical relation. □

Lemma 12 If $\Gamma_{\mathcal{T}} \vdash \tau \in \mathsf{Pointed}$ valid and

- for every α occurring in $\Gamma_{\mathcal{T}}$, $\rho(\alpha) \in Rel^{\infty}$ as well as
- for every α^{*} occurring in $\Gamma_{\mathcal{T}}$, $\rho(\alpha) \in Rel^{\perp}$,

then $\Delta^{\mathbf{fix}}_{\tau,\rho}$ strict and continuous.

Proof. Continuity is guaranteed by Lemma 11. Strictness is proved via induction on the depth of the derivation tree for $\Gamma_{\mathcal{T}} \vdash \tau \in \mathsf{Pointed}$, employing the definition of the logical relation. □

To allow for an enhanced parametricity theorem compared to the calculus with standard types, we adjust the typing rules such that the rules, where the proof cases for Theorem 2 enforce strictness conditions, enforce pointedness of types. We call the resulting calculus $\lambda^{\alpha}_{\mathbf{fix}^{*}}$. All typing rules of $\lambda^{\alpha}_{\mathbf{fix}^{*}}$ are given in Figure 4.6. Rules with a $\mathsf{Pointed}$ restriction as premise, i.e., the ones altered compared to Figures 2.2 and 2.7, are annotated with $'$ in their name. $\lambda^{\alpha}_{\mathbf{fix}^{*}}$

In the sense of typeability of terms the calculi $\lambda^{\alpha}_{\mathbf{fix}^{+}}$ and $\lambda^{\alpha}_{\mathbf{fix}^{*}}$ are equivalent.

Theorem 4 (Launchbury and Paterson (1996)) Whenever $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}^{+}}$, then $\Gamma^{*} \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}^{*}}$, where Γ^{*} is Γ with all type variables in $\Gamma_{\mathcal{T}}$ annotated by *.
Conversely, if $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}^{*}}$, then $\Gamma' \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}^{+}}$, where Γ' is Γ with all *-annotations at type variables removed.

Although, the same terms are typeable, the refined typing rules allow for a stronger parametricity theorem than the one we can prove in $\lambda^{\alpha}_{\mathbf{fix}^{+}}$. Strictness conditions on relations can partly be removed.

$$\Gamma, x :: \tau \vdash x :: \tau \ (\textsc{Var}) \qquad \Gamma \vdash [\,]_\tau :: [\tau] \ (\textsc{Nil}) \qquad \Gamma \vdash n :: \mathit{Nat} \ (\textsc{Nat}) \qquad \Gamma \vdash () :: () \ (\textsc{Unit})$$

$$\frac{\Gamma \vdash t_1 :: \mathit{Nat} \qquad \Gamma \vdash t_2 :: \mathit{Nat}}{\Gamma \vdash (t_1 + t_2) :: \mathit{Nat}} \ (\textsc{Sum}) \qquad \frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]} \ (\textsc{Cons})$$

$$\frac{\Gamma_\mathcal{T} \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1; x_1 : x_2 \to t_2\}) :: \tau} \ (\textsc{LCase}')$$

$$\frac{\Gamma_\mathcal{T} \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t :: () \qquad \Gamma \vdash t_1 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{() \to t_1\}) :: \tau} \ (\textsc{UCase}')$$

$$\frac{\Gamma_\mathcal{T} \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t :: \mathit{Nat} \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}) :: \tau} \ (\textsc{NCase}')$$

$$\frac{\Gamma_\mathcal{T} \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t :: \mathit{Either}\ \tau_1\ \tau_2 \qquad \Gamma, x :: \tau_1 \vdash t_1 :: \tau \qquad \Gamma, x :: \tau_2 \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{\mathbf{Left}\ x \to t_1; \mathbf{Right}\ x \to t_2\}) :: \tau} \ (\textsc{ECase}')$$

$$\frac{\Gamma_\mathcal{T} \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t :: (\tau_1, \tau_2) \qquad \Gamma, x_1 :: \tau_1, x_2 :: \tau_2 \vdash t_1 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{(x_1, x_2) \to t_1\}) :: \tau} \ (\textsc{PCase}')$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1 \to \tau_2} \ (\textsc{Abs}) \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2} \ (\textsc{App})$$

$$\frac{\Gamma_\mathcal{T} \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t :: \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ t :: \tau} \ (\textsc{Fix}') \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)} \ (\textsc{Pair})$$

$$\frac{\Gamma \vdash t :: \tau_1}{\Gamma \vdash \mathbf{Left}_{\tau_2}\ t :: \mathit{Either}\ \tau_1\ \tau_2} \ (\textsc{Left}) \qquad \frac{\Gamma \vdash t :: \tau_2}{\Gamma \vdash \mathbf{Right}_{\tau_1}\ t :: \mathit{Either}\ \tau_1\ \tau_2} \ (\textsc{Right})$$

Figure 4.6: Typing rules of $\lambda^{\alpha}_{\mathbf{fix}^*}$

Theorem 5 (Parametricity Theorem for $\lambda^{\alpha}_{\mathbf{fix}^*}$, Launchbury and Paterson (1996))

If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}^*}$, then for every θ_1, θ_2, ρ, σ_1, σ_2 such that

- for every α occurring in $\Gamma_\mathcal{T}$, $\rho(\alpha) \in \mathit{Rel}^{\infty}(\theta_1(\alpha), \theta_2(\alpha))$,
- for every α^* occurring in $\Gamma_\mathcal{T}$, $\rho(\alpha) \in \mathit{Rel}^{\perp}(\theta_1(\alpha), \theta_2(\alpha))$, and
- for every $x :: \tau'$ occurring in $\Gamma_\mathcal{V}$, $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathbf{fix}}_{\tau',\rho}$,

we have $([\![t]\!]_{\sigma_1}, [\![t]\!]_{\sigma_2}) \in \Delta^{\mathbf{fix}}_{\tau,\rho}$.

Proof. The proof is similar to the one of Theorem 2. Strictness conditions are guaranteed by pointedness of types whenever needed. □

Example 10 (type derivation in $\lambda^{\alpha}_{\mathbf{fix}^*}$)

Let us exemplify the new strength gained by refined typing. First, we reconsider the function from the introduction where the free theorem breaks if the strictness condition is dropped. The function is

$$f :: [\alpha] \to [\alpha] \text{ with } f = \lambda xs :: [\alpha].[\bot_\alpha]$$

When we type it by the rules from Figure 4.6, it is necessary to annotate α in Γ by *, i.e., to force the relational interpretation of α to be strict. The complete type derivation for f is as follows, where $\Gamma = \alpha^*, xs :: [\alpha]$.

$$\dfrac{\dfrac{\dfrac{\alpha^* \vdash \alpha \in \mathsf{Pointed} \qquad \dfrac{\Gamma, x :: \alpha \vdash x :: \alpha}{\Gamma \vdash (\lambda x :: \alpha.x) :: \alpha \to \alpha}}{\Gamma \vdash (\mathbf{fix}\ (\lambda x :: \alpha.x)) :: \alpha} \qquad \Gamma \vdash []_\alpha :: [\alpha]}{\Gamma \vdash (\bot_\alpha : []_\alpha) :: [\alpha]}}{\alpha^* \vdash (\lambda xs :: [\alpha].[\bot_\alpha]) :: [\alpha] \to [\alpha]}$$

Summarized, we get $\alpha^* \vdash f :: [\alpha] \to [\alpha]$ valid, but not $\alpha \vdash f :: [\alpha] \to [\alpha]$. In the just given example, we won nothing compared to the free theorem w.r.t. standard typing, because the strictness condition was really necessary.

In contrast, for the length function $l :: [\alpha] \to Nat$ with

$$l = \mathbf{fix}\ (\lambda l' :: [\alpha] \to Nat.\lambda xs :: [\alpha].\mathbf{case}\ xs\ \mathbf{of}\ \{[] \to 0; x : xs \to 1 + l'\ xs\})$$

we get the valid typing judgments $\alpha \vdash l :: [\alpha] \to Nat$ and $\alpha^* \vdash l :: [\alpha] \to Nat$. That is, we do not need to annotate α and hence can safely drop the strictness condition on $\rho(\alpha)$ concerning the parametricity theorem (Theorem 5) and consequently the strictness condition on the free theorem.

Contributing to our overall goal, refined typing helps to narrow down the search space for terms suitable for counterexamples. If we search for a function with standard type $f :: [\alpha] \to [\alpha]$, it can only give rise to a counterexample to the free theorem with dropped strictness restriction if in the refined system $\alpha^* \vdash f :: [\alpha] \to [\alpha]$ is valid[7], but not $\alpha \vdash f :: [\alpha] \to [\alpha]$.

The immediate idea, how to search for suitable functions f, is to use the refined typing rules backwards and focus on an application of (Fix′) to create a term of a type that is unpointed iff a certain type variable in Γ is not *-annotated.

Unfortunately, as already mentioned in the beginning of this chapter, the rule system in Figure 4.6 is not suitable for term construction. It lacks (at least a weak kind of) a subformula property, i.e., the property that whenever we use the typing rules backwards to search a way to construct a term, we can predict the rule applications necessary to complete the search. Hence, we have to find an appropriate replacement for the typing rules.

[7] By Theorem 4 this typing judgment must always be valid.

4.3 An Alternative System of Typing Rules

The problematic part of the typing rules in Figure 4.6 is the rule (App). Interpreted on the logical side (with terms and Γ omitted), the rule corresponds to modus ponens. As Dyckhoff (1992) shows for the intuitionistic propositional calculus, the rule can be replaced by several other rules to obtain a (w.r.t. the provable formulas) equivalent rule system that enjoys better algorithmic properties concerning proof search. In particular, the altered rule system features a weak form of the subformula property that ensures termination. Corbineau (2004) extends the results of Dyckhoff (1992) and achieves a rule system suitable for proof search in first-order intuitionistic logic. We are not interested in the first-order extension because we regard only the simply typed λ-calculus with type variables that corresponds to the propositional intuitionistic logic, but we are interested in his additional extension to inductive definitions. They correspond to algebraic data types in the λ-calculus.

The transliteration from Corbineau's rule system to the λ-calculus gives a system of typing rules that is, w.r.t. type inhabitation, equivalent to the original rule system of $\lambda^{\alpha}_{\mathbf{fix}^+}$ without **fix**. That the logic corresponds to a λ-calculus *without* general recursion is perspicuous when we consider the transliteration of (Fix) to the logical side. The rule corresponds to $(A \Rightarrow A) \Rightarrow A$ and therefore would make any logic inconsistent in the sense that we can derive every statement. Back on the λ-calculus side inconsistency corresponds to the fact that every type is inhabited when **fix** is allowed.

For our purpose of counterexample generation the directly translated term generator is not sufficient. Terms generated do never lead to counterexamples. In particular, they do not contain **fix** and hence cannot break the free theorem that lacks only the restrictions enforced by general recursion. Therefore, we must inject **fix** into the generated terms somehow. More precisely, we need to explore every possible way to inject **fix** such that for the constructed term an insufficiently restricted free theorem might break. The adjustment of the rules is the topic of the next section. In this section, we only discuss briefly the transliteration of the rules from Corbineau (2004). The transliterated rules are given in Figure 4.7.

Convention 7 **(abbreviations)**

In Figure 4.7 (and also later on) the following abbreviations are employed.

$$fst\ p = \mathbf{case}\ p\ \mathbf{of}\ \{(x,y) \to x\}$$
$$snd\ p = \mathbf{case}\ p\ \mathbf{of}\ \{(x,y) \to y\}$$
$$curry_{(\tau_1,\tau_2)}\ f = \lambda x :: \tau_1.\lambda y :: \tau_2.f\ (x,y)$$

Since we are only interested in one term of each type (that for the adjusted rules later on is a term that might break the unrestricted free theorem), we can restrict the search space for terms. It suffices if we are able to find one term

$$\Gamma, x :: \tau \vdash x :: \tau \; (\textsc{Var}) \qquad \Gamma \vdash []_{\tau} :: [\tau] \; (\textsc{Nil}) \qquad \Gamma \vdash 0 :: \mathit{Nat} \; (\textsc{Nat}) \qquad \Gamma \vdash () :: () \; (\textsc{Unit})$$

$$\frac{\Gamma \vdash t_1 :: \tau}{\Gamma, l :: [\tau] \vdash t_1 :: \tau} \; (\textsc{LDrop}) \qquad \frac{\Gamma \vdash t :: \tau}{\Gamma, x :: \mathit{Nat} \vdash t :: \tau} \; (\textsc{NDrop}) \qquad \frac{\Gamma \vdash t :: \tau}{\Gamma, x :: () \vdash t :: \tau} \; (\textsc{UDrop})$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1 \to \tau_2} \; (\textsc{Abs}) \qquad \frac{\Gamma, x :: \tau_1, y :: \tau_2 \vdash t_1 :: \tau}{\Gamma, x :: \tau_1, f :: \tau_1 \to \tau_2 \vdash t[f\ x / y] :: \tau} \; (\textsc{App}')$$

$$\frac{\Gamma, x :: \tau_2 \vdash t :: \tau}{\Gamma, f :: [\tau_1] \to \tau_2 \vdash t[f\ []_{\tau_1} / x] :: \tau} \; (\textsc{Wrap}{\to})$$

$$\frac{\Gamma, x :: \tau_1, g :: \tau_2 \to \tau_3 \vdash t_1 :: \tau_2 \qquad \Gamma, y :: \tau_3 \vdash t_2 :: \tau}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \vdash t_2[f\ (\lambda x :: \tau_1 . t_1[\lambda z :: \tau_2 . f\ (\lambda u :: \tau_1 . z) / g]) / y] :: \tau} \; (\textsc{Arrow}{\to})$$

$$\frac{\Gamma, y :: \tau_1 \vdash t :: \tau}{\Gamma, f :: \mathit{Nat} \to \tau_1 \vdash t[f\ 0 / y] :: \tau} \; (\textsc{NApp}) \qquad \frac{\Gamma, y :: \tau_1 \vdash t :: \tau}{\Gamma, f :: () \to \tau_1 \vdash t[f\ () / y] :: \tau} \; (\textsc{UApp})$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)} \; (\textsc{Pair}) \qquad \frac{\Gamma, x :: \tau_1, y :: \tau_2 \vdash t :: \tau}{\Gamma, p :: (\tau_1, \tau_2) \vdash t[\mathit{fst}\ p / x, \mathit{snd}\ p / y] :: \tau} \; (\textsc{Proj})$$

$$\frac{\Gamma, g :: \tau_1 \to \tau_2 \to \tau_3 \vdash t :: \tau}{\Gamma, f :: (\tau_1, \tau_2) \to \tau_3 \vdash t[\mathit{curry}_{(\tau_1, \tau_2)}\ f / g] :: \tau} \; (\textsc{Pair}{\to})$$

$$\frac{\Gamma \vdash t :: \tau_1}{\Gamma \vdash \mathbf{Left}_{\tau_2}\ t :: \mathit{Either}\ \tau_1\ \tau_2} \; (\textsc{Left}) \qquad \frac{\Gamma \vdash t :: \tau_2}{\Gamma \vdash \mathbf{Right}_{\tau_1}\ t :: \mathit{Either}\ \tau_1\ \tau_2} \; (\textsc{Right})$$

$$\frac{\Gamma, x :: \tau_1 \vdash t_1 :: \tau \qquad \Gamma, x :: \tau_2 \vdash t_2 :: \tau}{\Gamma, e :: \mathit{Either}\ \tau_1\ \tau_2 \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{Left}\ x \to t_1; \mathbf{Right}\ x \to t_2\} :: \tau} \; (\textsc{Dist})$$

$$\frac{\Gamma, g :: \tau_1 \to \tau_3, h :: \tau_2 \to \tau_3 \vdash t :: \tau}{\Gamma, f :: \mathit{Either}\ \tau_1\ \tau_2 \to \tau_3 \vdash t[\lambda x :: \tau_1 . f\ (\mathbf{Left}_{\tau_2}\ x) / g, \lambda y :: \tau_2 . f\ (\mathbf{Right}_{\tau_1}\ y) / h] :: \tau} \; (\textsc{Either}{\to})$$

Figure 4.7: Term search rule system

of a type if there exists a term, and if we do not lose this property when we adjust the rules in the next section. In particular, for a list we can always take the empty list, and each natural number we set to 0. Since we start out from the rules of Corbineau (2004) and only cut down some rules, the correctness proof of the rules from Corbineau (2004) carries over to our rule system. That is, whenever t is typeable to τ under Γ via the rules in Figure 4.7, then it is also typeable to τ under the same Γ via the original typing rules. Also, we claim the system presented in Figure 4.7 to be complete in the following sense.

For every type τ for which there exists a term t and context Γ with $\Gamma \vdash t :: \tau$ valid via the original typing rules of $\lambda^{\alpha}_{\mathbf{fix}^{+}}$, there exists t' with $\Gamma \vdash t' :: \tau$ valid via the rules in Figure 4.7.

Claim 1 (completeness)

Note that the just given notion of completeness does not imply that every term typeable under the original rule system is typeable under the altered one. This is not the case. But, whenever a type is inhabited w.r.t. the original rule system, it is inhabited w.r.t. the altered rules as well. Only if this claim is correct, the term construction described in the next section that leads to the algorithm $\mathrm{TermFind}$ is complete in the sense that whenever we might find a counterexample to a free theorem, then $\mathrm{TermFind}$ generates a term.

The application of the rule system in Figure 4.7 as a term construction algorithm, i.e., employment of the rules (with Γ and τ as input) backwards until a type derivation is found and the construction of a term t along the found derivation, forces backtracking. In particular, the rules (Left) and (Right) are in competition and if the first premise in the rule (Arrow→) fails, backtracking is also necessary (Corbineau, 2004). A discussion of a suitable (in the sense of an efficient algorithm) priority list for the rules when employing them for term construction is given by Corbineau (2004, Section 4). We omit a discussion here, but are geared to the priority list of Corbineau (2004) when prioritizing the rules of the adapted algorithm for counterexample search in the next section.

We close this section by an example on how terms of a given type are constructed via the just presented rules system.

Example 11

We consider the typing judgment

$$\alpha, \beta, \gamma \vdash t :: ((\alpha \to (\beta, Nat)) \to \gamma) \to (\alpha \to \beta) \to \gamma$$

The task is to construct t, such that the judgment is valid in $\lambda^{\alpha}_{\mathbf{fix}^+}$.

Guided by the term structure, we generate the following derivation tree for the typing judgment. The tree is constructed from the root to the leaves. For brevity, we omit the type variables in the typing context and summarize term variables in the context if they are not of interest for the construction. To do so, we set $\Gamma_1 = h :: (\beta, Nat) \to \gamma$, $\Gamma_2 = \Gamma_1, x :: \alpha$, $\Gamma_3 = \Gamma_2, y :: \beta$ and $\Gamma_4 = g :: (\alpha \to \beta)$. Furthermore, the rule name (A→) abbreviates (Arrow→).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2, y :: \beta \vdash t_5 :: \beta}\,(\textsc{Var}) \quad \dfrac{}{\Gamma_3 \vdash t_5' :: Nat}\,(\textsc{Nat})}{\Gamma_2, y :: \beta \vdash t_4 :: (\beta, Nat)}\,(\textsc{Pair})}{\Gamma_1, x :: \alpha, g :: (\alpha \to \beta) \vdash t_3 :: (\beta, Nat)}\,(\textsc{App}') \quad \dfrac{}{\Gamma_4, z :: \gamma \vdash t_3' :: \gamma}\,(\textsc{Var})}{f :: ((\alpha \to (\beta, Nat)) \to \gamma), g :: (\alpha \to \beta) \vdash t_2 :: \gamma}\,(\textsc{A}\!\to)}{f :: ((\alpha \to (\beta, Nat)) \to \gamma) \vdash t_1 :: (\alpha \to \beta) \to \gamma}\,(\textsc{Abs})}{\emptyset \vdash t :: ((\alpha \to (\beta, Nat)) \to \gamma) \to (\alpha \to \beta) \to \gamma}\,(\textsc{Abs})$$

The terms $t, t_1 \ldots t_5$, etc. are unknown while building the tree. We construct them starting at the leaves of the tree, following the typing rules' instructions on how to construct the terms in the conclusion with the help of the premises.

Hence we get:

$$t_5 = y \qquad t_5' = 0 \qquad t_4 = (y, 0) \qquad t_3 = (g\ x, 0) \qquad t_3' = z$$
$$t_2 = f\ (\lambda x :: \alpha.(g\ x, 0)) \qquad t_1 = \lambda g :: \alpha \to \beta.f\ (\lambda x :: \alpha.(g\ x, 0))$$

And finally we obtain, the term t we searched for:

$$t = \lambda f :: (\alpha \to (\beta, Nat)) \to \gamma.\lambda g :: \alpha \to \beta.f\ (\lambda x :: \alpha.(g\ x, 0))$$

4.4 Terms that Give Rise to Counterexamples

As already mentioned, to enable counterexample search the typing rule system presented in Section 4.3 (Figure 4.7) has to be adjusted. In particular, we need a rule to introduce **fix** at places where it enforces an additional strictness condition on the free theorem under investigation to make it correct. We call such a use of **fix** *harmful*. Regarding our insights from Section 4.2 we can reduce the search space for terms with a harmful application of **fix** significantly. Say, in $\lambda^{\alpha}_{\mathbf{fix}^+}$, we have a typing context Γ not containing the type variable α and a type τ. We search for a term t with $\alpha, \Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}^+}$ that breaks the corresponding instance of the parametricity theorem (and the free theorem) if the relation (or function in the specialized free theorem) $\rho(\alpha)$ is not required to be strict. By Theorem 5, which incorporates the knowledge of refined typing, we know that for t the typing judgment $\alpha^*, \Gamma \vdash t :: \tau$ must, but the typing judgment $\alpha, \Gamma \vdash t :: \tau$ must not be valid in $\lambda^{\alpha}_{\mathbf{fix}^*}$.

harmful

4.4.1 Term Generation via TermFind: Strategy and Definition

How can we, in an algorithmic way, find terms with $\alpha^*, \Gamma \vdash t :: \tau$, but not $\alpha, \Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fix}^*}$? Regarding the typing rules of $\lambda^{\alpha}_{\mathbf{fix}^*}$ presented in Figure 4.6 the most direct way is to introduce **fix** t' of type τ where τ is pointed if *and only if* α is annotated by * in the typing context. Hence, our main idea for the term find algorithm is represented by the following rule where $\bot_\tau$, as already defined, is the term **fix** id_τ.

$$\frac{\Gamma_T \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash \bot_\tau :: \tau}\ (\textsc{Bottom})$$

The property $\Gamma_T \vdash \tau \notin \mathsf{Pointed}$ is defined via the class membership rules for Pointed shown in Figure 4.5. It is fulfilled iff $\Gamma_T \vdash \tau \in \mathsf{Pointed}$ is not derivable.

$\Gamma_T \vdash \tau \notin \mathsf{Pointed}$

Note that we exchanged $\vdash$, employed in the typing judgments up to now, by $\Vdash$ for a typing judgment made by the algorithm we are going to design. That way, we distinguish the typing judgments made by different systems of typing rules. Later on, we introduce also the symbols $\Vdash^\circ$ and $\Vdash'$, but do not comment on them any further.

$\Vdash, \Vdash^\circ, \Vdash'$

Let us, regarding rule (Bottom), explain input and output of the term find algorithm we are going to design, in the following called TermFind. We search

TermFind

for terms of a given type, and in general for such terms that are closed under a given typing context. Initially, the typing context might be a type context containing the type variables occurring in the given type. But during the term search also term variables with type annotation can be present. Thus, a typing context Γ and a type τ are the input to $\mathrm{TermFind}$. A pair of such a type and typing context is called *external input* to $\mathrm{TermFind}$. Internally $\mathrm{TermFind}$ adds annotations to term variables in typing contexts, thus we distinguish between external and internal input. Since we utilize the refined typing in the style of Section 4.2, by typing context and type we refer to the respective definitions in the calculus $\lambda^{\alpha}_{\mathbf{fix}^*}$.

Definition 17 (external input)

A tuple $\mathcal{I}^{ext} = (\Gamma; \tau)$ with Γ a typing context as in $\lambda^{\alpha}_{\mathbf{fix}^*}$ and τ a type closed under Γ is called *external input*.

Γ^*

As output we construct, if possible, a term t typeable to τ *not* under the given typing context Γ but under Γ^*, i.e., Γ with all type variables *-annotated.[8] Hence $\mathrm{TermFind}$ — if successful — returns a term t with $\Gamma \vdash t :: \tau$ invalid in $\lambda^{\alpha}_{\mathbf{fix}^*}$, but $\Gamma^* \vdash t :: \tau$ valid. For example, having successfully applied (Bottom) for a given type τ and typing context Γ as input, $\mathrm{TermFind}$ will return the term $\bot_\tau$ as output.

Note that the use of the refined type system allows fine-grained analysis of strictness conditions. If we annotate a type variable in the typing context of the input by *, we take for granted that the relational interpretation for the variable will be strict, as reflected in the membership rules for type class Pointed. Only for type variables without a *-annotation, it is tested if relaxing the strictness condition on the relational interpretation of these type variables might lead to a counterexample for the, then insufficiently restricted, parametricity theorem.

Alike the rule (Bottom) that arises as alteration of the rule (Fix′) from Figure 4.6, we could alter other rules from Figure 4.6 that enforce pointedness conditions on a type. But it is unnecessary to regard alterations of these rules because for every type τ and context Γ with τ unpointed under Γ we can immediately apply (Bottom) to obtain a term t such that $\Gamma \vdash t :: \tau$ is not derivable, i.e., valid, in $\lambda^{\alpha}_{\mathbf{fix}^*}$, but $\Gamma^* \vdash t :: \tau$ is.

locally harmful

Unfortunately, the introduction of $\bot_{\tau'}$ on an unpointed type τ', which we call *locally harmful* use of **fix**, is not always sufficient to generate the intended counterexample for an external input $(\Gamma; \tau)$. Locally, for type τ' and the respective context, the parametricity theorem breaks but it is not guaranteed that employing (Bottom) during term search on a type τ' implies that we generate a counterexample for the type and context given as input. The impossibility to always propagate a local breach of the parametricity theorem to a breach of the parametricity theorem on the overall type implies that not every term t

[8]In the remainder of this Chapter for a given typing context Γ, the typing context Γ^* always denotes Γ with all type variables *-annotated.

typeable to τ under Γ^*, but not under Γ, is suitable for a counterexample. Each term with a locally harmful use of **fix** cannot be typed under Γ in $\lambda^{\alpha}_{\mathbf{fix}^*}$, even if it does not give rise to a counterexample to the parametricity theorem with missing strictness conditions at all. To illustrate how such a term looks like, here are two examples.

EXAMPLE 12

The terms

$$(\lambda x :: \alpha.0)\ \bot_\alpha \quad \text{and} \quad \mathbf{case}\ [\bot_\alpha]\ \mathbf{of}\ \{[\,] \to 0; x : xs \to 0\}$$

are of type Nat under typing context $\Gamma = \alpha^*$ and not typeable under $\Gamma = \alpha$ in $\lambda^{\alpha}_{\mathbf{fix}^*}$. Nevertheless, both are semantically equivalent to the very benign term 0 and hence will never enforce a strictness condition for $\rho(\alpha)$ in the statement of the parametricity theorem and thus on the free theorem. Consequently, both terms never give rise to a counterexample, regardless of the locally harmful $\bot_\alpha$.

We can describe the strategy for term search in TermFind by two aims:

- Introduce $\bot_\tau$ whenever τ is unpointed, to provoke a local disrelation[9], i.e., a breach concerning the parametricity theorem.
- Ensure that the provoked local disrelation is propagated to the final term.

Of course, an additional main goal when designing the algorithm is to guarantee termination and efficiency. To obtain these algorithmic properties, we start out from the rules of the always terminating term construction algorithm presented in Section 4.3 and alter them w.r.t. the just described strategy where the primary way to introduce a local disrelation, i.e., a locally harmful **fix**, is the extra rule (BOTTOM). The strategy-guided alteration of the rule system from Section 4.3 yields the algorithm TermFind that has several phases. The rules for the different phases are shown in Figures 4.8, 4.9 and 4.10. As we prove later on, even with full backtracking the algorithm terminates. Nevertheless, for efficiency reasons we propose an order of application on the rules that reduces backtracking. The order is inspired by the one given by Corbineau (2004, Section 4) for (the more general version of) the algorithm from Section 4.3. It is reflected in the way the rules are arranged in the figures. Rules further up have higher precedence. As visible from the figures, backtracking is reduced by the proposed order of rules.

CONVENTION 8 (abbreviations)

Presenting the rules of TermFind, we employ the following abbreviations.

$$\begin{aligned} head_\tau\ l &= \mathbf{case}\ l\ \mathbf{of}\ \{[\,] \to \bot_\tau; x{:}\,\_ \to x\} \\ fromLeft_\tau\ e &= \mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{Left}\ x \to x; \mathbf{Right}\ x \to \bot_\tau\} \\ fromRight_\tau\ e &= \mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{Left}\ x \to \bot_\tau; \mathbf{Right}\ x \to x\} \end{aligned}$$

[9]By disrelation we mean that there exist related environments under which the semantics of the term under consideration is not related to itself.

$$\frac{\Gamma_T \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash \bot_\tau :: \tau}\ (\textsc{Bottom}) \qquad \frac{\Gamma_T \vdash \tau_1 \notin \mathsf{Pointed} \qquad \Gamma \Vdash t :: \tau}{\Gamma, x :: \tau_1 \Vdash t :: \tau}\ (\textsc{UPDrop})$$

$$\frac{\Gamma \Vdash t :: \tau}{\Gamma, x :: \mathit{Nat} \Vdash t :: \tau}\ (\textsc{NDrop}) \qquad \frac{\Gamma \Vdash t :: \tau}{\Gamma, x :: () \Vdash t :: \tau}\ (\textsc{UDrop})$$

$$\frac{\Gamma, x :: \tau_1 \Vdash t :: \tau_2}{\Gamma \Vdash (\lambda x :: \tau_1 . t) :: \tau_1 \to \tau_2}\ (\textsc{Abs}) \qquad \frac{\Gamma_T \vdash \tau_2 \in \mathsf{Pointed} \qquad \Gamma, g :: \tau_1 \to \tau_2 \Vdash t :: \tau}{\Gamma, f :: [\tau_1] \to \tau_2 \Vdash t[\lambda y :: \tau_1 . f\ (y : [\,]_{\tau_1})/g] :: \tau}\ (\textsc{Wrap}{\to}')$$

$$\frac{\Gamma, h :: \tau_1 \Vdash t :: \tau}{\Gamma, l :: [\tau_1] \Vdash t[(\mathit{head}_{\tau_1}\ l)/h] :: \tau}\ (\textsc{Head})$$

$$\frac{\Gamma_T \vdash \tau_3 \in \mathsf{Pointed} \qquad \Gamma, g :: \tau_1 \to \tau_2 \to \tau_3 \Vdash t :: \tau}{\Gamma, f :: (\tau_1, \tau_2) \to \tau_3 \Vdash t[\mathit{curry}_{(\tau_1,\tau_2)}\ f/g] :: \tau}\ (\textsc{Pair}{\to})$$

$$\frac{\Gamma, x :: \tau_1, y :: \tau_2 \Vdash t :: \tau}{\Gamma, p :: (\tau_1, \tau_2) \Vdash t[\mathit{fst}\ p/x, \mathit{snd}\ p/y] :: \tau}\ (\textsc{Proj})$$

$$\frac{\Gamma_T \vdash \tau_3 \in \mathsf{Pointed} \qquad \Gamma, g :: \tau_1 \to \tau_3, h :: \tau_2 \to \tau_3 \Vdash t :: \tau}{\Gamma, f :: \mathit{Either}\ \tau_1\ \tau_2 \to \tau_3 \Vdash t[\lambda x :: \tau_1 . f\ (\mathbf{Left}_{\tau_2}\ x)/g, \lambda y :: \tau_2 . f\ (\mathbf{Right}_{\tau_1}\ y)/h] :: \tau}\ (\textsc{Either}{\to})$$

backtracking only necessary below

$$\frac{\Gamma, x :: \tau_1 \Vdash t :: \tau}{\Gamma, e :: \mathit{Either}\ \tau_1\ \tau_2 \Vdash t[\mathit{fromLeft}_{\tau_2}\ e/x] :: \tau}\ (\textsc{Dist}_1)$$

$$\frac{\Gamma, x :: \tau_2 \Vdash t :: \tau}{\Gamma, e :: \mathit{Either}\ \tau_1\ \tau_2 \Vdash t[\mathit{fromRight}_{\tau_1}\ e/x] :: \tau}\ (\textsc{Dist}_2)$$

$$\frac{\Gamma_T \vdash \tau_1 \notin \mathsf{Pointed} \qquad \Gamma, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma, f :: \tau_1 \to \tau_2 \Vdash t[f\ \bot_{\tau_1}/y] :: \tau}\ (\textsc{Bottom}{\to}')$$

$$\frac{\Gamma_T \vdash \tau_2, \tau_3 \in \mathsf{Pointed} \quad \Gamma, w :: \tau_1, g :: \tau_2 \to \tau_3 \Vdash t_1 :: \tau_2 \quad \Gamma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \Vdash t_2[f\ (\lambda x :: \tau_1 . t_1[\lambda z :: \tau_2 . f\ (\lambda u :: \tau_1 . z)/g, x/w])/y] :: \tau}\ (\textsc{Arrow}{\to}')$$

$$\frac{\Gamma \Vdash t :: \tau}{\Gamma \Vdash t : [\,]_\tau :: [\tau]}\ (\textsc{Wrap}) \qquad \frac{\Gamma \Vdash t :: \tau_1}{\Gamma \Vdash (t, \bot_{\tau_2}) :: (\tau_1, \tau_2)}\ (\textsc{Pair}_1) \qquad \frac{\Gamma \Vdash t :: \tau_2}{\Gamma \Vdash (\bot_{\tau_1}, t) :: (\tau_1, \tau_2)}\ (\textsc{Pair}_2)$$

$$\frac{\Gamma \Vdash t :: \tau_1}{\Gamma \Vdash \mathbf{Left}_{\tau_2}\ t :: \mathit{Either}\ \tau_1\ \tau_2}\ (\textsc{Left}) \qquad \frac{\Gamma \Vdash t :: \tau_2}{\Gamma \Vdash \mathbf{Right}_{\tau_1}\ t :: \mathit{Either}\ \tau_1\ \tau_2}\ (\textsc{Right})$$

$$\frac{\Gamma, y :: \tau_2 \Vdash t :: \tau}{\Gamma, f :: \tau_1 \to \tau_2 \Vdash t[f\ \bot_{\tau_1}/y] :: \tau}\ (\textsc{Bottom}{\to})$$

Figure 4.8: Phase I rules of TermFind

$$\Gamma, x^\circ :: \tau \Vdash^\circ x :: \tau \ (\textsc{Var}^\circ)$$

$$\frac{\Gamma, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma, f^\circ :: \tau_1 \to \tau_2 \Vdash^\circ t[f\ \bot_{\tau_1}/y] :: \tau}\ (\textsc{Bottom}{\to}^\circ)$$

$$\frac{\Gamma_T \vdash \tau_2 \in \mathsf{Pointed} \qquad \Gamma, x^\circ :: \tau_1, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma, x^\circ :: \tau_1, f :: \tau_1 \to \tau_2 \Vdash^\circ t[f\ x/y] :: \tau}\ (\textsc{App}'^\circ)$$

backtracking only necessary below

$$\frac{\Gamma_V \Vdash' t :: \tau}{\Gamma, x^\circ :: \mathit{Nat} \Vdash^\circ \mathbf{case}\ x\ \mathbf{of}\ \{0 \to t\} :: \tau}\ (\textsc{Nat}^\circ)$$

$$\frac{\Gamma_V \Vdash' t :: \tau}{\Gamma, x^\circ :: () \Vdash^\circ \mathbf{case}\ x\ \mathbf{of}\ \{() \to t\} :: \tau}\ (\textsc{Unit}^\circ)$$

$$\frac{\Gamma_V \Vdash' t :: \tau}{\Gamma, l^\circ :: [\tau_1] \Vdash^\circ \mathbf{case}\ l\ \mathbf{of}\ \{[x] \to t\} :: \tau}\ (\textsc{List}^\circ)$$

$$\frac{\Gamma_V \Vdash' t :: \tau}{\Gamma, p^\circ :: (\tau_1, \tau_2) \Vdash^\circ \mathbf{case}\ p\ \mathbf{of}\ \{(x, y) \to t\} :: \tau}\ (\textsc{Pair}^\circ)$$

$$\frac{\Gamma_V \Vdash' t :: \tau}{\Gamma, e^\circ :: \mathit{Either}\ \tau_1\ \tau_2 \Vdash^\circ \mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{Left}\ x \to t\} :: \tau}\ (\textsc{Either}^\circ)$$

$$\frac{\Gamma, h^\circ :: \tau_1 \Vdash^\circ t :: \tau}{\Gamma, l^\circ :: [\tau_1] \Vdash^\circ t[\mathit{head}_{\tau_1}\ l/h] :: \tau}\ (\textsc{Head}^\circ)$$

$$\frac{\Gamma, x^\circ :: \tau_1, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma, p^\circ :: (\tau_1, \tau_2) \Vdash^\circ t[\mathit{fst}\ p/x, \mathit{snd}\ p/y] :: \tau}\ (\textsc{Proj}^\circ)$$

$$\frac{\Gamma, x^\circ :: \tau_1 \Vdash^\circ t :: \tau}{\Gamma, e^\circ :: \mathit{Either}\ \tau_1\ \tau_2 \Vdash^\circ t[\mathit{fromLeft}_{\tau_2}\ e/x] :: \tau}\ (\textsc{Dist}_1^\circ)$$

$$\frac{\Gamma, x^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma, e^\circ :: \mathit{Either}\ \tau_1\ \tau_2 \Vdash^\circ t[\mathit{fromRight}_{\tau_1}\ e/x] :: \tau}\ (\textsc{Dist}_2^\circ)$$

Figure 4.9: Phase II rules of TermFind

$$\Gamma_V, x :: \tau \Vdash' x :: \tau\ (\textsc{Var}') \qquad \Gamma_V \Vdash' 0 :: \mathit{Nat}\ (\textsc{Nat}') \qquad \Gamma_V \Vdash' () :: ()\ (\textsc{Unit}') \qquad \Gamma_V \Vdash' [\bot_\tau] :: [\tau]\ (\textsc{List}')$$

$$\Gamma_V \Vdash' (\bot_{\tau_1}, \bot_{\tau_2}) :: (\tau_1, \tau_2)\ (\textsc{Pair}') \qquad \Gamma_V \Vdash' \mathbf{Left}_{\tau_2}\ \bot_{\tau_1} :: \mathit{Either}\ \tau_1\ \tau_2\ (\textsc{Either}')$$

Figure 4.10: Phase III rules of TermFind

All three are used on the semantic level later on, then written without type annotations. Additionally, in case expressions we omit branches that yield the undefined value ($\bot_\tau$). In particular we write

$$\begin{array}{l}\textbf{case}\ l\ \textbf{of}\ \{[x] \to t\}\\ \textbf{case}\ e\ \textbf{of}\ \{\textbf{Left}\ x \to t\}\\ \textbf{case}\ e\ \textbf{of}\ \{\textbf{Right}\ x \to t\}\\ \textbf{case}\ i\ \textbf{of}\ \{0 \to t\}\end{array}$$

where the first shortened case expression yields a defined value only for singleton lists.

In (Arrow→') we write $\Gamma_\tau \vdash \tau_2, \tau_3 \in \mathsf{Pointed}$ instead of two separate premises stating τ_2 pointed and τ_3 pointed, respectively. The abbreviations are also employed later on, in other places than the rules of $\mathrm{TermFind}$.

4.4.2 Detailed Explanations on the Design of $\mathrm{TermFind}$

Before we describe the alterations to the rules from Section 4.3, let us clarify guidelines for the transformation that arise from the term search strategy just described.

- Firstly, when $\mathrm{TermFind}$ terminates, we never return a term where no $\bot_\tau$ with τ unpointed is injected, i.e., where not a single, at least locally, harmful **fix** is injected. Returning such a term would immediately render our algorithm incorrect in the sense that the refined type system of $\lambda^{\alpha}_{\mathbf{fix}^*}$ would tell that the strictness condition we want to prove necessary is superfluous regarding the free theorem for the concrete term we constructed.
- Secondly, we ensure not to miss any opportunity to insert $\bot_\tau$ for unpointed τ when we can propagate the thus enforced local disrelation.
- Thirdly, we try to keep the algorithm efficient. In particular, besides choosing a reasonable order of rules, we avoid unnecessary construction steps and unnecessarily big contexts. To avoid unnecessary construction steps, we freely employ $\bot_\tau$ also on pointed types whenever we want to cut a search branch. For example, if we search for a pair we simply insert $\bot_\tau$ of appropriate type in the second component if we injected a harmful **fix** already in the first component. To gain small typing contexts, we decide to remove variables in the context that do not provide any additional possibilities to find a term. That handling is already present in the rule system presented in Section 4.3 via the rules that drop variables from the context, i.e., (NDrop), (UDrop) and (LDrop) (cf. Figure 4.7).

Having given the guidelines for the transformation of the **fix**-free term generator from Section 4.3 into a generator that yields terms that are probably[10]

[10]In fact, not all terms we construct will be suitable to generate counterexamples. But more on this later (Section 4.5.6, in particular Example 21).

suitable for counterexample generation, we describe the transformation in detail. First of all, as already mentioned, we are interested in inserting $\bot_\tau$ for τ unpointed. The primary way to do so is the rule (BOTTOM), hence it is also the first rule we employ in $\mathrm{TermFind}$. The remaining rules arise as adaptations of the rules from Figure 4.7. We discuss how the individual rules of the **fix**-free term generator shown in Figure 4.7 translate into the rules of $\mathrm{TermFind}$. The axioms (VAR), (NIL), (UNIT) and (NAT) cannot be applied directly in $\mathrm{TermFind}$ because the constructed terms do not involve **fix** at all and thus will not serve as part of a counterexample. Hence, all four axioms are not part of (at least the first phase of) $\mathrm{TermFind}$. But, are there adapted versions that we can employ in $\mathrm{TermFind}$? Indeed, an adaptation of the rule (NIL) will help to find counterexamples. The general term search algorithm in Section 4.3 introduces empty lists whenever possible, which is reasonable because it searches only for some arbitrary term. Rethinking that design in the light of our new task of counterexample generation, empty lists are an unsatisfactory choice. We lose possibilities to insert terms of the list element type into the overall term and hence may miss the possibility to inject $\bot_\tau$ for τ unpointed. The solution is a switch to non-empty lists, in particular to singleton lists. We replace (NIL) by

$$\frac{\Gamma \Vdash t :: \tau}{\Gamma \Vdash t : [\,]_\tau :: [\tau]} \;(\textsc{Wrap})$$

The rule reduces the search for a term of list type to the search for a term of the list's element type. The choice of singleton lists appears reasonable because for lists with more elements we only duplicate possibilities to insert a harmful **fix**. In contrast to (NIL), the rules (VAR), (UNIT) and (NAT) cannot be changed similarly. There is no possibility to inject a harmful **fix** via a variable, via $()$ or via a natural number.

The drop rules (NDROP) and (UDROP) from the **fix**-free term generator, we take over unchanged. We can directly place constants instead of the variables if term generation requires a natural number or a term of unit type, so variables of such types are superfluous in the context. Additionally, whenever we encounter a variable of unpointed type in the typing context, we can remove it safely, which is expressed by the new rule (UPDROP) in Figure 4.8. Since we are interested in the introduction of $\bot_\tau$ for τ unpointed, we always prefer injecting the undefined value instead of the variable from the typing context into our term. Care has to be taken concerning the rule (LDROP). By a switch from empty to nonempty lists we can pull out an element of the list that might be useful for counterexample search. Hence, we replace (LDROP) by

$$\frac{\Gamma, h :: \tau_1 \Vdash t :: \tau}{\Gamma, l :: [\tau_1] \Vdash t[(head_{\tau_1}\; l)/h] :: \tau} \;(\textsc{Head})$$

The rule (ABS) is taken over directly. If we cannot insert a harmful $\bot_\tau$ for the type $\tau = \tau_1 \to \tau_2$ via (BOTTOM), we decompose the function type, which on the one hand allows us to further decompose τ_2 if possible, or on the other hand to employ the newly gained context information, i.e., that a variable of type τ_1 is in the typing context.

The rule (APP′) gives rise to two different rules. We ease the conditions of the rule and do not require a variable $x :: \tau_1$ as argument for $f :: \tau_1 \to \tau_2$ in the typing context. We simply employ $\bot_{\tau_1}$ as argument. That handling directly yields the rule

$$\frac{\Gamma, y :: \tau_2 \Vdash t :: \tau}{\Gamma, f :: \tau_1 \to \tau_2 \Vdash t[f\ \bot_{\tau_1}/y] :: \tau}\ (\textsc{Bottom}{\to})$$

If we know that τ_1 is unpointed then the rule already injects a harmful **fix**. That is, for this case it suffices to guarantee $f\ \bot_{\tau_1}$ is really part of the finally generated term. Hence, it suffices to ensure the presence of y in t in the premise of (BOTTOM→). This task is handled by a second rule system (phase II) of TermFind. The system is presented in Figure 4.9. We discuss the rules of phase II later on. The rule to enter phase II in the just described case is

$$\frac{\Gamma_{\mathcal{T}} \vdash \tau_1 \notin \mathsf{Pointed} \qquad \Gamma, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma, f :: \tau_1 \to \tau_2 \Vdash t[f\ \bot_{\tau_1}/y] :: \tau}\ (\textsc{Bottom}{\to}')$$

where the $^\circ$-annotation at the term variable y expresses that y must be used somewhere in t. Of course, we prefer (BOTTOM→′) to (BOTTOM→), because it is the more direct way to a term possibly yielding a counterexample.

In the light of rule (BOTTOM→), the rules (NAPP) and (UAPP) from the original term generator are superfluous. Their conclusions are special cases of the conclusion of (BOTTOM→) and the premises are identical.[11]

For the rule (WRAP→) the situation looks the same as for (NAPP) and (UAPP), but it is different. We chose to switch from empty lists as default to singleton lists. The switch alters rule (WRAP→) to

$$\frac{\Gamma_{\mathcal{T}} \vdash \tau_2 \in \mathsf{Pointed} \qquad \Gamma, g :: \tau_1 \to \tau_2 \Vdash t :: \tau}{\Gamma, f :: [\tau_1] \to \tau_2 \Vdash t[\lambda y :: \tau_1 . f\ (y : [\,]_{\tau_1})/g] :: \tau}\ (\textsc{Wrap}{\to}')$$

and (WRAP→′) has a slightly more general premise than (WRAP→). Thus, it is not subsumed by (BOTTOM→). The generalization consists of the function $g :: \tau_1 \to \tau_2$ in the premise instead of a value $y :: \tau_2$. The pointedness restriction is not a limitation at all because term variables of unpointed type, as already explained when we described (UPDROP), will not enable us to find more suitable terms for counterexample generation, and if τ_2 is unpointed, so is $\tau_1 \to \tau_2$ and hence f is useless for our term search. We add pointedness restrictions only where necessary later on for counterexample generation via the algorithm ExFind, described in Section 4.5. There, some constructions for term environment entries require pointedness. We could add even more pointedness restrictions, e.g. at (DIST$_1$) and (DIST$_2$).

Because of the more general premise, also the rules (ARROW→), (PAIR→) and (EITHER→) are not subsumed by (BOTTOM→). The (ARROW→) rule requires

[11] Keep in mind that we use the rules backwards to search terms, i.e., the roles of premise and conclusion are switched and thus (BOTTOM→) really subsumes the two other rules.

a slight change when translated to the algorithm for counterexample search. The original version does not guarantee that the constructed term contains a harmful **fix**, even if t_1 in the first premise of (ARROW→) does. The term t_1 is only used in t_2 if y is present in t_2 in the second premise of (ARROW→). Consequently, to guarantee the use of y in t_2 we alter the second premise of (ARROW→) to a call to phase II of TermFind. The resulting rule (ARROW→') is

$$\frac{\Gamma_\tau \vdash \tau_2, \tau_3 \in \mathsf{Pointed} \quad \Gamma, w :: \tau_1, g :: \tau_2 \to \tau_3 \Vdash t_1 :: \tau_2 \quad \Gamma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \Vdash t_2[f\ (\lambda x :: \tau_1 . t_1[\lambda z :: \tau_2 . f\ (\lambda u :: \tau_1 . z)/g, x/w])/y] :: \tau}$$

where we additionally enforce two pointedness restrictions. Let us recheck that we did not weaken the rule in a way such that we lose terms. First, consider there is a term t_2 that does not introduce the harmful **fix** via the substitution of y, but still is suitable for a counterexample, i.e., a term where the second premise of (ARROW→) introduces the harmful **fix**. That case is covered by the rule (BOTTOM→). Second, consider the pointedness conditions we added. If τ_2 is unpointed, so is $\tau_1 \to \tau_2$ (cf. class membership rules for Pointed, Figure 4.5) and hence the case is covered by rule (BOTTOM→'). Pointedness of τ_3 we already explained to be not a real restriction when elaborating on rule (UPDROP).

The rules (PAIR→) and (EITHER→) carry over unchanged, except for the pointedness checks. But these we know (see the discussion for (UPDROP)) not to prevent us from finding suitable terms.

The remaining rules to consider are (PAIR), (PROJ), (LEFT), (RIGHT) and (DIST). The rules (PROJ), (LEFT) and (RIGHT) are taken over unchanged. The rules (PAIR) and (DIST) can be enhanced in terms of more relaxed premises. For (PAIR) we can split the rule into

$$\frac{\Gamma \Vdash t :: \tau_1}{\Gamma \Vdash (t, \bot_{\tau_2}) :: (\tau_1, \tau_2)}\ (\text{PAIR}_1) \qquad \frac{\Gamma \Vdash t :: \tau_2}{\Gamma \Vdash (\bot_{\tau_1}, t) :: (\tau_1, \tau_2)}\ (\text{PAIR}_2)$$

Each rule requires exactly one of the two premises of (PAIR). The idea is to search for a harmful **fix** only in one component of a pair, because this is sufficient for a counterexample. Consequently, we set the other component to $\bot_\tau$ of appropriate type τ and thus abandon the search for a place to insert a harmful **fix** in that component. By similar considerations, we split the rule (DIST) into

$$\frac{\Gamma, x :: \tau_1 \Vdash t :: \tau}{\Gamma, e :: \mathit{Either}\ \tau_1\ \tau_2 \Vdash t[\mathit{fromLeft}_{\tau_2}\ e/x] :: \tau}\ (\text{DIST}_1)$$

$$\frac{\Gamma, y :: \tau_2 \Vdash t :: \tau}{\Gamma, e :: \mathit{Either}\ \tau_1\ \tau_2 \Vdash t[\mathit{fromRight}_{\tau_1}\ e/y] :: \tau}\ (\text{DIST}_2)$$

Again, both rules have only one premise of the original rule (DIST). This way, we ensure not to enforce multiple insertions of harmful **fix** to generate a counterexample.

Having completed the description of the first phase of $\mathrm{TermFind}$ we move on to explain phase II. As already stated, the essential task of phase II is to ensure a relevant use of a $^\circ$-annotated term variable in the term that is constructed during that phase, i.e., a use such that the value of the variable really influences the semantics of the whole term.

EXAMPLE 13 (relevant use of a variable)

Consider the following terms, typeable to Nat under the context $x :: Nat$:

$$\textbf{case } x \textbf{ of } \{0 \to 3\} \quad \text{and} \quad \textbf{case } x \textbf{ of } \{0 \to \bot_{Nat}\}$$

In the first term the use of x is relevant, in the second it is not.

In principle, phase II could be another translation of the rules from Figure 4.7 in Section 4.3, but driven by a different strategy. Fortunately, we can reduce the effort. We can avoid all rules that decompose the type of the resulting term. All decomposition does already take place in the first phase. Moreover, it suffices to enter phase II with the type of the resulting term not a function type. Additionally, it is not necessary to manipulate term variables in the typing context that are not annotated by $^\circ$. These manipulations can take place in the first phase as well. Hence, the rules of phase II reduce to rules that either manipulate $^\circ$-annotated variables in the typing context or inject such variables in the term that we generate.

The most direct way phase II can succeed is to return a $^\circ$-annotated variable directly as produced term. Of course, immediate success is only possible if a respective $^\circ$-annotated variable of the result type exists, which is expressed via the rule

$$\Gamma, x^\circ :: \tau \Vdash^\circ x :: \tau \ (\text{VAR}^\circ)$$

If we cannot succeed directly, we manipulate the $^\circ$-annotated variables in the same way we did for unannotated variables in phase I. That is, we take over the rules (BOTTOM→), (HEAD), (PROJ), (DIST$_1$) and (DIST$_2$), altering them to work with $^\circ$-annotated variables. The adjusted rules are named like the original ones, up to an additional $^\circ$-symbol. We do not take over rules (PAIR→), (EITHER→) and alike. Though their adjusted versions would produce new $^\circ$-annotated variables, they are subsumed by the rule (BOTTOM→$^\circ$), the adjustment of (BOTTOM→). The reason is as follows: The only benefit the just mentioned rules have compared to (BOTTOM→) in phase I is that they produce new typing context entries of function type, where (BOTTOM→) would produce directly entries of the result type of these new entries. Since the only way to inject $^\circ$-annotated variables of function type into the term constructed by phase II is to apply them to some argument, the rule (BOTTOM→$^\circ$) subsumes all, on the first sight reasonable, translations of (PAIR→) etc. [12]

[12] Keep in mind that the type of the term constructed in phase II is never a function type, hence also rule (VAR$^\circ$) never fires for x° of function type.

It seems the rules of phase II just described are already sufficient for an exhaustive search, but we missed one option to insert $^\circ$-annotated variables into the term constructed by phase II. Like the strategy of "injecting a harmful **fix** whenever possible" in phase I required a switch from empty lists to singleton lists when adjusting the rules from the original term generator from Section 4.3, the "insert a $^\circ$-annotated variable whenever possible"-strategy of phase II requires the consideration of a broader range of terms. In particular, we need to consider case expressions, as apparent already from Example 13, page 78. We can inject a $^\circ$-annotated variable as the scrutinee of a case expression whenever the variable has the respective type to do so. Therefore, we enrich phase II with the rules (NAT$^\circ$), (UNIT$^\circ$), (LIST$^\circ$), (PAIR$^\circ$), (EITHER$^\circ$) that try to introduce the respective case expressions.

The case expressions that we introduce must fulfill one condition, as also apparent from Example 13: At least one alternative of the case statement must be semantically different from $\perp$. Otherwise the whole case expression is, independent of the scrutinee, semantically equivalent to the $\perp$ and hence the use of a variable as scrutinee is not relevant. Thus, to successfully create a whole case expression, we must find a term that is semantically different from $\perp$ and returned for an alternative of the case expression. To find an at least partly defined term, we employ phase III of $\mathrm{TermFind}$. As τ is never a function type (since we are in phase II), we concentrate on finding terms for all other types. This is done by the rules of phase III, shown in Figure 4.10. We simply employ a constructor of the particular type we search for to generate a term of at least partially defined value (for at least one possible semantic interpretation of type variables), or employ a term variable present in the context. The rule (VAR′) is necessary if for example τ is a type variable, which might happen as we allow *-annotated type variables and hence have type variables of pointed type.

Now that we described the idea behind the rules of $\mathrm{TermFind}$ completely, let us regard an example.

EXAMPLE 14

Consider the type $((Nat \to [\alpha]) \to Either\ Nat\ \beta) \to [Nat]$. We are only interested in the strictness condition arising from the type variable α. Thus we feed the external input $(\alpha, \beta^*; ((Nat \to [\alpha]) \to Either\ Nat\ \beta) \to [Nat])$ to $\mathrm{TermFind}$.

$\mathrm{TermFind}$ generates

$$\begin{aligned} t = \lambda f :: (Nat \to [\alpha]) \to Either\ Nat\ \beta.& \\ \textbf{case}\ f\ (\lambda x :: Nat.[\perp_\alpha])\ \textbf{of}\ \{\textbf{Left}\ y \to [\perp_{Nat}]\}& \end{aligned}$$

It can be derived as follows[13] where we abbreviate $(Nat \to [\alpha])$ by τ_1, $Either\ Nat\ \beta$ by τ_2 and $\textbf{case}\ f\ (\lambda x :: Nat.[\perp_\alpha])\ \textbf{of}\ \{\textbf{Left}\ y \to [\perp_{Nat}]\}$ by t_1.

$$\dfrac{\dfrac{(1) \qquad (2) \qquad (3)}{\alpha, \beta^*, f :: \tau_1 \to \tau_2 \Vdash t_1 :: [\mathit{Nat}]}\ (\textsc{Arrow}{\to}')}{\alpha, \beta^* \Vdash \lambda f :: \tau_1 \to \tau_2 . t_1 :: (\tau_1 \to \tau_2) \to [\mathit{Nat}]}\ (\textsc{Abs})$$

For (1) we have to show $\alpha, \beta^* \vdash [\alpha], \tau_2 \in \mathsf{Pointed}$, which is directly by the rules ($\textsc{CP-List}$) and ($\textsc{CP-Either}$). The premise (2) is a placeholder for the derivation

$$\dfrac{\dfrac{\alpha, \beta^* \vdash \alpha \notin \mathsf{Pointed}}{\alpha, \beta^*, w :: \mathit{Nat}, g :: [\alpha] \to \tau_2 \Vdash \bot_\alpha :: \alpha}\ (\textsc{Bottom})}{\alpha, \beta^*, w :: \mathit{Nat}, g :: [\alpha] \to \tau_2 \Vdash [\bot_\alpha] :: [\alpha]}\ (\textsc{Wrap})$$

where pointedness of α is not derivable and hence the unpointedness condition holds. Premise (3) is a placeholder for

$$\dfrac{\dfrac{}{z^\circ :: \tau_2 \Vdash' [\bot_{\mathit{Nat}}] :: [\mathit{Nat}]}\ (\textsc{List}')}{\alpha, \beta^*, z^\circ :: \tau_2 \Vdash^\circ \textbf{case}\ z\ \textbf{of}\ \{\textbf{Left}\ y \to [\bot_{\mathit{Nat}}]\} :: [\mathit{Nat}]}\ (\textsc{Either}^\circ)$$

The reader may ask, if, and if then how, our algorithm avoids to create false positives in the style of Example 12, page 71. The first false positive is ruled out because we never generate a term that consists of a function application, except we apply a term variable of function type. The term variable can always be interpreted as a non-constant function, thus the applications we allow are really different from the one in Example 12. Note also that function applications where the function is not a variable cannot provide for additional terms that are suitable for counterexample generation because we can directly apply β-conversion and obtain a semantically equivalent term without function application. False positives as the second one in Example 12 are not created in general, because we completely omit rules that wrap terms into a constructor to use them as a scrutinee of a case expression. The restriction does not limit the ability to find terms suitable for counterexample generation. Assume, we could inject a harmful **fix** or a $^\circ$-annotated term variable in a scrutinee of a case expression, but not directly in one of its branches. Then, in one of the alternatives of the case expression, the **fix** or the $^\circ$-annotated variable must be employed to force its evaluation. But if so, we can directly omit the case expression and only construct the one important alternative.

4.4.3 Properties of $\mathrm{TermFind}$

We came from a term generator that created a term for every inhabited type and rewrote that generator to a generator for terms that are likely to be suitable for counterexample generation. We applied an “introduce and propagate a harmful **fix** whenever possible”-strategy to modify the generator. Based on

[13]We omitted redundant rule applications that would arise if we strictly follow the rule’s precedence.

this strategy, we claim that $\mathrm{TermFind}$ is complete in the sense that there is no counterexample to the free theorem for given input $(\Gamma; \tau)$, i.e., for the free theorem of type τ with missing strictness conditions for type variables without *-annotation in Γ, if $\mathrm{TermFind}$ returns without a term.

Claim 2 (completeness)

If $\mathrm{TermFind}$ cannot construct a term for some external input, then the tested strictness conditions can safely be dropped from the parametricity theorem.

We do not prove completeness, but we prove a correctness result in the sense that we find terms where at least $\lambda^{\alpha}_{\mathbf{fix}^*}$ would enforce strictness conditions on the parametricity theorem. Moreover, we prove termination of $\mathrm{TermFind}$. But first, we remark on a detail of $\mathrm{TermFind}$: Phase I operates on another kind of typing contexts than phase II, in particular the term variables in the context given to phase II can be $^\circ$-annotated. Hence, additionally to the already defined external input, we specify an internal input.

Definition 18 (internal input)

A tuple $\mathcal{I}^{int} = (\Gamma; \tau)$, where compared to an external input term variables in Γ_V can be $^\circ$-annotated, is called *internal input*.

Note that every external input is also an internal input.

By the following theorem we state correctness of $\mathrm{TermFind}$ in the sense of typeability of the results in $\lambda^{\alpha}_{\mathbf{fix}^*}$.

Theorem 6 (correctness)

If $\mathrm{TermFind}$ returns a term t for the external input $(\Gamma; \tau)$ then t is typeable to τ under Γ^* in $\lambda^{\alpha}_{\mathbf{fix}^*}$, but not under Γ.

Proof. The proof splits into two parts. First we show that $\Gamma \vdash t :: \tau$ is not valid in $\lambda^{\alpha}_{\mathbf{fix}^*}$. Therefor, we prove that every term t returned by $\mathrm{TermFind}$ contains a subterm $\perp_{\tau'}$ with $\Gamma_T \vdash \tau' \notin \mathsf{Pointed}$. If this is the case, we have immediately that $\Gamma \vdash t :: \tau$ is not valid in $\lambda^{\alpha}_{\mathbf{fix}^*}$. In $\lambda^{\alpha}_{\mathbf{fix}^*}$ we must employ (Fix) to introduce $\perp_{\tau'}$, and its use is only allowed if τ' is pointed. We show that whenever $\mathrm{TermFind}$ terminates successfully, it employs (Bottom) or (Bottom$\rightarrow$') that both introduce $\perp_{\tau'}$ for τ' unpointed. Once introduced in the term generated by $\mathrm{TermFind}$, $\perp_{\tau'}$ is propagated by each rule of the first phase of the algorithm.

The second part of the proof is to show that $\Gamma^* \vdash t :: \tau$ is valid in $\lambda^{\alpha}_{\mathbf{fix}^*}$. We relate every rule of $\mathrm{TermFind}$ to a rule or a rule sequence of the typing rules of $\lambda^{\alpha}_{\mathbf{fix}^*}$ that performs the identical term transformation under the assumption that all types are pointed, which is immediate because each type variable in the context is *-annotated at each step of the derivation.

A more detailed version of the proof is found in Appendix A.1. □

Example 15

Example 14 shows that for input $(\alpha, \beta^*; ((Nat \to [\alpha]) \to Either\ Nat\ \beta) \to [Nat])$ $\mathrm{TermFind}$ produces the term

$$\begin{aligned} t = \lambda f :: (Nat \to [\alpha]) \to Either\ Nat\ \beta.& \\ &\mathbf{case}\ f\ (\lambda x :: Nat.[\bot_\alpha])\ \mathbf{of}\ \{\mathbf{Left}\ y \to [\bot_{Nat}]\} \end{aligned}$$

To exemplify the statement of Theorem 6, we (partly) show the type derivation for $\alpha^*, \beta^* \vdash t :: ((Nat \to [\alpha]) \to Either\ Nat\ \beta) \to [Nat]$ in $\lambda^{\alpha}_{\mathbf{fix}^*}$. The respective rules are found in Figure 4.6. We also discuss why a derivation of the typing judgment under context α, β^* is impossible. As in Example 14, we abbreviate $(Nat \to [\alpha])$ by τ_1, $Either\ Nat\ \beta$ by τ_2 and $\mathbf{case}\ g\ (\lambda x :: Nat.[\bot_\alpha])\ \mathbf{of}\ \{\mathbf{Left}\ y \to [\bot_{Nat}]\}$ by t_1.

The derivation is as follows:

$$\dfrac{\dfrac{(1) \qquad (2) \qquad (3) \qquad (4)}{\alpha^*, \beta^*, f :: \tau_1 \to \tau_2 \vdash t_1 :: [Nat]}\ (\textsc{ECase}')}{\alpha^*, \beta^* \vdash t :: (\tau_1 \to \tau_2) \to [Nat]}\ (\textsc{Abs})$$

We substitute (1) by $\alpha^*, \beta^* \vdash [Nat] \in \mathsf{Pointed}$, which is derivable directly by $(\textsc{CP-List})$, and (2) by the derivation

$$\dfrac{(2.1) \qquad \dfrac{\dfrac{(2.2) \qquad \dfrac{}{\alpha^*, \beta^*, f :: \tau_1 \to \tau_2, x :: Nat \vdash [\,]_\alpha :: [\alpha]}\ (\textsc{Nil})}{\alpha^*, \beta^*, f :: \tau_1 \to \tau_2, x :: Nat \vdash [\bot_\alpha] :: [\alpha]}\ (\textsc{Cons})}{\alpha^*, \beta^*, f :: \tau_1 \to \tau_2 \vdash \lambda x :: Nat.[\bot_\alpha] :: \tau_1}\ (\textsc{Abs})}{\alpha^*, \beta^*, f :: \tau_1 \to \tau_2 \vdash f\ (\lambda x :: Nat.[\bot_\alpha]) :: Either\ Nat\ \beta}\ (\textsc{App})$$

where premise (2.1) is

$$\dfrac{}{\alpha^*, \beta^*, f :: \tau_1 \to \tau_2 \vdash f :: \tau_1 \to \tau_2}\ (\textsc{Var})$$

and premise (2.2), with $f :: \tau_1 \to \tau_2, x :: Nat$ omitted in the typing context, is

$$\dfrac{\dfrac{\alpha^* \in \alpha^*, \beta^*}{\alpha^*, \beta^* \vdash \alpha \in \mathsf{Pointed}}\ (\textsc{CP-Var}) \qquad \dfrac{\dfrac{}{\alpha^*, \beta^*, z :: \alpha \vdash z :: \alpha}\ (\textsc{Var})}{\alpha^*, \beta^* \vdash \lambda z :: \alpha.z :: \alpha \to \alpha}\ (\textsc{Abs})}{\alpha^*, \beta^* \vdash \mathbf{fix}\ (\lambda z :: \alpha.z) :: \alpha}\ (\textsc{Fix}')$$

We omit the rather uninteresting derivations (3) and (4). The most important point in the derivation is the application of $(\textsc{CP-Var})$ in derivation (2.2). It directly relies on α^* being an element of the type context. Hence, under the context we fed to $\mathrm{TermFind}$, i.e., the context with α not *-annotated, the term found by $\mathrm{TermFind}$ is not typeable in $\lambda^{\alpha}_{\mathbf{fix}^*}$.

Note, that the above correctness result does not imply correctness in the sense that every term found by $\mathrm{TermFind}$ is suitable to create a counterexample to a free theorem with a missing strictness condition. Even though we avoid the generation of false positives alike the ones presented in Example 12, there are other difficulties when creating whole counterexamples. We provide detailed discussion on the generation of whole counterexamples in the next section.

From Theorem 6 and Theorem 4 it follows immediately that the terms found by $\mathrm{TermFind}$ are typeable in $\lambda^{\alpha}_{\mathbf{fix}+}$, the calculus without refined typing.

COROLLARY 1

If $\mathrm{TermFind}$ returns a term t for the external input $(\Gamma;\tau)$ then t is typeable to τ under Γ' in $\lambda^{\alpha}_{\mathbf{fix}+}$, where Γ' is Γ with all *-annotations removed.

Concerning termination, we prove that $\mathrm{TermFind}$ always terminates for every possible external input and even if used with an arbitrary rule order and full backtracking.

THEOREM 7 (termination)

$\mathrm{TermFind}$ terminates for every external input.

Proof. To state the termination of $\mathrm{TermFind}$ we introduce a termination order that decreases with every (backward) rule application during the construction of the derivation tree and thus reaches its least element after finitely many rule applications.

A detailed proof is given in Appendix A.1. □

Finally, with $\mathrm{TermFind}$ we gained an algorithm that we claim to be complete in the sense of finding a term whenever a counterexample to an unsatisfactorily restricted free theorem w.r.t. the given input type and context exists, and that is also proved correct in the sense of Theorem 6. Furthermore, $\mathrm{TermFind}$ terminates for every input. But, it does not create whole counterexamples, as was our original goal. Reconsider the free theorem for type $[\alpha] \to [\alpha]$ from the introduction. It states that for every function $f :: [\alpha] \to [\alpha]$, all types τ_1, τ_2, functions $g :: \tau_1 \to \tau_2$ and lists $xs :: [\tau_1]$, we have $f\ (map\ g\ xs) \equiv map\ g\ (f\ xs)$. We proved the additional restriction that g must be strict by choosing $f = \lambda x :: [\alpha].[\bot_\alpha]$, and appropriate τ_1, τ_2, g and xs. $\mathrm{TermFind}$ will yield at most a suitable function f. Hence, we have to extend the algorithm. Before we do so in the next section, let us round up the description of $\mathrm{TermFind}$ showing how $\mathrm{TermFind}$ actually works for the small example from the introduction. We schematically show a run to calculate a function $f :: [\alpha] \to [\alpha]$ that enforces strictness of the just mentioned function g, i.e., strictness of the relation assigned to α in the parametricity theorem. Therefor, we start $\mathrm{TermFind}$ with input $(\alpha, [\alpha] \to [\alpha])$. The complete run is shown in Figure 4.11.

$$\begin{tikzcd}
\alpha \vdash \alpha \notin \mathsf{Pointed} \arrow[r] & {} \arrow[d] \\
\alpha, xs :: [\alpha] \Vdash t_2 :: \alpha \arrow[u, "(\textsc{Bottom})"'] & t_2 = \bot_\alpha \arrow[d] \\
\alpha, xs :: [\alpha] \Vdash t_1 :: [\alpha] \arrow[u, "(\textsc{Wrap})"'] & t_1 = [\bot_\alpha] \arrow[d] \\
\alpha \Vdash t_0 :: [\alpha] \to [\alpha] \arrow[u, "(\textsc{Abs})"'] & t_0 = \lambda x :: [\alpha].[\bot_\alpha]
\end{tikzcd}$$

Figure 4.11: An example run for TermFind

As we see, TermFind generates exactly the function f we chose manually in the introduction. Instead of just giving a type derivation as in Example 14, Figure 4.11 highlights schematically how TermFind works: It first searches for a derivation tree for a term of the given type under the given context, and then constructs the output term along the found derivation tree. That is, all rules are first applied backwards for term search and then forwards for term construction.

4.5 Generation of Complete Counterexamples

As discussed at the end of the last section, TermFind does not produce complete counterexamples to free theorems that lack a strictness condition. In this section we present an alteration of TermFind, called ExFind. It produces complete counterexamples to such insufficiently restricted free theorems.

ExFind

Let us examine the task in detail, again exemplified regarding the type $[\alpha] \to [\alpha]$ and the context $\Gamma = \alpha$. As free theorems arise via exploration of the parametricity theorem, the generated counterexample must also invalidate the parametricity theorem. In $\lambda^{\alpha}_{\mathbf{fix}^+}$ the theorem (Theorem 2) states that for every function f with $\alpha \vdash f :: [\alpha] \to [\alpha]$, it holds that $([\![f]\!]^{\mathsf{fix}}_{\emptyset}, [\![f]\!]^{\mathsf{fix}}_{\emptyset}) \in \Delta^{\mathbf{fix}}_{[\alpha]\to[\alpha],[\alpha\mapsto\mathcal{R}]}$ whenever D_1, D_2 are pcpos and $\mathcal{R} \in \mathit{Rel}^{\bot}(D_1, D_2)$. We want to show that the theorem does not hold if we relax the condition on $\mathcal{R}$ to $\mathcal{R} \in \mathit{Rel}^{\infty}(D_1, D_2)$, i.e., allow nonstrict relations. Therefor, we can employ the function f that TermFind creates, but furthermore we need instances for D_1, D_2 and a concrete, of course nonstrict, relation $\mathcal{R} \in \mathit{Rel}^{\infty}(D_1, D_2)$ such that $([\![f]\!]^{\mathsf{fix}}_{\emptyset}, [\![f]\!]^{\mathsf{fix}}_{\emptyset}) \notin \Delta^{\mathbf{fix}}_{[\alpha]\to[\alpha],[\alpha\mapsto\mathcal{R}]}$. Concerning the specialization of relations to functions and of pcpos to the semantics of types, as we employ it for functional free theorems, we search for types τ_1 and τ_2 and a (nonstrict) function $g :: \tau_1 \to \tau_2$, such that $([\![f]\!]^{\mathsf{fix}}_{\emptyset}], [\![f]\!]^{\mathsf{fix}}_{\emptyset}]) \notin \Delta^{\mathbf{fix}}_{[\alpha]\to[\alpha],[\alpha\mapsto[\![g]\!]^{\mathsf{fix}}_{\emptyset}]}$.[14]

If we find suitable τ_1, τ_2 and g, we gain a counterexample to the parametricity

[14]We regard $[\![g]\!]^{\mathsf{fix}}_{\emptyset}$ as the function's graph.

theorem, but it is not obvious *why* exactly it is a counterexample. In particular, we cannot read off why the two semantic interpretations of f are not related by the logical relation.

For type $[\alpha] \to [\alpha]$ and the term $f = \lambda x :: [\alpha].[\bot_\alpha]$ it is not hard to find τ_1, τ_2 and $g :: \tau_1 \to \tau_2$, such that $([\![f]\!]^{\mathbf{fix}}_\emptyset, [\![f]\!]^{\mathbf{fix}}_\emptyset) \notin \Delta^{\mathbf{fix}}_{[\alpha]\to[\alpha],[\alpha\mapsto[\![g]\!]^{\mathbf{fix}}_\emptyset]}$. We might choose $\tau_1 = \tau_2 = ()$ and $g = \lambda x :: ().()$. Then, for $[\bot_{()}]$ as input, the free theorem states $map\ g\ (f\ [\bot_{()}]) \equiv f\ (map\ g\ [\bot_{()}])$ which is equal to the obviously wrong assertion $[()] \equiv [\bot_{()}]$.

For more complicated types it becomes less obvious to decide whether the parametricity theorem holds without additional strictness conditions or not.

EXAMPLE 16

Consider Example 14 again. For $(\alpha, \beta^*; ((Nat \to [\alpha]) \to Either\ Nat\ \beta) \to [Nat])$ as input $\mathrm{TermFind}$ constructs the term

$$\begin{aligned} t = \lambda f &:: (Nat \to [\alpha]) \to Either\ Nat\ \beta. \\ &\mathbf{case}\ f\ (\lambda x :: Nat.[\bot_\alpha])\ \mathbf{of}\ \{\mathbf{Left}\ y \to [\bot_{Nat}]\} \end{aligned}$$

Employing t we want to show that in the parametricity theorem for the above type the strictness condition on the relational interpretation of α is necessary. In particular, we want to prove by a counterexample that Theorem 5, the parametricity theorem for $\lambda^\alpha_{\mathbf{fix}^*}$ does not hold for $\alpha, \beta^* \vdash t :: ((Nat \to [\alpha]) \to Either\ Nat\ \beta) \to [Nat]$. $\mathrm{TermFind}$ does not guarantee that t is suitable for a counterexample to this theorem. It only states that we cannot disqualify t as a candidate by refined typing.

Hence, we are not done when t is found. We need to choose cpos D_1^α, D_2^α, pcpos D_1^β, D_2^β, and relations $\mathcal{R}^\alpha \in Rel^\infty(D_1^\alpha, D_2^\alpha)$ and $\mathcal{R}^\beta \in Rel^\bot(D_1^\beta, D_2^\beta)$, such that

$$([\![t]\!]^{\mathbf{fix}}_\emptyset, [\![t]\!]^{\mathbf{fix}}_\emptyset) \notin \Delta^{\mathbf{fix}}_{((Nat\to[\alpha])\to Either\ Nat\ \beta)\to[Nat],[\alpha\mapsto\mathcal{R}^\alpha,\beta\mapsto\mathcal{R}^\beta]}$$

Since, in the end, we aim for a counterexample to the functional free theorem that is based on the parametricity theorem, we actually search for τ_1, τ_2, τ_1', τ_2', $g :: \tau_1 \to \tau_2$ and $h :: \tau_1' \to \tau_2'$ such that

$$([\![t]\!]^{\mathbf{fix}}_\emptyset, [\![t]\!]^{\mathbf{fix}}_\emptyset) \notin \Delta^{\mathbf{fix}}_{((Nat\to[\alpha])\to Either\ Nat\ \beta)\to[Nat],[\alpha\mapsto[\![g]\!]^{\mathbf{fix}}_\emptyset,\beta\mapsto[\![h]\!]^{\mathbf{fix}}_\emptyset]}$$

One suitable choice is $\tau_1 = \tau_2 = \tau_1' = \tau_2' = ()$, $g = \lambda x :: ().()$ and $h = id_{()}$.

But, even knowing the concrete choices above, it is hard to see that we state a counterexample to the parametricity theorem. Furthermore, we created not yet a complete (or more precisely completely instantiated) counterexample to a free theorem: We need to unfold the logical relation and find out *why* the parametricity theorem breaks. Regarding our example, unfolding requires to find even more concrete terms. We search $(p, q) \in \Delta^{\mathbf{fix}}_{(Nat \to [\alpha]) \to Either\ Nat\ \beta, [\alpha \mapsto [\![g]\!]^{\mathbf{fix}}_{\emptyset}, \beta \mapsto [\![h]\!]^{\mathbf{fix}}_{\emptyset}]}$ such that

$$([\![t\ p]\!]^{\mathbf{fix}}_{\emptyset}, [\![t\ q]\!]^{\mathbf{fix}}_{\emptyset}) \notin \Delta^{\mathbf{fix}}_{[Nat], [\alpha \mapsto [\![g]\!]^{\mathbf{fix}}_{\emptyset}, \beta \mapsto [\![h]\!]^{\mathbf{fix}}_{\emptyset}]}$$

or, written simpler, we search for $p :: (Nat \to [()]) \to Either\ Nat\ ()$ and $q :: (Nat \to [()]) \to Either\ Nat\ ()$, related as just specified, such that $t\ p \not\equiv t\ q$.

Possible choices, retaining the type and function instantiations from above, are:

$$p = \lambda x :: Nat \to [()].\mathbf{case}\ x\ 0\ \mathbf{of}\ \{[y] \to \mathbf{Left}\ 0\}$$
$$q = \lambda x :: Nat \to [()].\mathbf{case}\ x\ 0\ \mathbf{of}\ \{[y] \to \mathbf{case}\ y\ \mathbf{of}\ \{() \to \mathbf{Left}\ 0\}\}$$

We get $t\ p \equiv [\bot_{Nat}] \not\equiv \bot_{[Nat]} \equiv t\ q$.

In this section we describe how to automatically gain all required instantiations and terms for a complete counterexample to a free theorem with missing strictness condition. We define the algorithm ExFind. Its output for this example is shown in Figure 4.12.

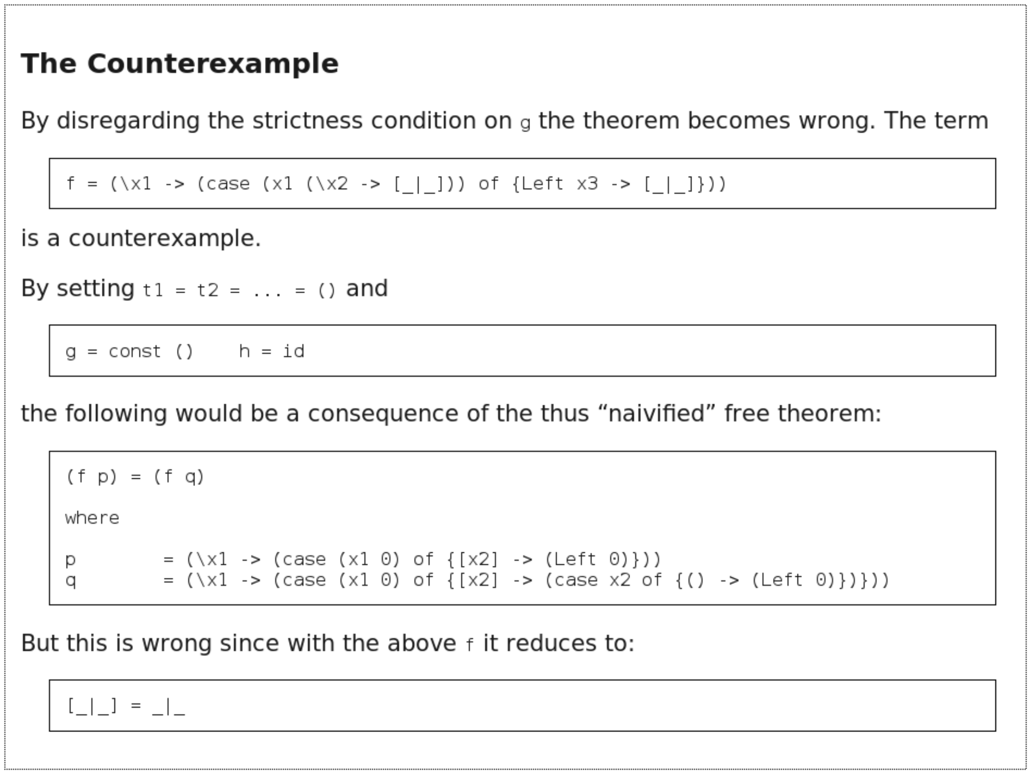
Example 16 demonstrates that additionally to the term found by TermFind extra information is necessary to create whole counterexamples to free theorems. Part of the information is *why* the free theorem breaks. To gain this information we must consider the liftings of the logical relation (cf. Figures 2.10 and 4.4):

- For functions, we need to find arguments related by the logical relation that yield unrelated results.
- For unrelated lists, we need to know if they have different size or if there are unrelated elements.
- For pairs, we need to know which component is unrelated.
- ...

Tracing back the origin why the interpretations of f are unrelated serves two goals. On the one hand it yields the last ingredient for a counterexample to a free theorem, for example for type $[\alpha] \to [\alpha]$ the argument to be given to f. On the other hand it provides even more information, allowing us to understand in detail why the counterexample really is a counterexample, i.e., why the interpretations of the found term are not related by the logical relation. Additionally, as we see later on, internally in ExFind the information has even another purpose.

Summarized, regarding the parametricity theorem (Theorem 2) and the above discussion, the algorithm ExFind, compared to TermFind, has to fulfill the

[14]The web interface is available at http://www-ps.iai.uni-bonn.de/cgi-bin/exfind.cgi



The Counterexample

By disregarding the strictness condition on `g` the theorem becomes wrong. The term

```
f = (\x1 -> (case (x1 (\x2 -> [_|_])) of {Left x3 -> [_|_]}))
```

is a counterexample.

By setting `t1 = t2 = ... = ()` and

```
g = const ()    h = id
```

the following would be a consequence of the thus "naivified" free theorem:

```
(f p) = (f q)

where

p        = (\x1 -> (case (x1 0) of {[x2] -> (Left 0)}))
q        = (\x1 -> (case (x1 0) of {[x2] -> (case x2 of {() -> (Left 0)})}))
```

But this is wrong since with the above `f` it reduces to:

```
[_|_] = _|_
```

Figure 4.12: ExFind's output for Example 16 as presented by the web interface[14]

following additional tasks:

- provide concrete type and relation environments θ_1, θ_2 and ρ,
- construct term environments σ_1 and σ_2,
- trace back the original breaking point of the parametricity theorem.

We design ExFind as an adaptation of TermFind such that in every step of term construction it provides a complete counterexample to the parametricity theorem (and also to the free theorem) for the respective typing judgment at the current step of TermFind.[15] Thus, we regard ExFind as an algorithm that searches for a suitable term for a counterexample by applying the rules of TermFind backwards, and then builds up not only the term defined by the derivation tree of TermFind, but additionally gathers all the other information necessary for a counterexample when applying the rules forwards. Hence, we enrich the rules of TermFind with additional constructions, in particular for term environments and tracing information. The type environments and the relation environment can be set in advance because they only depend

[15]This holds at least for phase I. The situation is a bit different in phases II and III, because of $^\circ$-annotated term variables.

on the type variables in the typing context and these never change during a run of $\mathrm{TermFind}$ (and thus ExFind). Construction of term environments is more involved and we will require the simplification of a rule of $\mathrm{TermFind}$, as well as tracking of additional information. In the end, this leads to less counterexamples constructed by ExFind than terms found by $\mathrm{TermFind}$.

The remainder of the section is structured as follows:

- We explain the choice of type and relation environments (Subsection 4.5.1),
- investigate requirements on the other extra constructions (Subsection 4.5.2),
- motivate why we restrict term search of $\mathrm{TermFind}$ (Subsection 4.5.3),
- provide a detailed description of ExFind (Subsection 4.5.4),
- sketch a correctness proof (Subsection 4.5.5),
- discuss the incompleteness (Subsection 4.5.6),
- and point to an existing implementation of ExFind (Subsection 4.5.7).

4.5.1 Choosing Type and Relation Environments

First, we consider the choice of type and relation environments. The choice for the type environments and the relation environment is only dependent on the type variables in the type context Γ_T. Since Γ_T is never changed by the rules of $\mathrm{TermFind}$, and hence by the rules of ExFind, we can choose type and relation environments once and for all, such that they are suitable as part of a counterexample at each step of counterexample generation via ExFind.

A suitable semantic interpretation of type variables via the type environments θ_1 and θ_2 is obtained by two considerations. First, for every type variable in the domain of θ_1 and θ_2 there should be a strict and a nonstrict relation between its type interpretations. The smallest (in terms of elements in the semantic domain) such interpretation is the semantics of the unit type: $[\![()]\!]^{\mathsf{fix}}_{\emptyset} = \{\bot, ()\}$. The corresponding strict relation is (the graph of) the identity function $[\![\lambda x :: ().x]\!]^{\mathsf{fix}}_{\emptyset}$ and the nonstrict relation is (the graph of) the constant function $[\![\lambda x :: ().()]\!]^{\mathsf{fix}}_{\emptyset}$. Second, each semantic interpretation with more values does not increase the number of counterexamples we can find. On the one hand, the concrete type instantiation for a type variable α is completely unknown to the term t that gives rise to the counterexample and hence it cannot contain a term of type α that evaluates to any value semantically different from $\bot$; and on the other hand, t can also not distinguish between different values of the type instantiated for α. Hence, different inputs as argument will not influence the behavior of t and consequently not provide new ways to break free theorems. Thus, mapping all type variables to $[\![()]\!]^{\mathsf{fix}}_{\emptyset}$ by θ_1 and θ_2 is a reasonable choice.

Next, we fix the relational interpretation of type variables, i.e., the environment ρ. We distinguish two kinds of type variables: As for $\mathrm{TermFind}$, an external input to ExFind consists of a pair $(\Gamma; \tau)$ where in Γ_T each individual type variable is *-annotated or not. For annotated variables α we accept the strictness condition on $\rho(\alpha)$ and hence map it to a strict relation, more precisely to the identity relation. For unannotated variables we choose the nonstrict relation that is defined by $[\![\lambda x :: ().()]\!]^{\mathsf{fix}}_{\emptyset}$. For those variables we want to show that ignorance of the strictness condition is harmful. We call the just described choice of environments *minimal* and summarize it by the following definition.

Definition 19 (minimal environments)

For a given type context Γ_τ (with possibly *-annotated type variables) the *minimal relation environment* ρ is the map from the set of type variables in Γ_τ to the set of binary relations over the pcpo $[\![()]\!]^{\mathsf{fix}}_{\emptyset}$ that maps each *-annotated type variable in Γ_τ to (the graph of) the identity function $[\![\lambda x :: ().x]\!]^{\mathsf{fix}}_{\emptyset}$ and each unannotated type variable in Γ_τ to (the graph of) the constant function $[\![\lambda x :: ().()]\!]^{\mathsf{fix}}_{\emptyset}$. The *type environments* θ_1 and θ_2 are *minimal* if they take each type variable in Γ_τ to $[\![()]\!]^{\mathsf{fix}}_{\emptyset}$.

Convention 9 ($\Delta^{\mathsf{fix}}_{\tau}$ and $[\![\tau]\!]^{\mathsf{fix}}$)

For the rest of the chapter we take type and relational environments all to be minimal and denote $\Delta^{\mathsf{fix}}_{\tau,\rho}$ with ρ minimal by $\Delta^{\mathsf{fix}}_{\tau}$. Since for minimal environments θ_1 and θ_2 are equal, we define $[\![\tau]\!]^{\mathsf{fix}} = [\![\tau]\!]^{\mathsf{fix}}_{\theta_1} = [\![\tau]\!]^{\mathsf{fix}}_{\theta_2}$.

4.5.2 Requirements for Term Environments

The requirements for term environments are best understood by a preview on a run of ExFind as given by the following example. Analyzing and discussing the strategy of ExFind, the requirements become apparent.

Example 17

Consider the example illustrated in Figure 4.13. The graphic shows a run of ExFind for the input $(\alpha, \beta^*, (\alpha \to \beta) \to [\beta])$. The first two columns correspond to a run of TermFind for the same input (cf. Figure 4.11).

The third column shows the term environment entries constructed by ExFind. In the lower three rows column two and three together state a counterexample to the parametricity theorem's assertion for the typing judgment in the first column (w.r.t. minimal type and relation environment). Row two (also three) reveals the requirements for the construction of term environment entries: To make columns two and three in row two a counterexample to the parametricity theorem three requirements have to be met:

1. The interpretations of f must be related (to meet the conditions of the theorem).
2. The interpretations of $f\ \bot_\alpha$ must not be related (to have a counterexample).
3. The interpretations of $\bot_\alpha$ must not be related (as a consequence of 1 and 2).

To meet requirement 3 we ensure that the relational interpretation of α is nonstrict. To meet the first two requirements, the interpretations of f have to be designed carefully. Concerning the interpretations of f's result type, we need a pair of related values and a pair of unrelated values, such that the interpretations of f can map potentially related inputs to related outputs and definitely unrelated inputs, in particular the interpretations of $\perp_\alpha$, to unrelated outputs. The related and the unrelated pair of values are generated in the first row as interpretations for x. Figure 4.13 shows only the related pair $(\sigma_1(x), \sigma_2(x))$. The unrelated pair is given implicitly. It is (in general, not only in the example) the pair $(\sigma_1(x), \perp)$. The environment entries for x in the first row and the fact that $(\sigma_1(x), \perp)$ is not in the logical relation are employed in the design of the interpretations for f.

Now take a closer look at the first row. It presents a state in the second phase of ExFind (and also TermFind). TermFind's goal is to employ the $^\circ$-annotated variable x somehow, and ExFind additionally searches for a related and an unrelated pair of interpretations for this x. As already explained, the interpretations are needed to design the interpretations of f when x is replaced. In the row itself columns two and three do not state a counterexample to the parametricity theorem for the typing judgment in column one. That is because of the different task of ExFind's second phase.

Last but not least consider row four. Columns two and three, again under minimal type/relation environments, state a complete counterexample to the parametricity theorem. Nonetheless, concerning a counterexample to the free theorem we lost information (regarding only column two and three). The free theorem for the example (functional and already instantiated with minimal environments) looks as follows:

For all related $g :: () \to ()$ and $h :: () \to ()$, i.e., g and h with $g \equiv \lambda y :: ().h\ ((\lambda x :: ().())\ y)$ we have

$$t_0\ g \equiv t_0\ h$$

Going from row three to four, we lose the term environment entries that provide suitable choices for g and h to set up a counterexample. An extra construction shown in column four remembers these entries. It is called disrelator. The disrelator ϖ tracks the information *why* the parametricity theorem does not hold. It is a tuple, with the first component the type on whose interpretation the counterexample is built (the point where the parametricity theorem originally breaks), and in the second component it logs how to navigate through all liftings of the logical relation to the original breach of it. In particular, going from row three to four, the disrelator stores the interpretations for f. Going from row two to three it stores the pair $(\mathbf{head}, \mathbf{head})$. This entry allows to uncover that the term constructed in row two wraps the term from row one in a list. The entry states that the results of the interpretations of f applied to $\perp$ are not related because the heads of the resulting lists are not related. In other words, it states that the list wrapping has to be undone to reveal the original breach of the logical relation.

$$\begin{tikzcd}
{} \arrow[r] & {} \arrow[d] \arrow[r] & {} \arrow[d] \arrow[r] & {} \arrow[d] \\
\alpha, \beta^*, x^\circ :: \beta \Vdash^\circ t_3 :: \beta \arrow[u, "(\textsc{Var}^\circ)"'] & t_3 = x \arrow[d] & \sigma_1 = \{x \mapsto ()\}, \sigma_2 = \{x \mapsto ()\} \arrow[d] & \varpi = (\beta, [\,]) \arrow[d] \\
\alpha, \beta^*, f :: \alpha \to \beta \Vdash t_2 :: \beta \arrow[u, "(\textsc{Bottom}{\to}')"'] & t_2 = f \perp_\alpha \arrow[d] & \sigma_1 = \{f \mapsto \mathbf{\lambda x.()}\}, \sigma_2 = \{f \mapsto \mathbf{\lambda x.case\ x\ of}\ \{() \to ()\}\} \arrow[d] & \varpi = (\beta, [\,]) \arrow[d] \\
\alpha, \beta^*, f :: \alpha \to \beta \Vdash t_1 :: [\beta] \arrow[u, "(\textsc{Wrap})"'] & t_1 = [f \perp_\alpha] \arrow[d] & \sigma_1 = \{f \mapsto \mathbf{\lambda x.()}\}, \sigma_2 = \{f \mapsto \mathbf{\lambda x.case\ x\ of}\ \{() \to ()\}\} \arrow[d] \arrow[dr, bend left] & \varpi = (\beta, [(\mathbf{head}, \mathbf{head})]) \arrow[d] \\
\alpha, \beta^* \Vdash t_0 :: (\alpha \to \beta) \to [\beta] \arrow[u, "(\textsc{Abs})"'] & t_0 = \lambda f :: \alpha \to \beta.[f \perp_\alpha] & \sigma_1 = \emptyset, \sigma_2 = \emptyset & \varpi = (\beta, [(\mathbf{\lambda t.t}\ (\mathbf{\lambda x.}()), \mathbf{\lambda t.t}\ (\mathbf{\lambda x.case\ x\ of}\ \{() \to ()\})), (\mathbf{head}, \mathbf{head})])
\end{tikzcd}$$

Figure 4.13: The constructions of ExFind for input $(\alpha, \beta^*, (\alpha \to \beta) \to [\beta])$

As final remark, note that for stating the counterexample we set g to $\lambda x :: ().()$ and h to $\lambda x :: ().\mathbf{case}\ x\ \mathbf{of}\ \{() \to ()\}$, whose semantic interpretations correspond to the values generated in the term environments and stored in the disrelator. Hence, it is necessary that each of these entries is the semantics of a term. Since the environment entries for f are generated from entries of the function's result type, the requirement is necessary for variables of each type (and not only for function types). That we meet it is not immediately clear because there are semantic values that do not correspond to the semantics of a term.

From Example 17 we can observe the strategy of ExFind. It is closely tied to the "introduce-and-propagate"-strategy of TermFind. Where TermFind introduces a local disrelation, ExFind generates, based on that disrelation, a counterexample to the parametricity theorem for the current type. Then ExFind, alike the propagation in TermFind, lifts the example step by step to a counterexample to the parametricity theorem of the input type. As for TermFind, phase II has a special role. We enter the phase always adding a $^\circ$-annotated variable that is later replaced by an application of a function f to a term for which we have ensured unrelated semantic interpretations (under the assumption that unpointed types must have nonstrict relational interpretations). Thus, for phase II, ExFind in particular has to create sufficient information about the $^\circ$-annotated variable to employ it for the construction of interpretations of f, which shall be related but yield unrelated values for the unrelated input. This information is a pair of related interpretations and a pair of unrelated interpretations for the variable.

To create counterexamples the way just described three restrictions are necessary (at each step of the counterexample generation):

1. For each term variable $x :: \tau$ in the term context we need related semantic interpretations, i.e., interpretations related by $\Delta^{\mathsf{fix}}_{\tau}$.

2. For each $^\circ$-annotated term variable $x :: \tau$ in the term context we need unrelated semantic interpretations, i.e., interpretations not related by $\Delta^{\mathsf{fix}}_{\tau}$.

3. The interpretations of $\perp_\tau$ must not be related for τ unpointed, i.e., unpointed types have nonstrict relational interpretations.

Whether we meet requirement 3, or not, depends on the concrete choices of type/relation environments. For our choice of minimal environments we meet it, as stated by the next lemma. The lemma also states that pointed types have strict relational interpretations (as direct consequence of Lemma 12).

Lemma 13 (properties of minimal environments) Let $\Gamma_\mathcal{T}$ a type context with possibly *-annotated type variables. If τ is closed under $\Gamma_\mathcal{T}$, then

$$(\bot, \bot) \in \Delta^{\mathbf{fix}}_\tau \qquad \text{if } \Gamma_\mathcal{T} \vdash \tau \in \mathsf{Pointed} \tag{1}$$
$$(\bot, \bot) \notin \Delta^{\mathbf{fix}}_\tau \qquad \text{if } \Gamma_\mathcal{T} \vdash \tau \notin \mathsf{Pointed} \tag{2}$$

Proof. Induction on the structure of τ, employing the definition of the logical relation and the definition of minimal environments. □

Requirements 1 and 2 can be summarized by a single requirement: For each type τ we must be able to construct values $\mathbf{x}$ and $\mathbf{y}$ with $(\mathbf{x}, \mathbf{y}) \in \Delta^{\mathbf{fix}}_\tau$ and also values $\mathbf{x}'$ and $\mathbf{y}'$ with $(\mathbf{x}', \mathbf{y}') \in [\![\tau]\!]^{\mathsf{fix}} \times [\![\tau]\!]^{\mathsf{fix}}$ and $(\mathbf{x}', \mathbf{y}') \notin \Delta^{\mathbf{fix}}_\tau$. The construction of such values is described in the next definition. The subsequent lemma states that the constructed values are sufficient to establish the required related/unrelated pair for each type.

Definition 20 (plus-value) Let $\Gamma_\mathcal{T}$ a type context and τ a type closed under $\Gamma_\mathcal{T}$. The value $\mathbf{p}^+_\tau$ is called *plus-value* of type τ and recursively defined over the structure of τ by

$$\mathbf{p}^+_\alpha = () \qquad \mathbf{p}^+_{Nat} = 0 \qquad \mathbf{p}^+_{()} = () \qquad \mathbf{p}^+_{[\tau]} = [\mathbf{p}^+_\tau]$$
$$\mathbf{p}^+_{(\tau_1,\tau_2)} = (\mathbf{p}^+_{\tau_1}, \mathbf{p}^+_{\tau_2}) \qquad \mathbf{p}^+_{Either\ \tau_1\ \tau_2} = \mathrm{Left}\ \mathbf{p}^+_{\tau_1} \qquad \mathbf{p}^+_{\tau_1 \to \tau_2} = \lambda \mathbf{x}.\mathbf{p}^+_{\tau_2}$$

Lemma 14 (properties of minimal environments) Let $\Gamma_\mathcal{T}$ a type context with possibly *-annotated type variables. If τ is closed under $\Gamma_\mathcal{T}$, then

$$(\mathbf{p}^+_\tau, \mathbf{p}^+_\tau) \in \Delta^{\mathbf{fix}}_\tau \tag{1}$$
$$(\mathbf{p}^+_\tau, \bot) \notin \Delta^{\mathbf{fix}}_\tau \tag{2}$$

Proof. Induction on the structure of τ, employing the definition of the logical relation, the definition of minimal environments and Definition 20. □

The requirements on term environment entries that we just identified and showed to be met easily are necessary but not yet sufficient to lift counterexamples in general. Knowing an arbitrary pair of related or unrelated values is not always enough. We already notice that further restrictions on term environments are necessary when we construct appropriate entries for f in

Example 17. We cannot simply take plus-values as entries for f. The values need to fulfill a kind of "disrelation propagation"-property, depending on the actual appearance of the term variable in the term we generate. The required property is best illustrated by another example for a type involving $Either$.

EXAMPLE 18

Consider the input $(\alpha; Either\ Int\ (\alpha \to Int) \to Int)$. TermFind, and also ExFind, will use the rules (ABS), (DIST$_2$), (BOTTOM→′) and (VAR°). The concrete instance of (DIST$_2$), as TermFind creates it, looks as follows:

$$\frac{\alpha, f :: \alpha \to Nat \Vdash f\ \bot_\alpha :: Nat}{\alpha, e :: Either\ Nat\ (\alpha \to Nat) \Vdash fromRight_{Nat}\ e\ \bot_\alpha :: Nat}\ (\text{DIST}_2)$$

Now, for ExFind a task is to find related environment entries $\mathbf{e_1}$ and $\mathbf{e_2}$ for e, that ensure

$$(\llbracket fromRight_{Nat}\ e\ \bot_\alpha \rrbracket^{\mathsf{fix}}_{[e \mapsto \mathbf{e_1}]}, \llbracket fromRight_{Nat}\ e\ \bot_\alpha \rrbracket^{\mathsf{fix}}_{[e \mapsto \mathbf{e_2}]}) \notin \Delta^{\mathsf{fix}}_{Nat} \qquad (4.1)$$

i.e., that the term in the conclusion of (DIST$_2$) and the choices for the environment entries for e establish a counterexample to the parametricity theorem for type Nat and typing context $\alpha, e :: Either\ Nat\ (\alpha \to Nat)$.

Obviously, we cannot choose plus-values of type $Either\ Nat\ (\alpha \to Nat)$ as environment entries for e. They are Left -values and hence the pair in (4.1) would be $(\bot, \bot) \in \Delta^{\mathsf{fix}}_{Nat}$. To find appropriate entries, we use the ones from the premise of the rule. For these we already know (if ExFind works according to our strategy) that, with the constructed term in the premise, they establish a counterexample to the parametricity theorem that fits to the premise. Thus, we employ the environment entries for f in the premise and modify them appropriately. For (DIST$_2$), led by the way the rule alters the constructed term, we wrap the entries as Right-values. Hence, we set $\mathbf{e_1} = \text{Right}\ \mathbf{f_1}$ and $\mathbf{e_2} = \text{Right}\ \mathbf{f_2}$, if $\mathbf{f_1}$ and $\mathbf{f_2}$ are the respective environment entries for f in the premise.

Note that it is really necessary to rely on the entries for f in the premise. They already were designed such that their results are unrelated if both are applied to $\bot$. And this is essential to obtain a counterexample.

As it is essential in Example 18 to generate new entries in the term environments of the conclusion of the rule with the help of the entries from the premise, it is always essential when appearances of a term variable in the premise's term are (in the conclusion) replaced by a term containing a new term variable.

Before we go into the details of environment construction in Subsection 4.5.4, we consider a general problem of the construction approach that forces us to simplify the rule (ARROW→′) and thereby restrict the search space.

4.5.3 Restrictions to TermFind

Reconsider rule (ARROW→′):

$$\frac{\Gamma_{\mathcal{T}} \vdash \tau_2, \tau_3 \in \mathsf{Pointed} \quad \Gamma, w :: \tau_1, g :: \tau_2 \to \tau_3 \Vdash t_1 :: \tau_2 \quad \Gamma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \Vdash t_2[f\ (\lambda x :: \tau_1.t_1[\lambda z :: \tau_2.f\ (\lambda u :: \tau_1.z)/g, x/w])/y] :: \tau}$$

Unfortunately, for this rule we encounter situations where it is impossible to enrich it the way that the conclusion states a counterexample (to the respective free theorem for the conclusion's type and context), even if the premises do (w.r.t. their respective types and contexts). In these situations TermFind creates a term that does not give rise to a counterexample.[16] ExFind can only abort in such situations, telling it cannot find a counterexample.

The reason for such a situation is that (ARROW→′) has two premises where term variables in the typing contexts may overlap and consequently ExFind may create different term environment entries for the same term variable. As Example 18 and the surrounding discussion in Subsection 4.5.2 clarify, term environment entries must comply to certain conditions. These conditions can be incompatible concerning the environments created in the different premises of (ARROW→′). For example, a variable can be mapped to a Left-value in an environment of one premise, but in the corresponding environment in the other premise it is mapped to a Right-value. Both choices can be essential. In such situations the environments cannot be merged to generate a counterexample w.r.t. the conclusion of (ARROW→′). We solve the problem by checking compatibility of two different values assigned to the same variable. The compatibility check requires us to track the history of a value assignment to a variable. ExFind keeps track of such histories via a *history environment*. The concrete handling is given in the next subsection.

But the rule (ARROW→′) poses another problem. The function f that is only present in the typing context of the conclusion may be employed twice in the term constructed in the conclusion. To establish a counterexample for the conclusion of (ARROW→′), f may need to meet different conditions for the different occurrences. To design a function f that satisfies the different conditions is not an easy task and we bypass it by a simplification of (ARROW→′). We remove $g :: \tau_2 \to \tau_3$ from the second premise and obtain, as a replacement for (ARROW→′), the rule

$$\frac{\Gamma_{\mathcal{T}} \vdash \tau_2, \tau_3 \in \mathsf{Pointed} \quad \Gamma, w :: \tau_1 \Vdash t_1 :: \tau_2 \quad \Gamma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \Vdash t_2[f\ (\lambda x :: \tau_1.t_1[x/w])/y] :: \tau} \ (\text{ARROW}\to^*)$$

with only one occurrence of f in the conclusion. The simplification (that g is omitted in the first premise) eases construction of environment entries for f. Unfortunately, we lose counterexamples by this simplification.

[16] As we discuss in Subsection 4.5.6 that does not necessarily mean that there is no counterexample.

A more detailed discussion on the necessity of the simplification and examples why the history environment is essential are given in Subsection 4.5.6.

Besides changing (ARROW$\rightarrow$') to (ARROW$\rightarrow^*$) we take over the rules of TermFind unchanged. Having fixed the rule set of ExFind, we take a closer look on how to precisely extend the rules with extra constructions.

4.5.4 Creating Extra Information — Concrete Constructions

Up to now, we informally argued about the requirements on the extra constructions of ExFind and we fixed the concrete rule set for the algorithm. In this subsection we formally describe ExFind. In the first part we formalize the requirements on its constructions, in the second part we define auxiliary constructions and in part three we provide a complete formalization of ExFind.

Formal Requirements on the Extra Constructions

First, we characterize the term environments constructed by ExFind. As already acknowledged in Example 17, in rules where variables are replaced by a function application in the conclusion (i.e., the rules (APP'$^\circ$), (ARROW$\rightarrow^*$) and (BOTTOM$\rightarrow^\circ$)), we have to know a pair of related and a pair of unrelated values for the variable that gets replaced. Both usually must satisfy extra conditions that are specified by the variable's use in the term in the premise. As also noticed, we provide these pairs of values via the term environments: σ_1, σ_2 and the extra condition that for each $^\circ$-annotated term variable $(x^\circ :: \tau) \in \Gamma_V$ we have $(\sigma_1(x), \bot) \notin \Delta^{\mathsf{fix}}_{\tau}$.

The conditions are summarized in the following definition.

DEFINITION 21 (result environment)

The tuple (σ_1, σ_2) of term environments is called a *result environment* w.r.t. typing context Γ, such as it appears in the internal input to ExFind, if the domain of the term environments is the set of term variables in Γ_V and for every x associated with type τ' in Γ_V it holds that:

- $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathsf{fix}}_{\tau'}$ and
- $(\sigma_1(x), \bot) \notin \Delta^{\mathsf{fix}}_{\tau'}$, if x annotated by $^\circ$ in Γ_V.

To ease notation, we define the operation $(\cdot)^\perp$ for term environments.

DEFINITION 22 ($\sigma^\perp$)

Let Γ_V a term context with possibly $^\circ$-annotated variables and σ a term environment with the variables in Γ_V as domain. Then

$$\sigma^\perp = \sigma[\overline{x_n \mapsto \bot}]$$

where $\overline{x_n}$ are the $^\circ$-annotated variables in Γ_V.

With respect to the just described result environments, the term produced by ExFind should fulfill the following definition (to serve as part of a counterexample to the parametricity theorem).

Definition 23 **(result term)** The term t is a *result term* w.r.t. the internal input $\mathcal{I}^{int} = (\Gamma; \tau)$ and a respective result environment (σ_1, σ_2), if $([\![t]\!]^{\mathsf{fix}}_{\sigma_1}, [\![t]\!]^{\mathsf{fix}}_{\sigma_2^\perp}) \notin \Delta^{\mathbf{fix}}_{\tau}$.

For technical reasons, we require for terms generated by the second phase of ExFind an even stronger condition.

Definition 24 **(strong result term)** A result term t w.r.t. an internal input and a respective result environment (σ_1, σ_2) is called *strong* if $[\![t]\!]^{\mathsf{fix}}_{\sigma_2^\perp} = \perp$.

Auxiliary Constructions

We characterized how the term environments should behave w.r.t. each other and also w.r.t. the term ExFind creates. But, the description is still not sufficient to give construction rules for the environments. Two extra constructions are necessary. First, we have to trace back where exactly the disrelation, breaking the parametricity theorem, was injected. Here the disrelator, already described in Example 17 comes in — but with a slightly different motivation. During the construction of the term environments it is necessary to find suitable entries for $f :: (\tau_1 \to \tau_2) \to \tau_3$ in the conclusion of (Arrow$\to^*$). That means, two related interpretations $\sigma_1(f)$ and $\sigma_2(f)$ of f that yield unrelated values for two specific unrelated inputs, i.e., for the semantic interpretations of $\lambda x :: \tau_1.t_1[x/w]$ (cf. rule (Arrow$\to^*$)). The construction requires information about why the semantic interpretations of $\lambda x :: \tau_1.t_1[x/w]$ are not related by the logical relation.

We describe the "why"-information by a disrelator. It consists of the type where the logical relation was originally broken and a series of, by the logical relation related, semantic function pairs that disassemble the actual unrelated values to values of the type the disrelation goes back to. In other words, the function pairs undo liftings of the logical relation.

Definition 25 **(disrelator)** Let $\Gamma_\mathcal{T}$ a type context in $\lambda^{\alpha}_{\mathbf{fix}^*}$, τ, τ' types closed under $\Gamma_\mathcal{T}$ and $(\mathbf{t_1}, \mathbf{t_2}) \in [\![\tau]\!]^{\mathsf{fix}} \times [\![\tau]\!]^{\mathsf{fix}}$. The pair $\varpi = (\tau', l)$ with $l = [(\mathbf{v_1}, \mathbf{v_1'}), \ldots, (\mathbf{v_n}, \mathbf{v_n'})]$ and each $(\mathbf{v_i}, \mathbf{v_i'})$ out of $\{(\mathbf{head}, \mathbf{head}), (\mathbf{fst}, \mathbf{fst}), (\mathbf{snd}, \mathbf{snd}), (\mathbf{fromLeft}, \mathbf{fromLeft}), (\mathbf{fromRight}, \mathbf{fromRight}), (\lambda\mathbf{u}.\mathbf{u}\ \mathbf{w}, \lambda\mathbf{u}.\mathbf{u}\ \mathbf{w'})\}$, where $(\mathbf{w}, \mathbf{w'}) \in \Delta^{\mathbf{fix}}_{\tau''}$ for appropriate τ'', is called a *disrelator* of $(\mathbf{t_1}, \mathbf{t_2})$ if

$$(\mathbf{v_n}(\ldots(\mathbf{v_1}\ \mathbf{t_1})), \mathbf{v_n'}(\ldots(\mathbf{v_1'}\ \mathbf{t_2}))) = (\mathbf{t}_{\tau'}, \perp) \notin \Delta^{\mathbf{fix}}_{\tau'}$$

with $\mathbf{t}_{\tau'}$ a constant function or $\perp$ if τ' is an arrow type.

Example 19

For the value pair

$$(\mathbf{t_1}, \mathbf{t_2}) = (\lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \rightarrow \text{Left}\ \mathbf{0}\}, \lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \rightarrow \text{Left}\ \bot\})$$

that is an element of $[\![Nat \rightarrow Nat]\!]^{\mathsf{fix}} \times [\![Nat \rightarrow Nat]\!]^{\mathsf{fix}}$ under the empty context $\Gamma_{\mathcal{T}} = \emptyset$, the pair

$$([(\lambda\mathbf{u}.\mathbf{u}\ \mathbf{0}, \lambda\mathbf{u}.\mathbf{u}\ \mathbf{0}), (\mathbf{fromLeft}, \mathbf{fromLeft})], Nat)$$

forms a disrelator. Applying it yields:

$$\begin{array}{rl} & (\mathbf{fromLeft}\ ((\lambda\mathbf{u}.\mathbf{u}\ \mathbf{0})\ (\lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \rightarrow \text{Left}\ \mathbf{0}\})) \\ & \quad , \mathbf{fromLeft}\ ((\lambda\mathbf{u}.\mathbf{u}\ \mathbf{0})\ (\lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \rightarrow \text{Left}\ \bot\}))) \\ = & (\mathbf{fromLeft}\ ((\lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \rightarrow \text{Left}\ \mathbf{0}\})\ \mathbf{0}) \\ & \quad , \mathbf{fromLeft}\ ((\lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \rightarrow \text{Left}\ \bot\})\ \mathbf{0})) \\ = & (\mathbf{fromLeft}\ (\text{Left}\ \mathbf{0}), \mathbf{fromLeft}\ (\text{Left}\ \bot)) \\ = & (\mathbf{0}, \bot) \notin \Delta^{\mathsf{fix}}_{Nat} = id_{\mathbb{N}_\bot} \end{array}$$

Note that the formal definition of disrelator is even stronger than the informal description above. The definition forces the initial disrelation uncovered by the disrelator to be of a special structure. This structure will be guaranteed by the constructions of ExFind and necessary for the proof of Lemma 15 (stated further below).

The next definition describes how to generate functions of arbitrary argument and result type, that are related by the logical relation but yield unrelated results for two unrelated inputs $\mathbf{t_1}$ and $\mathbf{t_2}$ whose disrelation is described by a disrelator ϖ.

Convention 10

We employ the syntax of case expressions also on the semantic level.

Definition 26

Let $\Gamma_{\mathcal{T}}$ a type context in $\lambda^{\alpha}_{\mathsf{fix}*}$, τ, τ' types closed under $\Gamma_{\mathcal{T}}$ and τ' pointed under $\Gamma_{\mathcal{T}}$. Furthermore, let $(\mathbf{t}'_1, \mathbf{t}'_2) \in \Delta^{\mathsf{fix}}_{\tau'}$ and $\varpi = (\tau'', l)$ a disrelator for some pair $(\mathbf{t_1}, \mathbf{t_2}) \in [\![\tau]\!]^{\mathsf{fix}} \times [\![\tau]\!]^{\mathsf{fix}}$. The pair $(\mathbf{g_1}(\mathbf{t}'_1), \mathbf{g_2}(\mathbf{t}'_2))^{\Gamma_{\mathcal{T}}, \tau}_{\varpi}$, abbreviated by $(\mathbf{g_1}, \mathbf{g_2})$ if all parameters are clear from the context, is defined recursively over the structure of τ and the disrelator as follows. The variable i ranges over 1 and 2, and α is a type variable.

$\tau = \alpha$ and pointed $\tau = ()$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{() \rightarrow\ \mathbf{t}'_\mathbf{i}\}$
$\tau = \alpha$ and unpointed	$\mathbf{g_1} = \lambda\mathbf{z}.\mathbf{t}'_1$, $\mathbf{g_2} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{() \rightarrow\ \mathbf{t}'_2\}$
$\tau = Nat$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{\mathbf{0} \rightarrow \mathbf{t}'_\mathbf{i}\}$

$\tau = [\tau_1]$ and $l = [\,]$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{[\mathbf{x}] \to \mathbf{t'_i}\}$
$\tau = [\tau_1]$ and $l = (\mathbf{v}, \mathbf{v'}) : l'$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{[\mathbf{x}] \to \mathbf{h_i\ x}\}$ with $(\mathbf{h_1}, \mathbf{h_2}) = (\mathbf{g_1}(\mathbf{t'_1}), \mathbf{g_2}(\mathbf{t'_2}))^{\Gamma_\mathcal{T},\tau_1}_{(\tau'',l')}$
$\tau = (\tau_1, \tau_2)$ and $l = [\,]$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{(\mathbf{x}, \mathbf{y}) \to \mathbf{t'_i}\}$
$\tau = (\tau_1, \tau_2)$ and $l = (\mathbf{v}, \mathbf{v'}) : l'$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{(\mathbf{x}, \mathbf{y}) \to \mathbf{h_{k_i}\ x\ y}\}$ with $(\mathbf{h_{k_1}}, \mathbf{h_{k_2}}) = \begin{cases} (\lambda\mathbf{x}.\lambda\mathbf{y}.\mathbf{k_1\ y}, \lambda\mathbf{x}.\lambda\mathbf{y}.\mathbf{k_2\ y}) & \text{if } \mathbf{v} = \mathbf{snd} \\ (\mathbf{g_1}(\mathbf{k_1}), \mathbf{g_2}(\mathbf{k_2}))^{\Gamma_\mathcal{T},\tau_1}_{(\tau'',l')} & \text{otherwise} \end{cases}$ and $(\mathbf{k_1}, \mathbf{k_2}) = \begin{cases} (\lambda\mathbf{x}.\mathbf{t'_1}, \lambda\mathbf{x}.\mathbf{t'_2}) & \text{if } \mathbf{v} = \mathbf{fst} \\ (\mathbf{g_1}(\mathbf{t'_1}), \mathbf{g_2}(\mathbf{t'_2}))^{\Gamma_\mathcal{T},\tau_2}_{(\tau'',l')} & \text{otherwise} \end{cases}$
$\tau = \mathit{Either}\ \tau_1\ \tau_2$ and $l = [\,]$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{\mathbf{Left\ x} \to \mathbf{t'_i}; \mathbf{Right\ x} \to \mathbf{t'_i}\}$
$\tau = \mathit{Either}\ \tau_1\ \tau_2$ and $l = (\mathbf{v}, \mathbf{v}) : l'$ with $\mathbf{v} = \mathbf{fromLeft}$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{\mathbf{Left\ x} \to \mathbf{h_i\ x}\}$ with $(\mathbf{h_1}, \mathbf{h_2}) = (\mathbf{g_1}(\mathbf{t'_1}), \mathbf{g_2}(\mathbf{t'_2}))^{\Gamma_\mathcal{T},\tau_1}_{(\tau'',l')}$
$\tau = \mathit{Either}\ \tau_1\ \tau_2$ and $l = (\mathbf{v}, \mathbf{v}) : l'$ with $\mathbf{v} = \mathbf{fromRight}$	$\mathbf{g_i} = \lambda\mathbf{z}.\mathbf{case\ z\ of}\ \{\mathbf{Right\ x} \to \mathbf{h_i\ x}\}$ with $(\mathbf{h_1}, \mathbf{h_2}) = (\mathbf{g_1}(\mathbf{t'_1}), \mathbf{g_2}(\mathbf{t'_2}))^{\Gamma_\mathcal{T},\tau_2}_{(\tau'',l')}$
$\tau = \tau_1 \to \tau_2$ and $l = [\,]$	$\mathbf{g_1} = \lambda\mathbf{z}.\mathbf{h_1}\ (\mathbf{z\ p^+_{\tau_1}})$, $\mathbf{g_2} = \lambda\mathbf{z}.\mathbf{h_2}\ (\mathbf{z\ p^+_{\tau_1}})$ with $(\mathbf{h_1}, \mathbf{h_2}) = (\mathbf{g_1}(\mathbf{t'_1}), \mathbf{g_2}(\mathbf{t'_2}))^{\Gamma_\mathcal{T},\tau_2}_{(\tau'',[\,])}$
$\tau = \tau_1 \to \tau_2$ and $l = (\mathbf{v}, \mathbf{v'}) : l'$	$\mathbf{g_1} = \lambda\mathbf{z}.\mathbf{h_1}\ (\mathbf{v\ z})$, $\mathbf{g_2} = \lambda\mathbf{z}.\mathbf{h_2}\ (\mathbf{v'\ z})$ with $(\mathbf{h_1}, \mathbf{h_2}) = (\mathbf{g_1}(\mathbf{t'_1}), \mathbf{g_2}(\mathbf{t'_2}))^{\Gamma_\mathcal{T},\tau_2}_{(\tau'',l')}$

Definition 26 satisfies the properties we require for a pair $(\mathbf{g_1}, \mathbf{g_2})$.

Lemma 15 Let $\Gamma_\mathcal{T}$ a type context in $\lambda^\alpha_{\mathbf{fix}^*}$, τ, τ' types closed under $\Gamma_\mathcal{T}$ and τ' pointed under $\Gamma_\mathcal{T}$. Furthermore, let $(\mathbf{t'_1}, \mathbf{t'_2}) \in \Delta^{\mathbf{fix}}_{\tau'}$ and $\varpi = (\tau'', l)$ a disrelator for some pair $(\mathbf{t_1}, \mathbf{t_2}) \in [\![\tau]\!]^{\mathsf{fix}} \times [\![\tau]\!]^{\mathsf{fix}}$. Then the construction from Definition 26 returns a pair $(\mathbf{g_1}, \mathbf{g_2}) = (\mathbf{g_1}(\mathbf{t'_1}), \mathbf{g_2}(\mathbf{t'_2}))^{\Gamma_\mathcal{T},\tau}_{\varpi}$ with

$$(\mathbf{g_1}, \mathbf{g_2}) \in \Delta^{\mathbf{fix}}_{\tau \to \tau'} \text{ and } (\mathbf{g_1\ t_1}, \mathbf{g_2\ t_2}) = (\mathbf{t'_1}, \bot)\,.$$

The proof is given in Appendix A. To provide an intuition about the construction, we consider the construction for the environment entries for f in Example 16.

EXAMPLE 20

Let us first describe a situation where the construction from Definition 26 is employed. In Example 16 ExFind applies rule (ARROW$\rightarrow^*$). During the rule's application, the environment entries for f are constructed, i.e., the semantic interpretations of p and q (see Example 16). The concrete construction is described in Definition 30. We only point to the use of Definition 26. It enables us to construct a pair of related functions that yield unrelated results for (a special pair of) unrelated inputs. This is necessary to propagate the disrelation we introduced via the first premise of (ARROW$\rightarrow^*$). In the concrete example we construct a pair of functions $(\mathbf{h_1}, \mathbf{h_2})$ as

$$(\mathbf{h_1}, \mathbf{h_2}) = (\mathbf{g_1}(\text{Left } \mathbf{0}), \mathbf{g_2}(\text{Left } \mathbf{0}))^{\alpha,\beta^*,[\alpha]}_{(\alpha,[(\mathbf{head},\mathbf{head})])}$$

In particular the entities in Definition 26 are specialized as:

- $\Gamma_T = \alpha, \beta^*$
- $\tau = [\alpha]$, $\tau' = \mathit{Either\ Nat\ \beta}$, $\tau'' = \alpha$,
- $(\mathbf{t_1'}, \mathbf{t_2'}) = (\text{Left } \mathbf{0}, \text{Left } \mathbf{0}) \in \Delta^{\mathbf{fix}}_{\mathit{Either\ Nat\ \beta}}$ and
- $\varpi = (\alpha, [(\mathbf{head}, \mathbf{head})])$ as a disrelator for
- $(\mathbf{t_1}, \mathbf{t_2}) = ([\bot], [\bot]) \in [\![[\alpha]]\!]^{\mathbf{fix}} \times [\![[\alpha]]\!]^{\mathbf{fix}}$.

By Lemma 15, the pair $(\mathbf{h_1}, \mathbf{h_2})$ will satisfy $(\mathbf{h_1}, \mathbf{h_2}) \in \Delta^{\mathbf{fix}}_{[\alpha]\rightarrow \mathit{Either\ Nat\ \beta}}$ and $(\mathbf{h_1}\ [\bot], \mathbf{h_2}\ [\bot]) = (\text{Left } \mathbf{0}, \bot)$.

For every application of the construction in Definition 26 that ExFind performs, it is guaranteed that (using notation of Lemma 15) $(\mathbf{t_1'}, \bot) \notin \Delta^{\mathbf{fix}}_{\tau'}$ holds. In this particular example, we have $(\text{Left } \mathbf{0}, \bot) \notin \Delta^{\mathbf{fix}}_{\mathit{Either\ Nat\ \beta}}$. The components of $(\text{Left } \mathbf{0}, \text{Left } \mathbf{0})$ are the environment entries of the $^\circ$-annotated term variable in the second premise of (ARROW$\rightarrow^*$), and for these — by the definition of result environment (Definition 21) as one will be constructed for this premise — it is guaranteed that the entry of the first environment is not related to $\bot$.

Let us investigate the construction of $(\mathbf{h_1}, \mathbf{h_2})$ for this example. Construction starts with the case for list types and a non-empty list as second component of the disrelator. Hence, we define

$$(\mathbf{h_1}, \mathbf{h_2}) = (\lambda \mathbf{z}.\mathbf{case\ z\ of}\ \{[\mathbf{x}] \rightarrow\ \mathbf{h_1'}\ \mathbf{x}\}, \lambda \mathbf{z}.\mathbf{case\ z\ of}\ \{[\mathbf{x}] \rightarrow\ \mathbf{h_2'}\ \mathbf{x}\})$$

where

$$(\mathbf{h_1'}, \mathbf{h_2'}) = (\mathbf{g_1}(\text{Left } \mathbf{0}), \mathbf{g_2}(\text{Left } \mathbf{0}))^{\alpha,\beta^*,\alpha}_{(\alpha,[])}$$

The tuple $(\mathbf{head}, \mathbf{head})$ in the disrelator tells that $\mathbf{t_1}$ and $\mathbf{t_2}$ are both singleton (as outcome of our constructions) lists. Hence for all non-singleton lists as input the function's output can (and if inputs are related it must) be related. Related outputs are $(\bot, \bot)$ since τ' is pointed. Consequently, we can and do construct a pair of functions that both yield $\bot$ for all non-singleton lists. The behavior of $\mathbf{h_1}$ and $\mathbf{h_2}$ on singleton lists is defined via a recursive call to the construction of Definition 26. We define the functions as given in Definition 26 for the case where τ is the element type of the list, in the example a type variable that is unpointed. We get

$$(\mathbf{h_1'}, \mathbf{h_2'}) = (\lambda \mathbf{z'}.\mathbf{t_1'}, \lambda \mathbf{z'}.\mathbf{case\ z'\ of}\ \{() \rightarrow\ \mathbf{t_2'}\})$$

The functions yield unrelated values exactly when the input to the second function is $\perp$. Since there is no pair $(\mathbf{x}, \perp) \in \Delta_{\alpha}^{\mathbf{fix}}$, it holds that $(\mathbf{h_1'}, \mathbf{h_2'}) \in \Delta_{\alpha \to Either\ Nat\ \beta}^{\mathbf{fix}}$ and as well that $(\mathbf{h_1'}\ \perp, \mathbf{h_2'}\ \perp) = (\mathrm{Left}\ \mathbf{0}, \perp) \notin \Delta_{Either\ Nat\ \beta}^{\mathbf{fix}}$. Consequently, we have

$$(\mathbf{h_1}, \mathbf{h_2}) \in \Delta_{[\alpha] \to Either\ Nat\ \beta}^{\mathbf{fix}}$$

and

$$(\mathbf{h_1}\ [\perp], \mathbf{h_2'}\ [\perp]) = (\mathrm{Left}\ \mathbf{0}, \perp) \notin \Delta_{Either\ Nat\ \beta}^{\mathbf{fix}}$$

as required.

The last auxiliary construction missing to create the term environments is a history environment. Consider again the rule (Arrow→*). It has (besides the pointedness checks) two premises whose typing contexts, and in particular term contexts, may overlap. Hence, for each premise we have constructed entries for the term environments, such that these form a result environment. To construct a result environment for the conclusion, we need to take over the term environment entries from the premises. But, if these environments' domains overlap, which entry shall we take if we have different entries for the same variable? In the conclusion of (Arrow→*) the terms constructed by the different premises (terms t_1 and t_2) are both present and may both contain the same free term variable, say z, but force different value assignments to z in the term environments to propagate a disrelation. For example, consider $z :: Either\ \tau_1\ \tau_2$. And consider t_1 employs $fromLeft_{\tau_2}\ z$ whereas t_2 employs $fromRight_{\tau_1}\ z$ to guarantee their semantic interpretation different from $\perp$ as might be essential to generate a counterexample (cf. Example 18). In such a case, we cannot find suitable entries for z in the term environments of the conclusion of (Arrow→*) because z needs to be interpreted as Left- and Right-value at the same time. The initial disrelation cannot be propagated and hence we need to abort the algorithm, returning without a counterexample.

To check whether we have to abort ExFind when merging term environments at the rule (Arrow→*), we create a history environment that tracks the usage of term variables and serves as primary information for an algorithm mergeEnv, described in Definition 29, that either tells that we have to abort ExFind because the term environments of the premises are incompatible, or returns successfully merged term environments that form a suitable result environment w.r.t. the conclusion of (Arrow→*). The history of a variable and the history environment are defined as follows.

Definition 27 (history)

A *history* is defined inductively via

$$h ::= \mathsf{Branch}\ x\ h \mid \mathsf{Split}\ x\ h\ h \mid \mathsf{Leaf}$$

where x ranges over the *history tags*

WrapTo	Head	PairTo	Proj	EitherTo
BottomTo′	ArrowTo*	Dist$_1$	Dist$_2$	BottomTo

The set of histories H is partially ordered by $\leqslant_H$, defined as

$$\begin{aligned}(\leqslant_H) = \mathit{lfp}(\lambda\mathcal{R}.\ &\{(\mathsf{Leaf}, h) \mid \forall h \in H\}\\ &\cup \{(\mathsf{Branch}\ x\ h, \mathsf{Branch}\ y\ h') \mid x = y \wedge (h, h') \in \mathcal{R}\}\\ &\cup \{(\mathsf{Split}\ x\ h_1\ h_2, \mathsf{Split}\ y\ h_1'\ h_2')\\ &\qquad \mid x = y \wedge (h_1, h_1') \in \mathcal{R} \wedge (h_2, h_2') \in \mathcal{R}\})\end{aligned}$$

Definition 28 (history environment)

Let Γ_V a term context. A map η that takes each term variable in Γ_V to a history is called *history environment* w.r.t. Γ_V.

Concerning the example stated above the last two definitions, the history environments for the two different premises would yield Branch Dist$_1$ h_1 and Branch Dist$_2$ h_2 for z, respectively, with h_1 and h_2 histories. We can tell incompatibility by the differing history tags Dist$_1$ and Dist$_2$. The naming of the tags is due to the rules they arise from, e.g. the (extended) rule (DIST$_1$) sets the history of $e :: \mathit{Either}\ \tau_1\ \tau_2$ in the conclusion of the rule to Branch Dist$_1$ h where h is the history of x in the premise of (DIST$_1$). The extension of the history tells that the values for e must be Left-values, because **fromLeft** is applied to them. Note that the same history tag is added by the rule (DIST$_1^\circ$), so the names of tags do match rule names only up to a $^\circ$-annotation. An exception to the naming convention is BottomTo′ which is not only introduced by (BOTTOM→′), but also by (APP′°). Whenever the value a term variable is mapped to is arbitrary up to fulfilling the requirements of a result environment, the variable has history Leaf. The precise construction of history environments is presented in Definition 30, along with the construction of result environment and disrelator.

The intention behind history tags in mind, we get a definition for the algorithm mergeEnv that checks compatibility of histories.

Definition 29 (mergeEnv)

Let σ_1' σ_2' term environments and η' history environment, each with domain V'. Let σ_1'', σ_2'' term environments and η'' history environment, each with domain V''. If for each $x \in V' \cap V''$ we have $\eta'(x) \leqslant_H \eta''(x)$ or $\eta''(x) \leqslant_H \eta'(x)$, then for $((\sigma_1', \sigma_2'), \eta')$ and $((\sigma_1', \sigma_2'), \eta')$ as input, the algorithm mergeEnv returns the environments (σ_1, σ_2) and η with domain $V = V' \cup V''$ and entries

$$\kappa(x) = \begin{cases} \kappa'(x) & \text{if } x \in V' \setminus V'' \vee \eta'' \leqslant_H \eta' \\ \kappa''(x) & \text{otherwise} \end{cases}$$

for $\kappa \in \{\sigma_1, \sigma_2, \eta\}$.

If not all $\eta'(x)$ and $\eta''(x)$ are comparable, then for the same input mergeEnv returns a failure.

The Formal Description of ExFind

Finally, we are ready for the precise definition of the construction of the result environment, the disrelator and the history environment. After we provided the construction, we explain the intention behind the different steps briefly.

DEFINITION 30 (ExFind)

The rules of TermFind, shown in Figures 4.8, 4.9 and 4.10, with (ARROW→′) in Figure 4.8 replaced by (ARROW→*), shown on page 94, and extended by the following constructions for result environments, disrelator and history environment, form the algorithm ExFind.

To describe the rules' extensions, we employ the names of types and variables from the rules' definitions, as shown in Figures 4.8, 4.9, 4.10 and on page 94 ((ARROW→*)) but we assume all newly introduced variable names to be fresh.

Furthermore, by $\mu \rightsquigarrow \mu'$ we denote the update of a (term or history) environment μ by an environment μ' with disjoint domain, such that μ is extended by the entries of μ' and afterward the environment's domain is reduced to the set of term variables appearing in the typing context of the current rule's conclusion. Except for the constructions for (ARROW→*), all environment names identify the environments of the premise, that are updated to gain suitable environments for the conclusion. For (ARROW→*) the result and history environments given to mergeEnv are indexed by the premises they arise from, i.e., by 1 or 2, and the environments that become updated are the ones returned by mergeEnv. By σ_i we generalize over the environments σ_1 and σ_2. Furthermore, updates where we only remove entries from the environments are omitted.

The update of a disrelator is denoted by $\varpi \to \varpi'$, meaning that employing the disrelator ϖ of the premise, we create the disrelator ϖ' for the conclusion of a rule. For the rule (ARROW→*) the disrelator of the first premise (that is not a pointedness check) is denoted by ϖ_1.

(VAR°)	σ_i	$=$	$\{z \mapsto \mathbf{p}^{+}_{\tau'} \mid (z :: \tau') \in \Gamma\}$
(VAR′)		$\cup$	$\{z \mapsto \mathbf{p}^{+}_{\tau'} \mid (z^{\circ} :: \tau') \in \Gamma\} \cup \{x \mapsto \mathbf{p}^{+}_{\tau}\}$
	ϖ	$=$	$(\tau, [\,])$
	η	$=$	$\{z \mapsto \mathsf{Leaf} \mid (z :: \tau') \in \Gamma\}$
		$\cup$	$\{z \mapsto \mathsf{Leaf} \mid (z^{\circ} :: \tau') \in \Gamma\} \cup \{x \mapsto \mathsf{Leaf}\}$
(NAT′)	σ_i	$=$	$\{z \mapsto \mathbf{p}^{+}_{\tau'} \mid (z :: \tau') \in \Gamma\}$
(UNIT′)		$\cup$	$\{z \mapsto \mathbf{p}^{+}_{\tau'} \mid (z^{\circ} :: \tau') \in \Gamma\}$
(LIST′)	ϖ	$=$	$(\tau, [\,])$
(PAIR′)	η	$=$	$\{z \mapsto \mathsf{Leaf} \mid (z :: \tau') \in \Gamma\}$
(EITHER′)		$\cup$	$\{z \mapsto \mathsf{Leaf} \mid (z^{\circ} :: \tau') \in \Gamma\}$
(BOTTOM→°)	σ_i	$\rightsquigarrow$	$\{f \mapsto \lambda \mathbf{x}.\sigma_i(y)\}$
(BOTTOM→)	η	$\rightsquigarrow$	$\{f \mapsto \mathsf{Branch\ BottomTo}\ \eta(y)\}$
(APP′°)	σ_i	$\rightsquigarrow$	$\{f \mapsto \mathbf{h_i}\}$
(BOTTOM→′)			with $(\mathbf{h_1}, \mathbf{h_2}) = (\mathbf{g_1}(\sigma_1(y)), \mathbf{g_2}(\sigma_2(y)))^{\Gamma,\tau_1}_{(\tau_1,[\,])}$
	η	$\rightsquigarrow$	$\{f \mapsto \mathsf{Branch\ BottomTo'}\ \eta(y)\}$
(NAT°)	σ_i	$\rightsquigarrow$	$\{x \mapsto \mathbf{p}^{+}_{Nat}\}$
(NDROP)	η	$\rightsquigarrow$	$\{x \mapsto \mathsf{Leaf}\}$
(UNIT°)	σ_i	$\rightsquigarrow$	$\{x \mapsto \mathbf{p}^{+}_{()}\}$
(UDROP)	η	$\rightsquigarrow$	$\{x \mapsto \mathsf{Leaf}\}$
(LIST°)	σ_i	$\rightsquigarrow$	$\{l \mapsto \mathbf{p}^{+}_{[\tau_1]}\}$
	η	$\rightsquigarrow$	$\{l \mapsto \mathsf{Leaf}\}$
(PAIR°)	σ_i	$\rightsquigarrow$	$\{p \mapsto \mathbf{p}^{+}_{(\tau_1,\tau_2)}\}$
	η	$\rightsquigarrow$	$\{p \mapsto \mathsf{Leaf}\}$
(EITHER°)	σ_i	$\rightsquigarrow$	$\{e \rightarrow \mathbf{p}^{+}_{Either\ \tau_1\ \tau_2}\}$
	η	$\rightsquigarrow$	$\{e \mapsto \mathsf{Leaf}\}$
(HEAD°)	σ_i	$\rightsquigarrow$	$\{l \mapsto [\sigma_i(h)]\}$
(HEAD)	η	$\rightsquigarrow$	$\{l \mapsto \mathsf{Branch\ Head}\ \eta(h)\}$
(PROJ°)	σ_i	$\rightsquigarrow$	$\{p \mapsto (\sigma_i(x), \sigma_i(y))\}$
	η	$\rightsquigarrow$	$\{p \mapsto \mathsf{Split\ Proj}\ \eta(x)\ \eta(y)\}$
(DIST$_1^{\circ}$)	σ_i	$\rightsquigarrow$	$\{e \mapsto \mathrm{Left}\ \sigma_i(x)\}$
(DIST$_1$)	η	$\rightsquigarrow$	$\{e \mapsto \mathsf{Branch\ Dist_1}\ \eta(x)\}$
(DIST$_2^{\circ}$)	σ_i	$\rightsquigarrow$	$\{e \mapsto \mathrm{Right}\ \sigma_i(x)\}$
(DIST$_2$)	η	$\rightsquigarrow$	$\{e \mapsto \mathsf{Branch\ Dist_2}\ \eta(x)\}$
(BOTTOM)	σ_i	$=$	$\{z \mapsto \mathbf{p}^{+}_{\tau'} \mid (z :: \tau') \in \Gamma\}$
	ϖ	$=$	$(\tau, [\,])$
	η	$=$	$\{z \mapsto \mathsf{Leaf} \mid (z :: \tau') \in \Gamma\}$

(UPDROP)
$$\begin{array}{lll} \sigma_i & \rightsquigarrow & \{x \mapsto \mathbf{p}^{+}_{\tau_1}\} \\ \eta & \rightsquigarrow & \{x \mapsto \mathsf{Leaf}\} \end{array}$$

(WRAP→′)
$$\begin{array}{lll} \sigma_i & \rightsquigarrow & \{f \mapsto \boldsymbol{\lambda}\mathbf{l}.\mathbf{case}\ \mathbf{l}\ \mathbf{of}\ \{[\mathbf{x}] \to \sigma_i(g)\ \mathbf{x}\}\} \\ \eta & \rightsquigarrow & \{f \mapsto \mathsf{Branch\ WrapTo}\ \eta(g)\} \end{array}$$

(PAIR→)
$$\begin{array}{lll} \sigma_i & \rightsquigarrow & \{f \mapsto \boldsymbol{\lambda}\mathbf{p}.\mathbf{case}\ \mathbf{p}\ \mathbf{of}\ \{(\mathbf{x},\mathbf{y}) \to \sigma_i(g)\ \mathbf{x}\ \mathbf{y}\}\} \\ \eta & \rightsquigarrow & \{f \mapsto \mathsf{Branch\ PairTo}\ \eta(g)\} \end{array}$$

(PROJ)
$$\begin{array}{lll} \sigma_i & \rightsquigarrow & \{p \mapsto (\sigma_i(x),\sigma_i(y))\} \\ \eta & \rightsquigarrow & \{p \mapsto \mathsf{Split\ Proj}\ \eta(x)\ \eta(y)\} \end{array}$$

(ABS)
$$(\tau_\varpi, l) \to (\tau_\varpi, (\boldsymbol{\lambda}\mathbf{u}.\mathbf{u}\ (\sigma_1(x)), \boldsymbol{\lambda}\mathbf{u}.\mathbf{u}\ (\sigma_2(x))) : l)$$

(WRAP)
$$(\tau_\varpi, l) \to (\tau_\varpi, (\mathbf{head}, \mathbf{head}) : l)$$

(EITHER→)
$$\begin{array}{lll} \sigma_i & \rightsquigarrow & \{f \mapsto \boldsymbol{\lambda}\mathbf{e}.\mathbf{case}\ \mathbf{e}\ \mathbf{of} \\ & & \quad \{\mathrm{Left}\ \ \mathbf{x} \to \sigma_i(g)\ \mathbf{x}; \mathrm{Right}\ \ \mathbf{x} \to \sigma_i(h)\ \mathbf{x}\}\} \\ \eta & \rightsquigarrow & \{f \mapsto \mathsf{Split\ EitherTo}\ \eta(g)\ \eta(h)\} \end{array}$$

(PAIR$_1$)
$$(\tau_\varpi, l) \to (\tau_\varpi, (\mathbf{fst}, \mathbf{fst}) : l)$$

(PAIR$_2$)
$$(\tau_\varpi, l) \to (\tau_\varpi, (\mathbf{snd}, \mathbf{snd}) : l)$$

(LEFT)
$$(\tau_\varpi, l) \to (\tau_\varpi, (\mathbf{fromLeft}, \mathbf{fromLeft}) : l)$$

(RIGHT)
$$(\tau_\varpi, l) \to (\tau_\varpi, (\mathbf{fromRight}, \mathbf{fromRight}) : l)$$

(ARROW→*)

If $\mathsf{mergeEnv}\ (\xi_1,\eta_1)\ (\xi_2,\eta_2)$ returns $((\sigma_1,\sigma_2),\eta)$ then
$$\begin{array}{lll} \sigma_i & \rightsquigarrow & \{f \mapsto \boldsymbol{\lambda}\mathbf{u}.\mathbf{h_i}\ (\mathbf{u}\ \sigma_i(w))\} \\ & & \text{with } (\mathbf{h_1},\mathbf{h_2}) = (\mathbf{g_1}(\sigma_1(y)), \mathbf{g_2}(\sigma_2(y)))^{\Gamma,\tau_2}_{(\varpi_1)} \\ \varpi & \to & (\tau, [\,]) \\ \eta & \rightsquigarrow & \{f \mapsto \mathsf{Split\ ArrowTo}^*\ \eta(x)\ \eta(y)\} \end{array}$$
otherwise abort ExFind.

The constructions in Definition 30 work similarly for most cases. In particular, if a new variable appears somewhere in the typing context and is not a replacement for some variable in the context of the rule's premise, e.g. as for the rule (UPDROP) or for all axioms, we only need to ensure that for each variable the term environment entry in σ_1 is related to the one in σ_2 and that the entries in σ_1 are not related to $\perp$ for $^\circ$-annotated variables. Since we choose plus-values as environment entries the conditions are fulfilled by Lemma 14 (1) and (2).

If a variable in the typing context arises as a substitute for a variable in the context of the rule's premise, we choose its value based on the term environment entries of the substituted variable. In such cases the history of a term variable comes in. Originally, newly created variables (that are not a substitute for another variable) have history Leaf. Whenever an entry for a new variable is generated using the entry of another variable, the way of generation is tracked in the variable's history. Given that explanation, the updates of the history environment are very natural. For example for (HEAD), the history of h in the premise, $\eta(h)$, is extended to Branch Head $\eta(h)$, pointing out that the generated term will employ the head of the list and that the values of that head in the different environments may be important to obtain a suitable counterexample to the parametricity theorem. While for variables of list type there is no (top level) way to conflict, the situation is different for variables of type $Either\ \tau_1\ \tau_2$. For them, as already mentioned, choices as Left- or Right-values may be essential. These choices are expressed via the history tags Dist_1 and Dist_2, respectively. The auxiliary algorithm mergeEnv (Definition 29) identifies the incompatibility between tags Dist_1 and Dist_2 and fails in such a conflicting case when called for the extra constructions of rule (ARROW$\rightarrow^*$). The constructions for (ARROW$\rightarrow^*$) also involve the second auxiliary algorithm, presented in Definition 26. Regarding (ARROW$\rightarrow^*$), the first premise tells that t_1 is not related to itself for a certain choice of related values for w, i.e., in the conclusion the semantics of $\lambda x :: \tau_1.t_1[x/w]$ (in the different environments) are not related if applied to the choices for w from the premise. In the extra construction for the entries of f in rule (ARROW$\rightarrow^*$), we design f such that its argument is applied to the values assigned to w in the environments of the first premise (that is not a pointedness restriction). This way, if f is applied to $\lambda x :: \tau_1.t_1[x/w]$, we generate the unrelated interpretations of t_1 from the first premise. Via the related functions $(\mathbf{h_1}, \mathbf{h_2})$ we guarantee to preserve the disrelation (see Definition 30 for the rule (ARROW$\rightarrow^*$) and Lemma 15).

The construction of $(\mathbf{h_1}, \mathbf{h_2})$ depends on the term t_1 via the disrelator of the first premise. The disrelator is initialized at the axioms and reinitialized for the rule (ARROW$\rightarrow^*$), telling that the parametricity theorem breaks because the constructed term's interpretations directly violate the conditions for relatedness. Here directly means to violate the definition for the current type, and this not by a violation for a (structural) subtype. Whenever the interpretations of the constructed term are unrelated essentially because requirements for a (structural) subtype are not met, the disrelator tells so. In the rule (HEAD) for example, the disrelator is updated and tells that the original disrelation arises on the elements of the list. Most importantly, for the rule (ABS) the disrelator tracks what related arguments must be given to a pair of functions to uncover that these functions are unrelated.

4.5.5 Correctness of ExFind

The following results verify the correctness of the constructions presented in Definition 30. That is, we prove that the constructed environments are result environments w.r.t. the given typing context and type, and that ExFind returns also a respective result term. We first give results for the second phase, and then extend them to the overall algorithm. The second phase constructs a strong result term, which is important for the correctness proof of the whole algorithm. We only state lemmas to structure a possible proof. The proofs of the respective lemmas proceed on the depth of the derivation tree and are found in Appendix A.1.

Lemma 16 For every internal input $\mathcal{I}^{int}$, the second phase of ExFind, if it returns a term, constructs a result environment w.r.t. $\mathcal{I}^{int}$.

Lemma 17 If, for an internal input $\mathcal{I}^{int}$, the second phase of ExFind returns a term, then it is a strong result term w.r.t. the input and the result environment returned with it.

Lemma 18 For all external inputs $\mathcal{I}^{ext}$

- the term environments (σ_1, σ_2) constructed by ExFind form a result environment w.r.t. $\mathcal{I}^{ext}$,
- the returned term t is a respective result term and
- the returned ϖ a disrelator for $([\![t]\!]^{\mathsf{fix}}_{\sigma_1}, [\![t]\!]^{\mathsf{fix}}_{\sigma_2})$.

By the definitions of result environment and result term, Lemma 18 implies that ExFind really creates counterexamples to the parametricity theorem. We highlight this fact by restating the lemma's assertion more explicitly via the following theorem.

Theorem 8 (correctness of ExFind) If ExFind returns a term t and a result environment $\xi = (\sigma_1, \sigma_2)$ for an external input $\mathcal{I}^{ext} = (\Gamma; \tau)$, then

$$\Gamma' \vdash t :: \tau$$

in $\lambda^{\alpha}_{\mathbf{fix}^+}$ where Γ' is Γ with all *-annotations removed, and

$$([\![t]\!]^{\mathsf{fix}}_{\sigma_1}, [\![t]\!]^{\mathsf{fix}}_{\sigma_2}) \notin \Delta^{\mathsf{fix}}_{\tau}$$

Additionally to the assertion of Theorem 8, we can extract all function arguments we may need when we derive a free theorem out of the parametricity theorem from the disrelator returned by ExFind. We will not separately state that result, for it follows immediately from the definition of a disrelator.

4.5.6 A Closer Look on Completeness

On the way from $\mathrm{TermFind}$ to ExFind we reduced the set of inputs for which a term is generated. Thereby, the completeness claim that the algorithm yields a term whenever a counterexample to the parametricity theorem (due to a certain missing strictness condition) exists, becomes unsustainable. The alterations to $\mathrm{TermFind}$ that reduce the possibilities to generate terms are the replacement of (Arrow→′) by (Arrow→*) and the mergeEnv check at (Arrow→*).

Unfortunately, the restriction (of (Arrow→′) to (Arrow→*)) prevents us from finding counterexamples for some types.

Example 21

Consider the following external input

$$(\alpha, \beta^*; ((Nat \to \beta) \to [\alpha \to \beta]) \to Nat)$$

It says that we test the strictness restriction for α, but take the one for β as given. $\mathrm{TermFind}$, employing (Arrow→′) in the derivation, yields the term

$$\begin{aligned} t = \lambda f :: (Nat \to \beta) \to [\alpha \to \beta].\\ &\mathbf{case}\ f\ (\lambda x :: Nat.(head_{\alpha\to\beta}\ ((\lambda z :: \beta.f\ (\lambda u :: Nat.z))\ \bot_\beta)\ \bot_\alpha)\\ &\quad \mathbf{of}\ \{[y] \to 0\}\end{aligned}$$

that simplifies to the semantically equivalent term

$$\begin{aligned} t = \lambda f :: (Nat \to \beta) \to [\alpha \to \beta].\\ &\mathbf{case}\ f\ (\lambda x :: Nat.(head_{\alpha\to\beta}\ (f\ \bot_{Nat\to\beta})\ \bot_\alpha))\ \mathbf{of}\ \{[y] \to 0\}\end{aligned}$$

We can choose minimal environments and the functions

$$\begin{aligned} f\ &= \lambda h :: Nat \to ().(\lambda x :: ().()) : (\mathbf{case}\ h\ 0\ \mathbf{of}\ \{() \to []\})\\ f' &= \lambda h :: Nat \to ().(\lambda x :: ().\mathbf{case}\ x\ \mathbf{of}\ \{() \to ()\})\\ &\qquad\qquad : (\mathbf{case}\ h\ 0\ \mathbf{of}\ \{() \to []\})\end{aligned}$$

and get $([\![f]\!]^{\mathsf{fix}}_\emptyset, [\![f']\!]^{\mathsf{fix}}_\emptyset) \in \Delta^{\mathsf{fix}}_{(Nat\to\beta)\to[\alpha\to\beta]}$, as well as $[\![t\ f]\!]^{\mathsf{fix}}_\emptyset = 0$ and $[\![t\ f']\!]^{\mathsf{fix}}_\emptyset = \bot$, i.e., we have a counterexample to the free theorem and thus to the parametricity theorem for type $((Nat \to \beta) \to [\alpha \to \beta]) \to Nat$. The example is only found by $\mathrm{TermFind}$, but not by ExFind.

Seidel and Voigtländer (2009b) wrongly count the above counterexample as a false positive of $\mathrm{TermFind}$, excluded via the restriction to (Arrow→*).[17] At the moment, no false positive generated by $\mathrm{TermFind}$ and excluded in ExFind due to the restriction of (Arrow→′) to (Arrow→*) is known. Hence, we argue that the restriction might not be necessary, but omitting it causes at least technical difficulties. Further investigation might be a topic for future research.

[17] Seidel and Voigtländer (2010, Section 6) do not discuss the example, but (wrongly?) assert that a false positive exists.

The compatibility check via mergeEnv indeed rules out false positives.

Example 22 Consider the external input

$$(\alpha, \beta^*, \gamma^*, \delta^*; (\gamma \to \mathit{Either}\ (\alpha \to \beta)\ (\gamma \to \delta)) \to ((\mathit{Nat} \to \beta) \to \gamma) \to \delta)$$

TermFind generates term

$$t = \lambda f :: \gamma \to \mathit{Either}\ (\alpha \to \beta)\ (\gamma \to \delta).\lambda g :: (\mathit{Nat} \to \beta) \to \gamma.$$
$$\mathit{fromRight}_{\gamma \to \delta}\ (f\ (g\ (\lambda x :: \mathit{Nat}.(\mathit{fromLeft}_{\alpha \to \beta}\ (f\ \bot_\gamma)\ \bot_\alpha))))\ \bot_\gamma$$

We will not find type interpretations and $(\mathbf{f}, \mathbf{f}') \in \Delta^{\mathbf{fix}}_{\gamma \to \mathit{Either}\ (\alpha \to \beta)\ (\gamma \to \delta), \rho}$ such that we gain a counterexample. We prove that no suitable pair $(\mathbf{f}, \mathbf{f}')$ exists by a case distinction on the possible behaviors of the two functions on $\bot$. In particular, we consider the following three cases

(a) $\mathbf{f}\ \bot = \bot$ and $\mathbf{f}'\ \bot = \bot$

(b) $\mathbf{f}\ \bot = \mathrm{Left}\ \mathbf{x}$ and $\mathbf{f}'\ \bot = \mathrm{Left}\ \mathbf{y}$ with $(\mathbf{x}, \mathbf{y}) \in \Delta^{\mathbf{fix}}_{\alpha \to \beta, \rho}$

(c) $\mathbf{f}\ \bot = \mathrm{Right}\ \mathbf{x}$ and $\mathbf{f}'\ \bot = \mathrm{Right}\ \mathbf{y}$ with $(\mathbf{x}, \mathbf{y}) \in \Delta^{\mathbf{fix}}_{\gamma \to \delta, \rho}$

Different behavior of $\mathbf{f}$ and $\mathbf{f}'$ on $\bot$ than described by the above cases immediately implies $(\mathbf{f}, \mathbf{f}') \notin \Delta^{\mathbf{fix}}_{\gamma \to \mathit{Either}\ (\alpha \to \beta)\ (\gamma \to \delta), \rho}$ (independently of the interpretation of type variables).

We consider case *(a)* first and get

$$([\![t]\!]^{\mathbf{fix}}_{\emptyset}\ \mathbf{f}, [\![t]\!]^{\mathbf{fix}}_{\emptyset}\ \mathbf{f}') = (\lambda \mathbf{g}.\mathbf{fromRight}\ (\mathbf{f}\ (\mathbf{g}\ \bot))\ \bot, \lambda \mathbf{g}.\mathbf{fromRight}\ (\mathbf{f}'\ (\mathbf{g}\ \bot))\ \bot)$$

The pair is in $\Delta^{\mathbf{fix}}_{((\mathit{Nat} \to \beta) \to \gamma) \to \delta, \rho}$ independently of the concrete ρ: The $\bot$s given to $\mathbf{g}$ as argument are related because $\mathit{Nat} \to \beta$ is pointed, and hence the results of $\mathbf{g}\ \bot$ for related choices for $\mathbf{g}$ are related. Now, $\mathbf{f}$ and $\mathbf{f}'$ are related, and thus relatedness propagates. Equally the application of $\mathbf{fromRight}$, as it is related to itself, preserves relatedness. Consequently, we cannot break the parametricity theorem.

For case *(b)* continuity of functions forces $\mathbf{f}$ and $\mathbf{f}'$ to yield Left -values for every input. Hence, the application of $\mathbf{fromRight}$ yields $\bot$ in $[\![t]\!]^{\mathbf{fix}}_{\emptyset}\ \mathbf{f}$ and $[\![t]\!]^{\mathbf{fix}}_{\emptyset}\ \mathbf{f}'$, respectively, and we get $([\![t]\!]^{\mathbf{fix}}_{\emptyset}\ \mathbf{f}, [\![t]\!]^{\mathbf{fix}}_{\emptyset}\ \mathbf{f}') = (\bot, \bot) \in \Delta^{\mathbf{fix}}_{((\mathit{Nat} \to \beta) \to \gamma) \to \delta, \rho}$.

Case *(c)* is similar to case *(b)* with left and right switched.

Unfortunately, a term generated by TermFind sometimes does not give rise to a counterexample even if a slight change to it allows for such an example. In such situations ExFind cannot yield a counterexample. We point out two reasons for such situations and illustrate them by slight changes to Example 22.

One reason that the terms generated by $\mathrm{TermFind}$ are sometimes not suitable for counterexample generation is that $\mathrm{TermFind}$ injects $\bot_\tau$ into the term it constructs, even if other choices are possible. For example it uses $\bot_{Nat}$ where it could introduce 0, 1, or alike terms. Thereby, $\mathrm{TermFind}$ may fail to produce a suitable term, as the following example shows.

EXAMPLE 23

Consider Example 22 with γ replaced by Nat, i.e., the input

$$\begin{aligned}&(\alpha, \beta^*, \delta^*;\\ &\quad (Nat \to Either\ (\alpha \to \beta)\ (Nat \to \delta)) \to ((Nat \to \beta) \to Nat) \to \delta)\end{aligned}$$

Up to type annotations, $\mathrm{TermFind}$ generates the term from Example 22:

$$\begin{aligned}t = \lambda f :: Nat \to Either\ (\alpha \to \beta)\ (Nat \to \delta).\lambda g :: (Nat \to \beta) \to Nat.\\ fromRight_{Nat \to \delta}\qquad\qquad\qquad\qquad\qquad\\ (f\ (g\ (\lambda x :: Nat.(fromLeft_{\alpha \to \beta}\ (f\ \bot_{Nat})\ \bot_\alpha)))) \bot_{Nat}\end{aligned}$$

But if we replace $\bot_{Nat}$ by 0 we gain a counterexample. We can choose:

$$\begin{aligned}\mathbf{f} &= \lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \to \mathrm{Left}\ \lambda\mathbf{y}.(); \_ \to \mathrm{Right}\ \lambda\mathbf{y}.()\}\\ \mathbf{f}' &= \lambda\mathbf{x}.\mathbf{case\ x\ of}\ \{\mathbf{0} \to \mathrm{Left}\ \lambda\mathbf{y}.\mathbf{case\ y\ of}\ \{() \to ()\};\\ &\qquad\qquad\qquad \_ \to \mathrm{Right}\ \lambda\mathbf{y}.()\}\\ \mathbf{g} = \mathbf{g}' &= \lambda\mathbf{x}.\mathbf{case\ x}\ \bot\ \mathbf{of}\ \{() \to \mathbf{1}\}\end{aligned}$$

A second reason why the generation of suitable terms fails is the restriction to singleton lists. Lists with more than one element would allow to get multiple variables of element type in the typing context. In turn, we could merge more environments via mergeEnv because we could employ different variables in the different environments. The next example, again just a slight modification of Example 22, illustrates the advantage.

EXAMPLE 24

Consider the external input

$$(\alpha, \beta^*, \gamma^*, \delta^*; [\gamma \to Either\ (\alpha \to \beta)\ (\gamma \to \delta)] \to ((Nat \to \beta) \to \gamma) \to \delta)$$

$\mathrm{TermFind}$ generates the term

$$\begin{aligned}t = \lambda l :: [\tau].\lambda g :: (Nat \to \beta) \to \gamma.\qquad\qquad\\ fromRight_{\gamma \to \delta}\ (head_\tau\ l\qquad\qquad\\ (g\ (\lambda x :: Nat.fromLeft_{\alpha \to \beta}\ (head_\tau\ l\ \bot_\gamma)\ \bot_\alpha))) \bot_\gamma\end{aligned}$$

where $\tau = \gamma \to Either\ (\alpha \to \beta)\ (\gamma \to \delta)$.

Compared to Example 22 we essentially replaced f by $head_\tau\ l$ in the term and, hence, the term is still not suitable for counterexample generation. But, we can replace one appearance of $head_\tau\ l$ by $head_\tau\ (tail_\tau\ l)$, where

$tail_\tau = \lambda xs :: [\tau].\mathbf{case}\ xs\ \mathbf{of}\ \{[\,] \to \bot_{[\tau]}; (y : ys) \to ys\}$, to allow for a counterexample. The manipulated term

$$\begin{aligned} t = \lambda l :: [\tau].\lambda g :: (Nat \to \beta) \to \gamma. \\ &fromRight_{\gamma\to\delta}\ (head_\tau\ (tail_\tau\ l) \\ &\quad (g\ (\lambda x :: Nat.fromLeft_{\alpha\to\beta}\ (head_\tau\ l\ \bot_\gamma)\ \bot_\alpha)))\ \bot_\gamma \end{aligned}$$

minimal environments and

$$\begin{aligned} \mathbf{l} &= [\boldsymbol{\lambda}\mathbf{x}.\mathrm{Left}\ \ (\boldsymbol{\lambda}\mathbf{y}.()), \boldsymbol{\lambda}\mathbf{x}.\mathrm{Right}\ \ (\boldsymbol{\lambda}\mathbf{y}.\mathbf{case\ x\ of}\ \{() \to ()\})] \\ \mathbf{l}' &= [\boldsymbol{\lambda}\mathbf{x}.\mathrm{Left}\ \ (\boldsymbol{\lambda}\mathbf{y}.\mathbf{case\ y\ of}\ \{() \to ()\}) \\ &\quad , \boldsymbol{\lambda}\mathbf{x}.\mathrm{Right}\ \ (\boldsymbol{\lambda}\mathbf{y}.\mathbf{case\ x\ of}\ \{() \to ()\})] \\ \mathbf{g} = \mathbf{g}' &= \boldsymbol{\lambda}\mathbf{h}.\mathbf{case\ h\ 0\ of}\ \{() \to ()\} \end{aligned}$$

establish such an example.

Summing up the results of the completeness discussion, we encountered three different reasons why we lose counterexamples in ExFind where TermFind yields a term:

- the restriction of (ARROW→') to (ARROW→*),
- the introduction of $\bot_\tau$ for types where terms that evaluate to non-$\bot$ values are available and
- the lack of lists with multiple elements.

All three problems only appear if the (ARROW→') rule is applied in TermFind. Hence, since (ARROW→') is only needed in TermFind if (ARROW→*) is applied in ExFind, we claim (based on the completeness claim for TermFind) that whenever ExFind returns without a counterexample for a given input $\mathcal{I}^{ext}$ and never employs (ARROW→*) during the counterexample search, then there exists no counterexample for the input $\mathcal{I}^{ext}$.

4.5.7 The Implementation of ExFind

ExFind is implemented and can be used via a web interface at http://www-ps.iai.uni-bonn.de/cgi-bin/exfind.cgi . The source code of the implementation is available for download at Hackage: http://hackage.haskell.org/package/free-theorems-counterexamples-0.3.1.0 . Figure 4.14 shows the web interface with the output for type $[\alpha] \to [\alpha]$. First, the sufficiently restricted free theorem is presented and afterward the necessity of a strictness condition is shown. Therefore, a counterexample to the insufficiently restricted theorem is presented the following way:

- The term f that gives rise to the counterexample is shown.
- The choice of the minimal environments is given.
- The term f is employed in a context that discloses the breach of the logical relation.

The context is built from the disrelator and comprises the arguments given to f to disprove the insufficiently restricted free theorem.

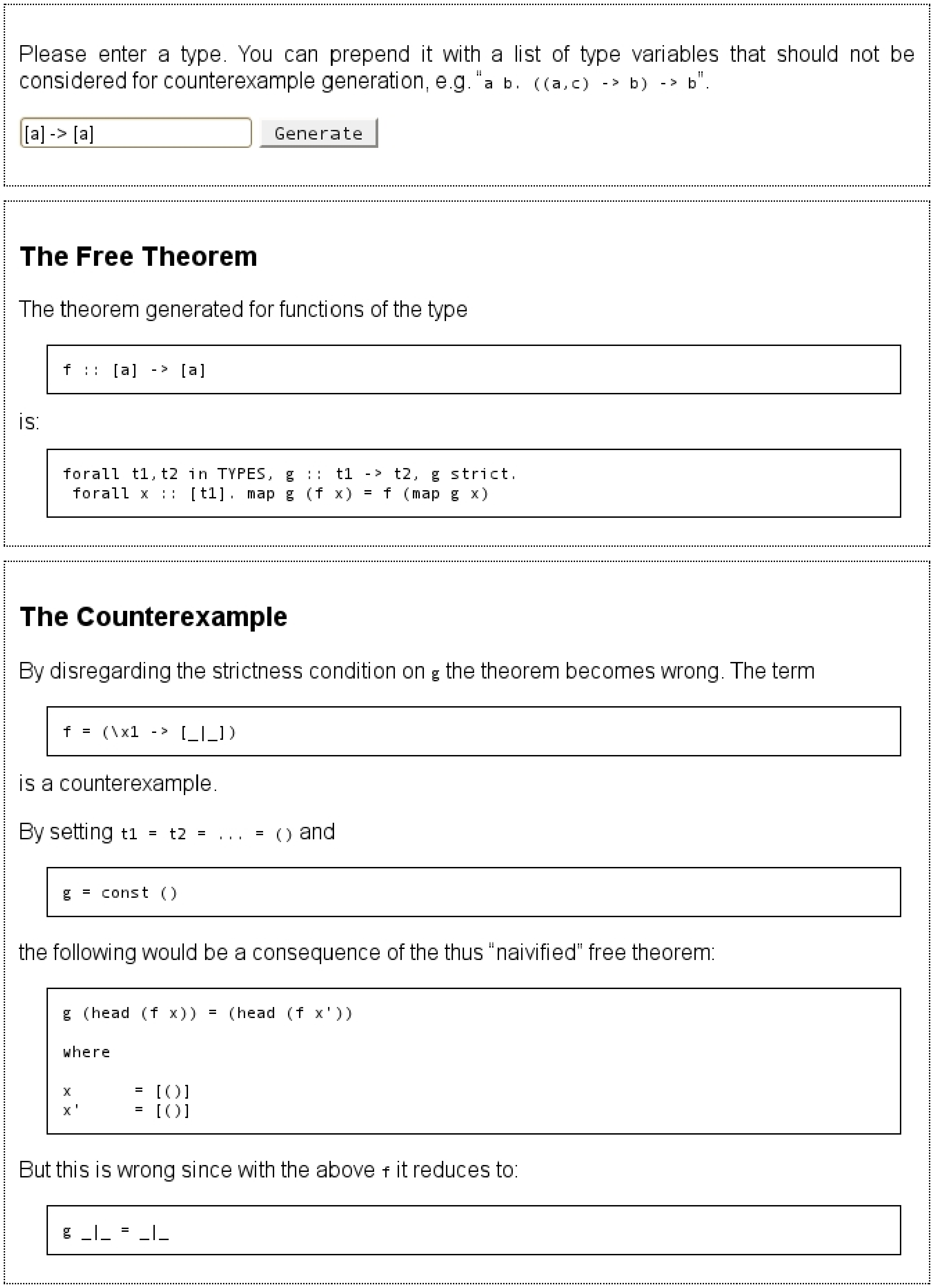
Regarding examples with more than one type variable, we can restrict the counterexample search to examples that arise due to a specific strictness restriction. Therefor, we only prepend type variables we do *not* want to investigate the strictness condition for. For example, for the input $\alpha\ \beta.((\alpha,\gamma) \to \beta) \to \beta$ only the strictness condition on the relation chosen for γ is considered.

4.6 Summary

Our aim was to set up an algorithm that checks for a given type τ if certain strictness conditions imposed on the free theorem for τ in the calculus $\lambda^{\alpha}_{\mathbf{fix}^+}$ are necessary or superfluous. In the end, we designed two algorithms, $\mathrm{TermFind}$ and ExFind.

$\mathrm{TermFind}$ is meant to construct terms that are likely to break the free theorem for the given type if one of the considered strictness conditions (given via the unannotated type variables in the typing context) is missing. For the design of $\mathrm{TermFind}$ we employed an alternative system of typing rules for $\lambda^{\alpha}_{\mathbf{fix}^+}$ without **fix**, because we found the original one unsuitable w.r.t. its algorithmic properties. The alternative rule system is a translation of a sequent calculus for proof search in intuitionistic logic that was established by Corbineau (2004). Translation of the rules yields a term search algorithm for **fix**-free terms that is complete in the way that it generates a term for every inhabited type.

In order to construct terms suitable for the generation of counterexamples to free theorems, adjustments to the rule system translated from Corbineau (2004) were necessary. Essentially, we searched for possibilities to introduce **fix** in the constructed term in a way that it enforces one of the strictness conditions in question. To (roughly) figure out where to insert **fix**, we employed the refined type system of Launchbury and Paterson (1996). The adaptation of the rule system translated from Corbineau (2004) resulted in $\mathrm{TermFind}$ and was driven by a worst-case strategy: We injected **fix** wherever it enforced one of the strictness conditions under consideration locally (i.e., for the current type) and with the possibility to propagate the enforcement to the type that was given as input to the algorithm. For the resulting algorithm $\mathrm{TermFind}$ we claim completeness in the way that whenever it returns without a term for a given external input, the considered strictness conditions for the input type are superfluous. We base the claim on the completeness of the rule system by Corbineau (2004) and the way we translated and manipulated his system. $\mathrm{TermFind}$ is correct in the way that whenever it returns a term, the refined type system enforces at least one of the considered strictness conditions. That notion of correctness does not imply that all terms found by $\mathrm{TermFind}$ are really suitable for counterexample generation, as exemplified in Subsection 4.5.6. Furthermore, we proved termination of $\mathrm{TermFind}$ for all external inputs.



Please enter a type. You can prepend it with a list of type variables that should not be considered for counterexample generation, e.g. "`a b. ((a,c) -> b) -> b`".

[a] -> [a] Generate

The Free Theorem

The theorem generated for functions of the type

```
f :: [a] -> [a]
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
 forall x :: [t1]. map g (f x) = f (map g x)
```

The Counterexample

By disregarding the strictness condition on `g` the theorem becomes wrong. The term

```
f = (\x1 -> [_|_])
```

is a counterexample.

By setting `t1 = t2 = ... = ()` and

```
g = const ()
```

the following would be a consequence of the thus "naivified" free theorem:

```
g (head (f x)) = (head (f x'))

where

x      = [()]
x'     = [()]
```

But this is wrong since with the above `f` it reduces to:

```
g _|_ = _|_
```

Figure 4.14: The output of the web interface for ExFind

The algorithm ExFind produces complete counterexamples to free theorems. It is proved correct and terminates for every external input. ExFind mainly extends the rules of TermFind by extra constructions sufficient to establish complete counterexamples. Unfortunately, to manage all extra constructions, the rule (ARROW$\rightarrow'$) from TermFind is simplified to (ARROW$\rightarrow^*$) and additionally ExFind aborts under certain conditions at rule (ARROW$\rightarrow^*$) because it cannot always construct suitable term environments. The changes render ExFind provably incomplete, but at least the second change is necessary to exclude false positives, and thus to get a (provably) correct algorithm.

ExFind is implemented and can be used online via a web interface at http://www-ps.iai.uni-bonn.de/cgi-bin/exfind.cgi . The source code is available at: http://hackage.haskell.org/package/free-theorems-counterexamples-0.3.1.0 .

4.7 Outlook

Concerning counterexample generation to free theorems with missing strictness restrictions in a calculus with general recursion, the generator ExFind may be improved in several ways:

- We can try to replace (ARROW$\rightarrow^*$) by the rule (ARROW$\rightarrow'$) enriched with the respective extra constructions as described in Definition 30. The primary problem will be the generation of suitable term environment entries for f in the conclusion of (ARROW$\rightarrow^*$).
- We can distinguish types with differently many values, such that we can use different, differently and incomparably defined, inputs for a function if it is called several times. The new feature will facilitate examples like Example 23. But, the distinction of the different values will complicate the generation of term environments.
- We may add a special treatment for variables in the typing context that represent elements from a list, because of these we have an arbitrary number of independent copies present, and hence can treat them differently w.r.t. merging of environments. In particular, we could give different copies to the premises of (ARROW$\rightarrow^*$). The change would enable us to generate the counterexample in Example 24.

The second and the third suggestion for enhancement of ExFind imply that ExFind must create a slightly different term than TermFind does. It is not a compulsory enhancement, if we force TermFind to create also different terms. It does not build complete (or, more precisely, completely instantiated) counterexamples anyway, and thus it suffices if just *a* term is found, when a counterexample exists. But still, concerning TermFind there are reasonable goals for further investigations. First, we may try to prove that TermFind is complete, and second we may consider the gap between (the possibly improved version of) ExFind and TermFind a bit closer. Example 22 suggests that TermFind is incorrect. It returns a term for an input that does not serve as the base for a counterexample and also slight modifications, as suggested as improvements for ExFind, do not allow for a counterexample. For the tested input, there seems to exist no counterexample to the respective free theorem and hence TermFind seems to be incorrect in the way that it finds terms for inputs that do not allow for a counterexample.

The "perfect", but seemingly difficult or unachievable result we could head for is to close the gap between TermFind and ExFind, and establish a complete and correct counterexample generator.

Aside of testing the necessity of strictness restrictions in a calculus with general recursion, we could also examine a counterexample generator that exemplifies the necessity of totality restrictions in a calculus with selective strictness. For that investigation the refined type system developed in the next chapter might be employed.

Chapter 5

Taming Selective Strictness

[1] As already mentioned (cf. Section 1.2 and 2.3.1), Haskell provides the ability to enforce strict evaluation, e.g. via the strictness primitive **seq** that is powerful enough to simulate all other strictness annotations available in Haskell. To illustrate the usefulness and the problems of selective strictness we present four different versions of the Haskell function $foldl$. All of them take a binary function c, an initial argument n and a list as arguments (all appropriately typed). The functions consume the list from left to right, calculating $(c\ (\ldots (c\ n\ x_1) \ldots)\ x_n)$ for c, n and $[x_1, \ldots, x_n]$ as input, return n for the empty list as input and $\bot$ for lists of the form $x_1 : \ldots : x_n : \bot$ as well as for infinite lists (where for infinite lists depending on the version of $foldl$ and the definedness of its arguments, the calculation either yields a finite failure or does not terminate). Figure 5.1 illustrates how $foldl$ transforms a list.[2]

Haskell provides two versions of $foldl$, the standard function $foldl$ and a strict version $foldl'$[3]. We show possible implementations in the upper row of Figure 5.2. Let us exemplify the benefit of strict evaluation by the implementation of a sum function.

Example 25
(benefit of strict evaluation)

Consider the functions

$$sum = foldl\ (+)\ 0 \qquad \text{and} \qquad sum' = foldl'\ (+)\ 0$$

Both sum up the elements of a list, but evaluation proceeds differently. For example, compare the evaluation of both functions when applied to $[1, 2]$:

[1]Results have been published in Seidel and Voigtländer (2009a, 2011a) and are also found in the technical report Seidel and Voigtländer (2009c).

[2]The illustration fits all versions of $foldl$ we consider during this chapter, as long as all arguments are total. When failure comes in, the functions sometimes differ in the definedness of their results.

[3]The function $foldl'$ is found in the module `Data.List`, while $foldl$ is present via the `Prelude`.

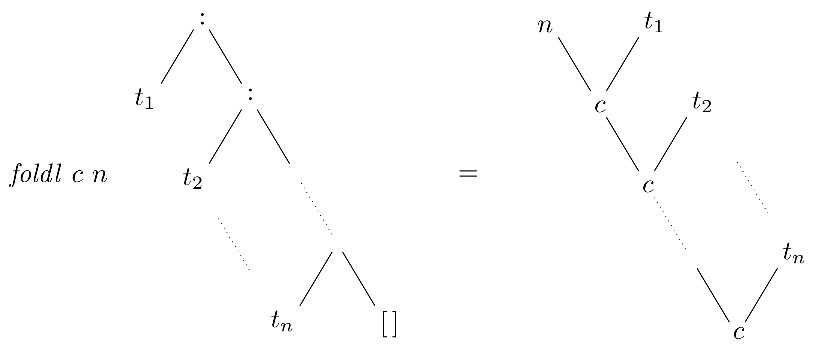


Figure 5.1: Visualization of the function *foldl* (for lists with total spine, i.e., of form $x_1 : \ldots : x_n : []$)

$$\begin{array}{llll}
& \mathit{sum}\ [1,2] & & \mathit{sum}'\ [1,2] \\
\equiv & \mathit{foldl}\ (+)\ 0\ [1,2] & \equiv & \mathit{foldl}'\ (+)\ 0\ [1,2] \\
& & \equiv & \mathbf{let}\ n' = 0+1\ \mathbf{in} \\
& & & \quad \mathbf{seq}\ n'\ (\mathit{foldl}'\ (+)\ n'\ [2]) \\
\equiv & \mathit{foldl}\ (+)\ (0+1)\ [2] & \equiv & \mathit{foldl}'\ (+)\ 1\ [2] \\
& & \equiv & \mathbf{let}\ n' = 1+2\ \mathbf{in} \\
& & & \quad \mathbf{seq}\ n'\ (\mathit{foldl}'\ (+)\ n'\ []) \\
\equiv & \mathit{foldl}\ (+)\ ((0+1)+2)\ [] & \equiv & \mathit{foldl}'\ (+)\ 3\ [] \\
\equiv & (0+1)+2 & & \\
\equiv & 1+2 & & \\
\equiv & 3 & \equiv & 3
\end{array}$$

We observe that *sum* does not evaluate the addition until all elements of the list have been consumed. Thus, the unevaluated sum must be stored until the list is completely consumed. Consequently we unnecessarily allocate memory linear in the list length. In contrast, sum' starts to calculate the sum immediately when two summands are available and thus needs only constant space to manage the addition. Hence, sum' is more efficient w.r.t. memory usage, and as a consequence very likely also w.r.t. runtime.

Example 25 shows the benefit of strict evaluation, but from Section 2.3 we already know its negative influence on parametricity results. In this chapter, we develop a refined type system to reduce the restrictions that forced strict evaluation enforces on the parametricity theorem and hence on free theorems. Section 5.1 provides motivation for refined typing, while Section 5.2 introduces a refined type system for $\lambda^{\alpha}_{\mathbf{seq}}$[4]. Finally, we aim for a retyping algorithm that takes a term with standard type annotations as in $\lambda^{\alpha}_{\mathbf{seq}}$ and returns the term with the best possible refined type annotations, as well as the best possible refined type(s) of that term. Here "best" refers to "yielding a minimally restricted

[4]The calculus $\lambda^{\alpha}_{\mathbf{seq}}$ is introduced in Section 2.3.

$$\begin{array}{l} \mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\ \mathit{foldl}\ c\ n\ [\,] \qquad = n \\ \mathit{foldl}\ c\ n\ (x : \mathit{xs}) = \mathit{foldl}\ c\ (c\ n\ x)\ \mathit{xs} \end{array}$$

$$\begin{array}{l} \mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\ \mathit{foldl}'\ c\ n\ [\,] \qquad = n \\ \mathit{foldl}'\ c\ n\ (x : \mathit{xs}) = \\ \quad \mathbf{let}\ n' = c\ n\ x\ \mathbf{in}\ \mathbf{seq}\ n'\ (\mathit{foldl}'\ c\ n'\ \mathit{xs}) \end{array}$$

$$\begin{array}{l} \mathit{foldl}'' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\ \mathit{foldl}''\ c\ n\ [\,] \qquad = \mathbf{seq}\ (c\ n)\ n \\ \mathit{foldl}''\ c\ n\ (x : \mathit{xs}) = \\ \quad \mathbf{seq}\ \mathit{xs}\ (\mathbf{seq}\ x\ (\mathit{foldl}''\ c\ (c\ n\ x)\ \mathit{xs})) \end{array}$$

$$\begin{array}{l} \mathit{foldl}''' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\ \mathit{foldl}'''\ c\ n\ [\,] \qquad = \mathbf{seq}\ c\ n \\ \mathit{foldl}'''\ c\ n\ (x : \mathit{xs}) = \mathbf{seq}\ c\ (\mathit{foldl}'''\ c\ (c\ n\ x)\ \mathit{xs}) \end{array}$$

Figure 5.2: The Haskell standard functions *foldl* and *foldl′*, and two more variants

parametricity theorem". Unfortunately, the refined type system presented in Section 5.2 is not suitable as a retyping algorithm. Neither can we guarantee termination nor can we directly use input with standard types when we want to employ the type systems' rules as type refinement algorithm. Therefore, in Section 5.3, we alter the typing rules such that we obtain an algorithmic rule system. Based on the obtained rule system a retyping algorithm is implemented. Section 5.4 describes how to use the implementation. In Section 5.5 we present a summary of the whole chapter, and Section 5.6 discusses directions for further research.

5.1 Motivation for a Refined Type System

Consider the following two snippets of Haskell code.

$$(+1) \circ (\mathit{foldl}\ (+)\ 0) \quad \text{and} \quad \mathit{foldl}\ (+)\ 1$$

The second snippet is an optimized version of the first one. The semantic equivalence of the two code snippets is an instance of a more general equivalence, called fusion property (Hutton, 1999). It says that for all appropriately typed f, c, c', n, xs, we have

$$(\forall x, y.\ f\ (c\ x\ y) \equiv c'\ (f\ x)\ y) \Rightarrow f\ (\mathit{foldl}\ c\ n\ \mathit{xs}) \equiv \mathit{foldl}\ c'\ (f\ n)\ \mathit{xs} \tag{5.1}$$

We can prove (5.1) inductively over the structure of xs, employing the definition of *foldl*, or gain it directly via the free theorem for *foldl*'s type.

The free theorem for *foldl*'s type in a calculus without the possibility to enforce strict evaluation (like $\lambda^{\alpha}_{\mathbf{fix}}$) states that for all appropriately typed strict f and g, and appropriately typed c, c', n, xs,

$$\begin{aligned} &(\forall x, y.\ f\ (c\ x\ y) \equiv c'\ (f\ x)\ (g\ y)) \\ &\qquad \Rightarrow f\ (\mathit{foldl}\ c\ n\ \mathit{xs}) \equiv \mathit{foldl}\ c'\ (f\ n)\ (\mathit{map}\ g\ \mathit{xs}) \end{aligned} \tag{5.2}$$

holds. Via refined typing à la Launchbury and Paterson (1996) we can also remove the strictness conditions on f and g; and if we choose $g = id$ we immediately get the fusion property (5.1).

But, how about $foldl'$, does (5.1) hold for it? Unfortunately, it does not. If we choose

$$\begin{aligned} f &= \lambda x \to \textbf{if } x \textbf{ then } True \textbf{ else } \bot & n &= False \\ c = c' &= \lambda x\ y \to \textbf{if } y \textbf{ then } True \textbf{ else } x & xs &= [False, True] \end{aligned}$$

then we obtain a counterexample to (5.1) with $foldl$ replaced by $foldl'$.[5] An attempt to prove (5.1) with $foldl$ replaced by $foldl'$ via the definition of $foldl'$ reveals a new sufficient extra condition to reestablish the fusion property: If we force f *total* then (5.1) holds for $foldl'$. Totality of a function means that for every argument different from $\bot$, the result is different from $\bot$. Or, put differently, the total functions are exactly the functions with a bottom-reflecting graph.

total function

Now, how about the proof of (5.1) for $foldl'$ via the free theorem? Since $foldl'$ employs **seq**, we have to choose a free theorem aware of selective strict evaluation. We already described the influence of selective strictness on relational parametricity in Section 2.3. Employing the parametricity theorem for a language with selective strictness (Theorem 3) we obtain (5.2), but with the extra conditions that

- f and g strict and total,
- $c \equiv \bot$ iff $c' \equiv \bot$,
- $c\ z \equiv \bot$ iff $c'\ (f\ z) \equiv \bot$, for all appropriately typed z.

Applying the resulting statement for a proof of the fusion property, we instantiate g by id, which is strict and total, and we require f to be total, as the inductive proof does. The strictness requirement on f can probably be dropped by refined typing à la Launchbury and Paterson (1996). But, the conditions on c and c', that are superfluous as we know from the inductive proof, remain. So, why do these restrictions arise and are they really necessary? Therefor, consider $foldl''$ and $foldl'''$, given in Figure 5.2 and basically equivalent to $foldl$ and $foldl'$ but with different applications of **seq**.

We regard the free theorem given in (5.2) and show by example the necessity of all additional restrictions that arise in the presence of **seq**. We choose the following three instantiations for the variables in (5.2):

$$f = id \quad g = t_1 \quad c = t_2 \quad c' = t_2 \quad n = True \quad xs = [False] \tag{5.3}$$
$$f = id \quad g = id \quad c = t_2 \quad c' = t_3 \quad n = False \quad xs = [\,] \tag{5.4}$$
$$f = id \quad g = id \quad c = \bot \quad c' = \lambda x \to \bot \quad n = False \quad xs = [\,] \tag{5.5}$$

[5] We obtain $True \equiv \bot$.

with

$$t_1 = \lambda x \rightarrow \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } \bot$$
$$t_2 = \lambda x\ y \rightarrow \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } \bot$$
$$t_3 = \lambda x \rightarrow \textbf{if } x \textbf{ then } (\lambda y \rightarrow \mathit{True}) \textbf{ else } \bot$$

For all instantiations the free theorem (5.2) holds for foldl and with foldl' instead of foldl. But, instantiations (5.3) and (5.4) break the theorem if we replace foldl by foldl''. Furthermore, (5.5) breaks it if we replace foldl by foldl'''. All three instantiations violate a different condition that arises through **seq**. For the first instantiation g is not total, for the second $c\ z \not\equiv \bot$ while $c'\ (f\ z) \equiv \bot$ for $z = \mathit{False}$, and for the last one $c \equiv \bot$ while $c' \not\equiv \bot$. Hence, we see that none of the extra conditions on (5.2) that arise in a selectively strict language can be dropped. We also observe that the necessity of a restriction seems to depend on the actual use of **seq**, i.e., not only on the fact that **seq** is employed, but crucially on the fact where it is employed. Our intuition will prove right: We show that the totality restriction on f arises from the strict evaluation of a term of type α, the totality restriction on g arises from the strict evaluation of a term of type β, and the other two restrictions arise from the strict evaluation of c and $c\ n$. Concerning the different uses of **seq** in the versions of foldl shown in Figure 5.2, only the strict evaluation of xs in the definition of foldl'' does not enforce a restriction. Strictly evaluating a list's structure is nothing we must employ **seq** for. We can simulate the same behavior via a case expression and thus **seq** has no negative effect on the free theorem when used on lists.

The fundamental problem when establishing stronger free theorems in a calculus with a strictness primitive is that (in the standard type system) the theorems cannot be sensitive to the actual applications of the strictness primitive because the applications are not reflected in the type of a term. For example, all versions of foldl given in Figure 5.2 have the same type. Since a free theorem only relies on the type of a function, obviously for all versions of foldl the free theorem must be identical. In particular, all restrictions that arise through the different uses of **seq** must be enforced by the free theorem. This situation is unsatisfactory. For example, consider that we get the fusion property for foldl directly as an instance of the free theorem of foldl's type in a calculus without selective strictness. But, in a calculus that allows selective strict evaluation, the free theorem is encumbered with several superfluous side conditions, the same for foldl' where only one extra condition is important. Hence, we would benefit if free theorems were to somehow incorporate knowledge about the use of strictness primitives. The only way to achieve such theorems is to enrich the type of a function by information about the employment of the strictness primitive. If we can guarantee that at certain places we do not evaluate strictly, we are able to avoid restrictions on free theorems that typically arise in a selectively strict language. The idea to use refined types to control the influence of a language feature on parametricity results stems from Launchbury and Paterson (1996). They develop a refined type system to track general recursion and thereby reduce strictness restrictions on free theorems that arise in a calculus with general recursion as presented in Section 2.2. We already used the original refinement in Chapter 4. Here, we realize the idea w.r.t. the influence of selective strict evaluation on parametricity results.

5.2 A Refined Type System

We develop a refined type system for $\lambda^{\alpha}_{\mathbf{seq}}$. The calculus $\lambda^{\alpha}_{\mathbf{seq}}$ can be seen as a core of Haskell that captures the (w.r.t. our investigations of free theorems) important features of selective strictness and general recursion. Before we start the development, let us point out that this work is not the first approach that focuses on the problems selective strictness causes w.r.t. parametricity results.

5.2.1 A Former Refinement Approach — The Type Class Eval

The influence of selective strict evaluation on parametricity results and hence on free theorems was noticed already when selective strict evaluation was introduced in Haskell. It was known that the polymorphically typed function $\mathbf{seq} :: \alpha \to \beta \to \beta$ does not satisfy the corresponding parametricity property and also functions using it may not. As a consequence, in Haskell 1.3 (Peterson et al., 1996), the first Haskell version supporting selective strict evaluation, it was not typed completely polymorphic, but constrained by a type class Eval: $\mathbf{seq} :: \mathsf{Eval}\ \alpha \Rightarrow \alpha \to \beta \to \beta$. Hence, terms had to be of some type in Eval to be strictly evaluated. The restriction was meant to prevent from additional side conditions regarding parametricity results, in particular it was intended to guarantee the optimization rule *foldr* / *build*, implemented in GHC and relying on parametricity, to remain correct (Hudak et al., 2007, Section 10.3).[6] But, the type class approach turned out to be improper.

Even with the type class restriction on **seq**, parametricity results were not completely restored, as observed by Johann and Voigtländer (2004, 2006). The Haskell language report for version 1.3 states that *"Functions as well as all other built-in types are in* Eval.*"* (Peterson et al., 1996, Section 6.2.7). As a consequence, function *foldl'''*, shown in Figure 5.2, would not require a single Eval restriction, even though it does not satisfy the "naive" free theorem that is not aware of selective strict evaluation. That means, with the type class approach some restrictions on free theorems that arise from the strict evaluation of functions are not tracked. The problem is that with only the Haskell type class approach (Wadler and Blott, 1989) we can hardly do better. If we try to distinguish function types in Eval from those not in Eval, the Haskell type class approach is insufficient. Type classes in Haskell constrain the types that we can instantiate for a type variable, and as a pay off, they guarantee certain operations to be available at those types. For Eval, the available operation is **seq**. Starting from the type class assertions for type variables, class membership of all other types

[6]The GHC allows to optimize code via explicitly given rewrite rules. Directed by such rules GHC replaces code snippets (Peyton Jones et al., 2001). The programmer writing the rule is responsible for its correctness, i.e., semantic equivalence of both code snippets. The module `Data.List` from the standard libraries implements several rewrite rules that are special cases of *foldr* / *build*.

must be derivable in a fixed manner. Thus, we need to read off only from the argument or the result type of a function if it is in Eval or not. Unfortunately, neither of the types has an influence on whether we allow strict evaluation of functions of a certain type or not. Hence, the type class approach is unsuitable. But also an extension that allows to set type class restrictions on function types directly does not solve the problem completely.[7] Consider the function $f :: \mathsf{Eval}\ (\alpha \to \mathit{Int}) \Rightarrow (\alpha \to \mathit{Int}) \to (\alpha \to \mathit{Int}) \to \mathit{Int}$ with the definition $f = \lambda g\ h \to \mathbf{seq}\ g\ 5$. Here the type class restriction tells that **seq** might be employed on g and h. Actually, it is not employed on h, but we cannot express that by removing a type class constraint, because then we would remove the necessary restriction on g. Hence, a type system that really allows fine-grained insights about the use of a strictness primitive must take another approach than only the Haskell type class system for refinement.

Just as a side note: The type class Eval was removed in Haskell 98 (Peyton Jones, 2003), making **seq** completely polymorphic. As Hudak et al. (2007) recall, the reason was rather practical: **seq** is usually introduced when optimizing an already complete program. In this scenario, the type class constraint might force a change of the type signature of the function that is optimized and thereby cause changes of type signatures all over the program.

5.2.2 The New Approach — An Annotated Type System

Before we design a new refined type system, we have to clarify what the type system has to express. Therefore, we consider the influence of selective strictness in detail. The parametricity theorems for $\lambda^{\alpha}_{\mathbf{fix}}$ (Theorem 2) and $\lambda^{\alpha}_{\mathbf{seq}}$ (Theorem 3), i.e., for calculi without/with a strictness primitive, differ only in the bottom-reflectingness requirement on the logical relation. Concerning this requirement two questions arise:

- How is bottom-reflectingness guaranteed?
- Where is bottom-reflectingness really used?

A closer look on the definitions of the logical relations for $\lambda^{\alpha}_{\mathbf{fix}}$ and $\lambda^{\alpha}_{\mathbf{seq}}$, as well as on Theorems 2 and 3, reveals the differences that suffice to guarantee bottom-reflectingness of the logical relation: Relations between functions have to be forced bottom-reflecting by definition and the range of ρ must also be restricted to bottom-reflecting relations.

Concerning the use of bottom-reflectingness, a closer look at the proof of Theorem 3 reveals that the property is only needed for the relational interpretation of a type if a term of that type is strictly evaluated, i.e., if a term is bound to a variable via a strict let expression, like t_1 of type τ_1 in $\mathbf{let!}\ x :: \tau_1 = t_1\ \mathbf{in}\ t$. Hence, if we distinguish types whose relational interpretation is guaranteed to be bottom-reflecting from those whose relational interpretation is not and if

[7]In Haskell 98 and Haskell 2010 we can only add class constraints to type variables, but not to function types. Nevertheless, GHC provides an extension that allows to constrain also other types. It is enabled via the flag `-XFlexibleContexts`.

we allow strict evaluation only for terms whose type's relational interpretation is guaranteed to be bottom-reflecting, then the parametricity theorem should still hold. To distinguish the different kinds of types, we set up the type class Seqable, containing only the types that guarantee a bottom-reflecting relational interpretation.

Seqable

By the just described insights, we see that the relevant restrictions to guarantee bottom-reflectingness are manifested only in additional conditions on function types and type variables. If we know guarantees about bottom-reflectingness of the relational interpretations of function types and type variables, we can determine all types for which the bottom-reflectingness guarantee holds via class membership rules for Seqable. Hence, we (only) need a way to distinguish function types and type variables that do allow to safely omit the extra bottom-reflectingness condition (because terms of these types are not allowed to be strictly evaluated) from those who do not.

annotated type system

In the words of Nielson and Nielson (1999), we set up an *annotated type system* over the underlying type system of $\lambda^{\alpha}_{\mathbf{seq}}$ to express the necessity of extra conditions. Annotations are made on the types where we may or may not omit extra conditions. For type variables we decide to annotate them either with ϵ if they allow for strict evaluation of terms of their type (and therefore force a bottom-reflectingness restriction), or by $\circ$, otherwise. The handling is in line with the introduction of the type class Eval in Haskell: type variables with an ϵ-annotation are exactly the ones declared to be in Eval via a type class constraint. For function types, we distinguish the annotated type constructors $\to^{\epsilon}$ and $\to^{\circ}$; where terms of type $\tau_1 \to^{\epsilon} \tau_2$ can be strictly evaluated, while terms of type $\tau_1 \to^{\circ} \tau_2$ cannot. The annotations on the function types allow to express for every single term of such types if we are allowed to evaluate it strictly, or not. They constitute the main difference of our approach to a refined type system compared to the approach via Eval. The distinction of two function types is essential to allow for minimally restrictive free theorems.

$\lambda^{\alpha}_{\mathbf{seq}*}$

The calculus $\lambda^{\alpha}_{\mathbf{seq}}$ with the annotated type system is called $\lambda^{\alpha}_{\mathbf{seq}*}$. Type syntax is given in Figure 5.3, term syntax has not changed compared to $\lambda^{\alpha}_{\mathbf{seq}}$ (except for type annotations). The refined typing rules (except the ones involving naturals that do not provide any extra insight) are shown in Figure 5.4, the class membership rules for Seqable in Figure 5.5. Typing contexts contain type variables that are annotated by either ϵ or $\circ$. The subtype relation that is described in the next paragraphs and stated in Figure 5.6 suggests to interpret the annotations as an ordered set.

Definition 31 (ordered set of annotations)

The annotations $\circ$ and ϵ form an ordered set $(\{\circ, \epsilon\}, \leqslant)$ with $\leqslant$ the reflexive closure of $\circ \leqslant \epsilon$.

From the discussion above, most typing and class membership rules should be clear: There are two function types, a Seqable-check is only performed by

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{type variable} \\ & \mid & \tau \rightarrow^{\epsilon} \tau & \text{function type whose terms can be strictly evaluated} \\ & \mid & \tau \rightarrow^{\circ} \tau & \text{function type whose terms must not be strictly evaluated} \\ & \mid & [\tau] & \text{list type} \end{array}$$

Figure 5.3: Annotated type syntax of $\lambda^{\alpha}_{\mathbf{seq}*}$, term syntax remains as for $\lambda^{\alpha}_{\mathbf{seq}}$ (Figure 2.11)

$$\Gamma, x :: \tau \vdash x :: \tau \ (\textsc{Var}) \qquad \Gamma \vdash [\,]_{\tau} :: [\tau] \ (\textsc{Nil})$$

$$\frac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ [\,] \rightarrow t_1; (x_1 : x_2) \rightarrow t_2) :: \tau}\ (\textsc{LCase})$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1 \rightarrow^{\nu} \tau_2}\ (\textsc{Abs}_{\nu})_{\nu \in \{\circ,\epsilon\}} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow^{\nu} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\ (\textsc{App}_{\nu})_{\nu \in \{\circ,\epsilon\}}$$

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]}\ (\textsc{Cons}) \qquad \frac{\Gamma \vdash t :: \tau_1 \qquad \tau_1 \preceq \tau_2}{\Gamma \vdash t :: \tau_2}\ (\textsc{Sub})$$

$$\frac{\Gamma \vdash t :: \tau \rightarrow^{\nu} \tau}{\Gamma \vdash \mathbf{fix}\ t :: \tau}\ (\textsc{Fix}_{\nu})_{\nu \in \{\circ,\epsilon\}} \qquad \frac{\Gamma \vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}\ (\textsc{SLet}')$$

Figure 5.4: Refined typing rules of $\lambda^{\alpha}_{\mathbf{seq}*}$

$$\Gamma_{\mathcal{T}} \vdash [\tau] \in \mathsf{Seqable}\ (\textsc{CS-List}) \qquad \Gamma_{\mathcal{T}} \vdash \tau_1 \rightarrow^{\epsilon} \tau_2 \in \mathsf{Seqable}\ (\textsc{CS-Arrow}) \qquad \frac{\alpha^{\epsilon} \in \mathsf{Seqable}}{\Gamma_{\mathcal{T}} \vdash \alpha \in \mathsf{Seqable}}\ (\textsc{CS-Var})$$

Figure 5.5: Class membership rules for Seqable in $\lambda^{\alpha}_{\mathbf{seq}*}$ and $\lambda^{\alpha}_{\mathbf{seq}+}$

the rule (SLet′), and the only types not in Seqable are type variables annotated by $\circ$ in the typing context and function types constructed by $\rightarrow^{\circ}$. Note that some typing rules are defined as rule families, thus for example the family $(\textsc{Abs}_{\nu})_{\nu \in \{\circ,\epsilon\}}$ consists of the rules $(\textsc{Abs}_{\circ})$ and $(\textsc{Abs}_{\epsilon})$, where ν in $(\textsc{Abs}_{\nu})_{\nu \in \{\circ,\epsilon\}}$ is replaced by $\circ$ and ϵ, respectively, i.e., ν is a meta-level variable.

The rule (Sub) may come as surprise. The rule is introduced because the two different function types induce a subtype relation. Basically, we can type every function f with standard type $\tau_1 \rightarrow \tau_2$ either to $\tau_1 \rightarrow^{\epsilon} \tau_2$ or to $\tau_1 \rightarrow^{\circ} \tau_2$. But, the type has a significant influence when we pass the function as argument to another function.

On the one hand, consider function

$$g = \lambda h :: \alpha \rightarrow^{\epsilon} \beta.\mathbf{let!}\ h' = h\ \mathbf{in}\ 42$$

It strictly evaluates its argument and hence, should take only arguments h that are allowed to be strictly evaluated, i.e., g should be typeable to $(\alpha \rightarrow^{\epsilon} \beta) \rightarrow^{\epsilon}$

$$\alpha \preceq \alpha \ (\textsc{S-Var}) \qquad \frac{\tau_1' \preceq \tau_1 \qquad \tau_2 \preceq \tau_2'}{\tau_1 \to^{\nu} \tau_2 \preceq \tau_1' \to^{\nu'} \tau_2'} \ (\textsc{S-Arrow}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\},\nu' \leqslant \nu} \qquad \frac{\tau \preceq \tau'}{[\tau] \preceq [\tau']} \ (\textsc{S-List})$$

Figure 5.6: Subtyping rules of $\lambda^{\alpha}_{\mathbf{seq}*}$

Nat, but not to $(\alpha \to^{\circ} \beta) \to^{\epsilon} Nat$. Also annotating the argument h in g by $\alpha \to^{\circ} \beta$ should make g untypeable. Thus, a function f serves as input to g only if it is associated with type $\alpha \to^{\epsilon} \beta$.

On the other hand, consider the function

$$g' = \lambda h :: \alpha \to^{\circ} \beta.42$$

It does not evaluate its argument strictly and hence is allowed to take functions of type $\alpha \to^{\circ} \beta$ as argument. Nevertheless, g' should be allowed to take functions that are suitable for strict evaluation as input, i.e., g' should be typeable to $(\alpha \to^{\circ} \beta) \to^{\epsilon} Nat$ and $(\alpha \to^{\epsilon} \beta) \to^{\epsilon} Nat$; and it should also remain typeable if h is annotated by $\alpha \to^{\epsilon} \beta$.

In essence, there are really functions of type $(\tau_1 \to^{\epsilon} \tau_2) \to^{\epsilon} \tau_3$ that are not typeable to $(\tau_1 \to^{\circ} \tau_2) \to^{\epsilon} \tau_3$ but *not* vice versa, i.e., $(\tau_1 \to^{\circ} \tau_2) \to^{\epsilon} \tau_3$ is a subtype of $(\tau_1 \to^{\epsilon} \tau_2) \to^{\epsilon} \tau_3$. Additionally, even if every function of type $\tau_1 \to^{\epsilon} \tau_2$ is also typeable to $\tau_1 \to^{\circ} \tau_2$ and vice versa, the ones typed to $\tau_1 \to^{\epsilon} \tau_2$ can serve as input to more functions, i.e., they have an additional property and hence $\tau_1 \to^{\epsilon} \tau_2$ can be seen as a subtype of $\tau_1 \to^{\circ} \tau_2$. The rules in Figure 5.6 set up the just described subtype relation and are called via the typing rule (Sub) that allows each type to be transformed into one of its supertypes. The transformation is safe, because every element in a type interpretation is also an element of the interpretations of all its supertypes. Note also that the super- and subtype relation switches between argument and result position in the rule $(\textsc{S-Arrow}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\},\nu' \leqslant \nu}$, i.e., the standard contravariant interpretation of subtyping for function argument types applies.

shape conformant subtyping

In the terminology of annotated type systems, we establish a system with *shape conformant subtyping*, because the types of the underlying type system (the one of $\lambda^{\alpha}_{\mathbf{seq}}$) are not changed by the subtype relation. A very intuitive description of subtyping is to weaken the information contained in an annotated type.

The intention behind refined typing was to enhance free theorems *without* changing the expressiveness of the calculus, neither w.r.t. the terms typeable, nor w.r.t. their semantic interpretation. We can directly define the semantics of $\lambda^{\alpha}_{\mathbf{seq}*}$ in terms of the semantics of $\lambda^{\alpha}_{\mathbf{seq}}$ and also easily show equivalence of the set of typeable terms. The central idea to do so is an annotation eraser. To ease notation we set up a convention that ϵ is invisible.

Definition 32 (annotation eraser)

The function $|\cdot|$, called *annotation eraser*, takes a term, type or typing context in $\lambda^{\alpha}_{\mathsf{seq}*}$ and returns it with all annotations at type variables and arrows removed.

Convention 11 (invisible ϵ)

The annotation ϵ is invisible, i.e., for type variables α we can write α instead of α^{ϵ} and instead of $\rightarrow^{\epsilon}$ we can write $\rightarrow$.

Supported by the annotation eraser and the above convention, we describe the relation between typeability in $\lambda^{\alpha}_{\mathsf{seq}}$ and $\lambda^{\alpha}_{\mathsf{seq}*}$.

Lemma 19 (conservative type extension)

If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathsf{seq}}$, then $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathsf{seq}*}$. If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathsf{seq}*}$, then $|\Gamma| \vdash |t| :: |\tau|$ valid in $\lambda^{\alpha}_{\mathsf{seq}}$.

Lemma 19 states that the annotated type system is a *conservative extension* of the underlying type system, i.e., the system of $\lambda^{\alpha}_{\mathsf{seq}}$. The proof of Lemma 19 requires the following two auxiliary results.

conservative extension

Lemma 20

If τ is closed under Γ_T, then $|\Gamma_T| \vdash |\tau| \in \mathsf{Seqable}$.

Lemma 21 (shape conformance of subtyping)

If $\tau \preceq \tau'$, then $|\tau| = |\tau'|$.

Proof (Lemma 19). To prove the first statement of Lemma 19 we must, for each derivation tree in $\lambda^{\alpha}_{\mathsf{seq}}$, find a derivation tree in $\lambda^{\alpha}_{\mathsf{seq}*}$ that yields the same typing judgment. The proof proceeds by induction on the depth of the derivation tree in $\lambda^{\alpha}_{\mathsf{seq}}$, i.e., it suffices to regard only the rule at the root of a derivation tree (the rule applied last to yield the final typing judgment). We can translate each typing rule of $\lambda^{\alpha}_{\mathsf{seq}}$ to the corresponding rule in $\lambda^{\alpha}_{\mathsf{seq}*}$ by setting all annotations to ϵ. The additional premise on (SLET′) is fulfilled by Lemma 20.

To prove the second statement of Lemma 19 we convert each type derivation in $\lambda^{\alpha}_{\mathsf{seq}*}$ to one in $\lambda^{\alpha}_{\mathsf{seq}}$ that constructs the same typing judgment without any annotations. All rules in $\lambda^{\alpha}_{\mathsf{seq}*}$ with a corresponding unannotated version in $\lambda^{\alpha}_{\mathsf{seq}}$ can be replaced by the unannotated version. By Lemma 21 we can drop the remaining rule (SUB). $\square$

The semantics of types and terms in $\lambda^{\alpha}_{\mathsf{seq}*}$ is easily reduced to the semantics in $\lambda^{\alpha}_{\mathsf{seq}}$. Annotations in types do not influence the semantics at all, their only aim is to influence the relational interpretation of types and thereby to strengthen parametricity results. Or, in the words of Nielson and Nielson (1999, Section 3) we have no effect system. We define type and term semantics in $\lambda^{\alpha}_{\mathsf{seq}*}$ as follows.

$$\begin{aligned}
\Delta^{\mathbf{seq}*}_{\alpha,\rho} &= \rho(\alpha)\\
\Delta^{\mathbf{seq}*}_{[\tau],\rho} &= \mathit{lfp}(\lambda R.\ \{(\bot,\bot),([\,],[\,])\} \cup \{(\mathbf{a} : \mathbf{as}, \mathbf{b} : \mathbf{bs}) \mid (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau,\rho} \wedge (\mathbf{as},\mathbf{bs}) \in R\})\\
\Delta^{\mathbf{seq}*}_{\tau_1 \to^{\circ} \tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \forall (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_1,\rho}.\ (\mathbf{f}\ \$\ \mathbf{a}, \mathbf{g}\ \$\ \mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_2,\rho}\}\\
\Delta^{\mathbf{seq}*}_{\tau_1 \to^{\epsilon} \tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \mathbf{f} = \bot \text{ iff } \mathbf{g} = \bot, \forall (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_1,\rho}.\ (\mathbf{f}\ \$\ \mathbf{a}, \mathbf{g}\ \$\ \mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_2,\rho}\}
\end{aligned}$$

Figure 5.7: Logical relation for $\lambda^{\alpha}_{\mathbf{seq}*}$

Definition 33 (type semantics of $\lambda^{\alpha}_{\mathbf{seq}*}$)

Let τ a type in $\lambda^{\alpha}_{\mathbf{seq}*}$ and θ a type environment with all type variables in τ in its domain. The semantics of τ is defined as $[\![|\tau|]\!]_{\theta}$.

Definition 34 (term semantics of $\lambda^{\alpha}_{\mathbf{seq}*}$)

Let t such that there exist Γ and τ with $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{seq}*}$ and σ a term environment corresponding to Γ. The semantics of t is defined as $[\![|t|]\!]_{\sigma}$.

To complete the investigation of $\lambda^{\alpha}_{\mathbf{seq}*}$, we explore the appropriate logical relation and the resulting parametricity theorem. The main changes to the logical relation are the liftings for function types. The lifting for $\to^{\epsilon}$ is as the lifting in $\lambda^{\alpha}_{\mathbf{seq}}$, i.e., the bottom-reflectingness restriction remains in place, while the lifting for $\to^{\circ}$ is as the lifting in $\lambda^{\alpha}_{\mathbf{fix}}$, i.e., without the bottom-reflectingness restriction. Figure 5.7 shows the whole definition of the logical relation.

If, for a typing context Γ, the environment ρ maps type variables annotated by $\circ$ in Γ to strict and continuous relations and type variables annotated by ϵ to relations that are strict, continuous and bottom-reflecting, then the logical relation is strict and continuous for all types, while for types in Seqable it is additionally bottom-reflecting. Also, the logical relation has a very natural behavior w.r.t. subtyping and finally allows to state and prove the refined parametricity theorem we aim for. The next definition, lemmas and the theorem formalize these facts.

Definition 35 (appropriate relational environment in $\lambda^{\alpha}_{\mathbf{seq}*}$)

In $\lambda^{\alpha}_{\mathbf{seq}*}$ a relational environment ρ is *appropriate* w.r.t. a type context Γ_T, if $\rho(\alpha) \in \mathit{Rel}^{\bot}$ for every $\alpha^{\circ} \in \Gamma_T$ and $\rho(\alpha) \in \mathit{Rel}^{\top}$ for every $\alpha^{\epsilon} \in \Gamma_T$.

Lemma 22

Let τ closed under Γ_T and ρ appropriate, then

(a) $\Delta^{\mathbf{seq}*}_{\tau,\rho} \in \mathit{Rel}^{\bot}$ *(b)* $\Gamma_T \vdash \tau \in \mathsf{Seqable} \Rightarrow \Delta^{\mathbf{seq}*}_{\tau,\rho} \in \mathit{Rel}^{\top}$

Proof. For *(a)* the proof is by induction on the structure of τ, employing the definition of the logical relation. For the proof of *(b)* we regard the different class membership rules for Seqable (cf. Figure 5.5) and obtain the implication immediately by the definition of the logical relation. □

If τ_1, τ_2 closed under Γ_τ, then $\tau_1 \preceq \tau_2$ implies $\Delta^{\mathtt{seq}*}_{\tau_1,\rho} \subseteq \Delta^{\mathtt{seq}*}_{\tau_2,\rho}$ for every ρ appropriate for Γ_τ.

LEMMA 23

Proof. See Appendix A.2. □

As the final result about type refinement, we prove a refined version of the parametricity theorem that provides stronger, i.e., less restricted, free theorems.

THEOREM 9 (parametricity theorem for $\lambda^{\alpha}_{\mathtt{seq}*}$)

If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathtt{seq}*}$, then for every θ_1, θ_2, ρ, σ_1, σ_2 such that

- for every α° occurring in Γ, $\rho(\alpha) \in \mathit{Rel}^{\perp}(\theta_1(\alpha), \theta_2(\alpha))$,
- for every α^{ϵ} occurring in Γ, $\rho(\alpha) \in \mathit{Rel}^{\top}(\theta_1(\alpha), \theta_2(\alpha))$, and
- for every $x :: \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathtt{seq}*}_{\tau',\rho}$,

we have $([\![|t|]\!]^{\mathtt{seq}}_{\sigma_1}, [\![|t|]\!]^{\mathtt{seq}}_{\sigma_2}) \in \Delta^{\mathtt{seq}*}_{\tau,\rho}$.

Proof. We only point out differences to the proof of Theorem 3. The cases $(\textsc{Abs}_\nu)_{\nu\in\{\circ,\epsilon\}}$ and $(\textsc{App}_\nu)_{\nu\in\{\circ,\epsilon\}}$ (for all possible choices of annotations) are proved exactly like the cases (ABS) and (APP).

For (LCASE), now Lemma 22 *(a)* guarantees strictness of the logical relation. The lemma also ensures strictness and continuity for the cases $(\textsc{Fix}_\nu)_{\nu\in\{\circ,\epsilon\}}$.

For (SLET′) the Seqable-check ensures the required bottom-reflectingness of the logical relation (cf. Lemma 22 *(b)*) and hence the proof proceeds as for (SLET).

For the new rule (SUB), we reason as follows:

$$\begin{array}{l} ([\![|t|]\!]^{\mathtt{seq}}_{\sigma_1}, [\![|t|]\!]^{\mathtt{seq}}_{\sigma_2}) \in \Delta_{\tau_2,\rho} \\ \Leftarrow \quad \{\ \tau_1 \preceq \tau_2 \text{ and Lemma } 23\ \} \\ ([\![|t|]\!]^{\mathtt{seq}}_{\sigma_1}, [\![|t|]\!]^{\mathtt{seq}}_{\sigma_2}) \in \Delta_{\tau_1,\rho} \end{array}$$

□

We close the subsection with an example, showing what we gained by the refined type system. Consider again the fusion property of *foldl* and its restricted variant for *foldl′*. In Section 5.1 we found that to reestablish (5.1) from page 117 with *foldl* exchanged by *foldl′*, totality of *f* is sufficient. But, when we prove (5.1) via free theorems in $\lambda^{\alpha}_{\mathtt{seq}}$ (or normally typed Haskell) extra conditions on the function c arise. The situation changes if we employ the refined type system. To keep the different versions of *foldl* as examples, we translate the definitions given in Figure 5.2 to $\lambda^{\alpha}_{\mathtt{seq}}$. The translations are shown in Figure 5.8.

```
$foldl = \mathbf{fix}\ (\lambda h :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha.$
  $\lambda c :: \alpha \to \beta \to \alpha . \lambda n :: \alpha . \lambda ys :: [\beta].$
    $\mathbf{case}\ ys\ \mathbf{of}\ \{$
      $[]\quad \to n;$
      $x : xs \to h\ c\ (c\ n\ x)\ xs\})$
```

```
$foldl' = \mathbf{fix}\ (\lambda h :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha.$
  $\lambda c :: \alpha \to \beta \to \alpha . \lambda n :: \alpha . \lambda ys :: [\beta].$
    $\mathbf{case}\ ys\ \mathbf{of}\ \{$
      $[]\quad \to n$
      $x : xs \to \mathbf{let!}\ n' = c\ n\ x\ \mathbf{in}\ h\ c\ n'\ xs\})$
```

```
$foldl'' = \mathbf{fix}\ (\lambda h :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha.$
  $\lambda c :: \alpha \to \beta \to \alpha . \lambda n :: \alpha . \lambda ys :: [\beta].$
    $\mathbf{case}\ ys\ \mathbf{of}\ \{$
      $[]\quad \to \mathbf{let!}\ cn' = c\ n\ \mathbf{in}\ n;$
      $x : xs \to \mathbf{let!}\ xs' = xs\ \mathbf{in}$
        $\mathbf{let!}\ x' = x\ \mathbf{in}\ h\ c\ (c\ n\ x')\ xs'\})$
```

```
$foldl''' = \mathbf{fix}\ (\lambda h :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha.$
  $\lambda c :: \alpha \to \beta \to \alpha . \lambda n :: \alpha . \lambda ys :: [\beta].$
    $\mathbf{case}\ ys\ \mathbf{of}\ \{$
      $[]\quad \to \mathbf{let!}\ c' = c\ \mathbf{in}\ n$
      $x : xs \to \mathbf{let!}\ c' = c\ \mathbf{in}\ h\ c'\ (c'\ n\ x)\ xs\})$
```

Figure 5.8: The versions of *foldl* from Figure 5.2 in $\lambda^{\alpha}_{\mathbf{seq}}$-syntax

EXAMPLE 26

The function *foldl'*, as given in Figure 5.8 and with appropriately adjusted refined type annotations, is typeable to $(\alpha \to^{\circ} \beta \to^{\circ} \alpha) \to^{\epsilon} \alpha \to^{\epsilon} [\beta] \to^{\epsilon} \alpha$ under typing context $\Gamma = \alpha^{\epsilon}, \beta^{\circ}$ in $\lambda^{\alpha}_{\mathbf{seq}*}$, and hence, by the parametricity theorem for $\lambda^{\alpha}_{\mathbf{seq}*}$ (Theorem 9) we can deduce (5.2) from page 117 with f strict and total and g strict as conditions, but without any condition on c. Specializing g to the identity function, we obtain the fusion property for *foldl'* with only the conditions we employed for the inductive proof that is depending directly on the function definition.[8]

5.3 Improvement of the Algorithmic Properties of the Typing Rules

In Section 5.2 we end up with a calculus with a refined type system that allows stronger free theorems in the sense that side conditions that arise through the possibility of strict evaluation can safely be eliminated. To employ the typing rules as a typing algorithm good algorithmic properties of the rules are essential. Unfortunately, the rules presented in Figure 5.4 do not enjoy such properties. In particular, they form neither a deterministic nor an always terminating algorithm.

So, what problem exactly should the typing algorithm solve, and where do we encounter difficulties? In principle, the typing algorithm should take a typing context and a term, and, if any such type exists, produce a type to which the term is typeable under the given context. But, which kind of contexts and terms should the algorithm take? Ideally, standard typing contexts and

[8]Again, as in Section 5.1, we obtain a strictness requirement for f that could again probably be eliminated via refined typing à la Launchbury and Paterson (1996).

standardly annotated terms as in $\lambda^{\alpha}_{\mathbf{seq}}$ should be given to it and besides a refined type, also a refined typing context and the input term with refined type annotations added should be returned. Designing such an algorithm, we must be aware that in general one term is typeable to several refined types. We can always solely add ϵ-annotations to get a valid refined type, but depending on the applications of the strictness primitive, ϵ-annotations can be replaced by $\circ$-annotations. Consequently, whole sets of refined types, contexts and type annotations arise. Our interest is limited to the types that pose so few conditions on the parametricity theorem that we get a maximally strong (correct) assertion about the investigated function.

To obtain an always terminating typing algorithm that has the just described input/output behavior, three problems have to be solved:

- termination has to be guaranteed,
- the gap between the unannotated types in contexts and terms that serve as input and the annotated types that are required by the refined typing rules has to be bridged,
- the annotated types that allow maximally strong free theorems have to be extracted from all possible refined types.

In the next subsections, we treat the three problems separately. We end up with a type refinement algorithm that provides us with type annotations that allow for maximally strong free theorems.

Basically, the algorithm we develop forms the second stage of a two-stage type inference algorithm for annotated type systems: The standard types of the underlying type system are known and only the annotations are inferred.

5.3.1 Guarantee of Termination

First, we concentrate on termination. The problematic rule concerning termination is (SUB). It is always applicable. In particular, the reflexivity of the subtype relation allows us to apply it repeatedly and infinitely often. Hence, the rule may cause nontermination and we have to eliminate it — at least in its current form. The main idea is that (SUB) can usually be pushed trough other rules in the derivation tree. For example, consider the following part of a derivation tree:

$$\cfrac{\cfrac{\Gamma \vdash t_1 :: \tau_1 \to^{\circ} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\ (\textsc{App}_{\circ}) \qquad \tau_2 \preceq \tau_2'}{\Gamma \vdash (t_1\ t_2) :: \tau_2'}\ (\textsc{Sub})$$

We can transform it into:

$$\cfrac{\cfrac{\Gamma \vdash t_1 :: \tau_1 \to^{\circ} \tau_2 \qquad \tau_1 \to^{\circ} \tau_2 \preceq \tau_1 \to^{\circ} \tau_2'}{\Gamma \vdash t_1 :: \tau_1 \to^{\circ} \tau_2'}\ (\textsc{Sub}) \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2'}\ (\textsc{App}_{\circ})$$

The same way, we can push applications of (SUB) through most other rules. Only for some rules it is impossible. Consider:

$$\dfrac{\dfrac{\Gamma \vdash t :: \tau \rightarrow^{\circ} \tau}{\Gamma \vdash \textbf{fix}\ t :: \tau}\ (\textsc{Fix}_{\circ}) \qquad \tau \preceq \tau'}{\Gamma \vdash \textbf{fix}\ t :: \tau'}\ (\textsc{Sub})$$

We cannot push (SUB) through (FIX$_\circ$), because on the one hand $\tau' \rightarrow^{\circ} \tau'$ is not a supertype of $\tau \rightarrow^{\circ} \tau$ if τ' is a strict supertype of τ, and on the other hand $\tau \rightarrow^{\circ} \tau'$ is a supertype of $\tau \rightarrow^{\circ} \tau$ (if τ' is a supertype of τ) but not a valid premise for (FIX$_\circ$). Nevertheless, we can eliminate the call to (SUB) by integrating it into (FIX$_\circ$). We replace the above fragment of a derivation tree by

$$\dfrac{\Gamma \vdash t :: \tau \rightarrow^{\circ} \tau \qquad \tau \preceq \tau'}{\Gamma \vdash \textbf{fix}\ t :: \tau'}\ (\textsc{Fix}^{+}_{\circ})$$

Besides pushing applications of (SUB) through the derivation tree, we can fuse successive applications and insert new applications wherever we want because the subtype relation is transitive and reflexive.

LEMMA 24 The subtype relation $\preceq$, given in Figure 5.6, is reflexive and transitive.

Proof. Reflexivity is proved inductively over the type structure. Because subtyping is shape conformant (cf. Lemma 21), we can prove transitivity also on the structure of the type. Details are given in Appendix A.2. □

Eliminating and integrating (SUB) as just described, we transform the typing rules of $\lambda^{\alpha}_{\mathbf{seq}*}$, given in Figure 5.4, to the rules presented in Figure 5.9. The calculus $\lambda^{\alpha}_{\mathbf{seq}*}$ with the original typing rules substituted by the ones from Figure 5.9 is called $\lambda^{\alpha}_{\mathbf{seq}+}$. The two calculi are equivalent w.r.t. typeability.

$\lambda^{\alpha}_{\mathbf{seq}+}$

LEMMA 25 A typing judgment $\Gamma \vdash t :: \tau$ is valid in $\lambda^{\alpha}_{\mathbf{seq}*}$ iff it is valid in $\lambda^{\alpha}_{\mathbf{seq}+}$.

Proof. We use induction on the depth of the derivation tree. Consider $\Gamma \vdash t :: \tau$ derivable in $\lambda^{\alpha}_{\mathbf{seq}*}$. By Lemma 24 we assume that the root of the derivation tree is (SUB) followed by another rule from Figure 5.4. Hence, it suffices to replace every combination (SUB) plus another typing rule of $\lambda^{\alpha}_{\mathbf{seq}*}$ by a rule (sequence) from $\lambda^{\alpha}_{\mathbf{seq}+}$, potentially with calls to (SUB) at the leaves of the derivation fragment. We regard each rule (family) different from (SUB) separately.

To translate a derivation tree in $\lambda^{\alpha}_{\mathbf{seq}+}$ to one in $\lambda^{\alpha}_{\mathbf{seq}*}$ that yields the same typing judgment, we transform each typing rule in $\lambda^{\alpha}_{\mathbf{seq}+}$ into a (sequence of) typing rules in $\lambda^{\alpha}_{\mathbf{seq}*}$.

Concrete transformations are shown in Appendix A.2. □

$$\frac{\tau \preceq \tau'}{\Gamma, x :: \tau \vdash x :: \tau'}\ (\text{VAR}^+) \qquad \frac{\tau \preceq \tau'}{\Gamma \vdash [\,]_\tau :: \tau'}\ (\text{NIL}^+) \qquad \frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]}\ (\text{CONS})$$

$$\frac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau}{\Gamma \vdash (\mathbf{case}\ t_1\ \mathbf{of}\ \{[\,] \to t_1; x_1 : x_2 \to t_2\}) :: \tau}\ (\text{LCASE})$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2 \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1' \to^\nu \tau_2}\ (\text{ABS}_\nu^+)_{\nu \in \{\circ,\epsilon\}} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to^\nu \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\ (\text{APP}_\nu)_{\nu \in \{\circ,\epsilon\}}$$

$$\frac{\Gamma \vdash t :: \tau \to^\nu \tau \qquad \tau \preceq \tau'}{\Gamma \vdash (\mathbf{fix}\ t) :: \tau'}\ (\text{FIX}_\nu^+)_{\nu \in \{\circ,\epsilon\}}$$

$$\frac{\Gamma \vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}\ (\text{SLET}')$$

Figure 5.9: Typing rules of $\lambda^\alpha_{\mathbf{seq}+}$

By the switch from $\lambda^\alpha_{\mathbf{seq}*}$ to $\lambda^\alpha_{\mathbf{seq}+}$ we enhance the algorithmic properties of the typing rules in the way that a typing algorithm, taking a term with refined type annotations and a typing context possibly with $\circ$ and ϵ-annotations, will always terminate.

5.3.2 Allowing Non-Refined Input

To cope with a standard typing context and a standardly annotated term as input, different solutions are possible. As one possibility, we can introduce a preprocessing step to add annotations at unannotated input. Thus we can produce the set of all possible refined contexts and terms we can search types for. The main algorithm then consists of the rules in Figure 5.9 and the rules of the systems they call. The algorithm will be nondeterministic because rules of a rule family with an annotation in a premise that does not occur in the conclusion, like $(\text{ABS}_\nu)_{\nu \in \{\circ,\epsilon\}}$, are necessarily in competition to each other.

An alternative way to allow non-refined input is to factor out the nondeterminism of the typing algorithm. All different types we can find and all different inputs we can generate from standardly typed inputs only differ in the choice of annotations on the type variables in the context and at the arrows. Hence, if we do not require fixed annotations $\circ$ and ϵ, but allow variables instead, we obtain a deterministic algorithm. To do so, we must collect constraints on and between the different annotation variables. The constraints are necessary to restrict the possible instances of the found parametrized type to the ones valid in $\lambda^\alpha_{\mathbf{seq}+}$ and hence in $\lambda^\alpha_{\mathbf{seq}*}$. The set (or conjunction) of constraints captures all nondeterminism. Solving it will yield a set of possible concrete instantiations for the annotation variables, from which we select the ones yielding optimal, i.e., minimally restricted and therefore maximally strong, free theorems. A schematic overview of the final algorithm is given in Figure 5.10.

input: standard typing context and a term with standard type annotations

$\Downarrow$ add annotation variables

parametrized term and typing context

$\Downarrow$ the main algorithm

typing constraint and parametrized type

$\Downarrow$ solve constraint

all possible refined types, terms and contexts with concretely annotated types/type variables

$\Downarrow$ type comparison

output: the refined type-context-term-triples yielding the strongest free theorems

Figure 5.10: Schematic overview of the final algorithm for refined typing

To add annotation variables to standard typing contexts and at normally typed terms requires no special care. Every annotation consists of a distinguished annotation variable, e.g. we can take the variables $\nu_1, \ldots, \nu_n$ if n distinct annotations have to be set.

A typing context and a term, both with annotation variables added, serve as input to the main algorithm which is established by an adaptation of the typing rules of $\lambda^{\alpha}_{\mathbf{seq}+}$. We consider rule families that are parametrized over the different concrete annotations in $\lambda^{\alpha}_{\mathbf{seq}+}$, such as $(\textsc{App}_{\nu})_{\nu \in \{\circ,\epsilon\}}$ or $(\textsc{S-Arrow}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\}, \nu' \leqslant \nu}$, as one rule of the main algorithm that introduces object-level annotation variables with possibly some constraints. For example, the rule family

$$\frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow^{\nu} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\ (\textsc{App}_{\nu})_{\nu \in \{\circ,\epsilon\}}$$

is replaced by the rule:

$$\frac{\langle \dot{\Gamma} \vdash \dot{t_1} \rangle \Rrightarrow (C_1, \dot{\tau_1} \rightarrow^{\nu} \dot{\tau_2}) \qquad \langle \dot{\Gamma} \vdash \dot{t_2} \rangle \Rrightarrow (C_2, \dot{\tau_1'}) \qquad \langle \dot{\tau_1} = \dot{\tau_1'} \rangle \Rrightarrow (C_3)}{\langle \dot{\Gamma} \vdash (\dot{t_1}\ \dot{t_2}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot{\tau_2})}\ (\textsc{App}^{C})$$

The rule $(\textsc{App}^{C})$ sets up a conditional typing judgment for parametrized terms, types and contexts. Here are the respective definitions.

DEFINITION 36 **(parametrized/concrete typing context, term and type)**

A typing context, term or type is called *parametrized* if all type variables in the typing context and all arrows at type annotations or in the type are annotated by an annotation variable. It is called *concrete* if all annotations are fixed, i.e., each annotation is either $\circ$ or ϵ.

To distinguish parametrized from concrete entities, we introduce the following convention.

Parametrized entities are indicated by a dot, e.g. $\dot{\Gamma}$, $\dot{t}$ or $\dot{\tau}$.

Convention 12

A *conditional typing judgment* is a statement of the form

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

where $\dot{\Gamma}$ is a parametrized typing context, $\dot{t}$ a parametrized term, C a propositional logic formula and $\dot{\tau}$ a parametrized type. The propositional logic formula C is built from equations and inequations between the concrete annotations $\circ$ and ϵ and annotation variables.

Definition 37 (conditional typing judgment)

We call C in Definition 37 a *typing constraint*. If C contains no annotation variables it is called *concrete* and we denote its truth value by $[\![C]\!]$.

Definition 38 ((concrete) typing constraint)

Note that the way we denote conditional typing judgments already indicates the algorithmic use of the, though still declarative, typing rules: All statements are of the form $input \Rrightarrow output$. The conditional typing rules are given in Figure 5.11. The rules relate very directly to the typing rules of $\lambda^{\alpha}_{\mathbf{seq}+}$. Only regarding the rule systems that the main typing rules call, some comments are in order. Rules that test if a type is in Seqable now return a typing constraint for the annotations on the type, or the annotations of the type variables in the type context. The rules are shown in Figure 5.12. To keep the input/output-scheme, the rules for subtyping are split into rules for finding supertypes and for finding subtypes. The type searched for is moved to the right-hand side of $\Rrightarrow$, while the input type is placed on the left-hand side. The rules are shown in Figure 5.13. Furthermore, an additional rule system is necessary. Consider the rule family $(\textsc{App}_{\nu})_{\nu \in \{\circ, \epsilon\}}$ again. In both premises of the rules the type τ_1 is present, i.e., it has to be equal in both premises. But, regarding conditional typing via $(\textsc{App}^{C})$, the equality constraint is too restrictive. It would require to employ the same variable names for annotation variables in $\dot{\tau}_1$ in both premises. Instead of equality of variable names, we restrict the instantiation of differently named annotation variables in the way that both possibly differently parametrized occurrences of τ_1 are replaceable only by identical instantiations. Therefore, we set up a rule system that returns a typing constraint that forces the instances of two types to be equal. The rules are given in Figure 5.14. We call the calculus with the just introduced rules for conditional typing $\lambda^{\alpha}_{\mathbf{seq}^{C}}$.

$\lambda^{\alpha}_{\mathbf{seq}^{C}}$

A conditional typing judgment in $\lambda^{\alpha}_{\mathbf{seq}^{C}}$ gives rise to several concrete typing judgments, i.e., typing judgments as in $\lambda^{\alpha}_{\mathbf{seq}*}$ and $\lambda^{\alpha}_{\mathbf{seq}+}$. If we find correct instantiations for all annotation variables, we get a concrete typing judgment. To replace annotation variables by a concrete annotation, we define annotation substitutions by which we can instantiate parametrized entities and the typing constraint.

$$\frac{\langle \dot{\tau} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau'})}{\langle \dot{\Gamma}, x :: \dot{\tau} \vdash x \rangle \Rrightarrow (C, \dot{\tau'})}\ (\textsc{Var}^C) \qquad \frac{\langle \dot{\tau} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau'})}{\langle \dot{\Gamma} \vdash [\,]_{\dot{\tau}} \rangle \Rrightarrow (C, \dot{\tau'})}\ (\textsc{Nil}^C)$$

$$\frac{\langle \dot{\Gamma} \vdash \dot{t_1} \rangle \Rrightarrow (C_1, \dot{\tau}) \qquad \langle \dot{\Gamma} \vdash \dot{t_2} \rangle \Rrightarrow (C_2, [\dot{\tau'}]) \qquad \langle \dot{\tau} = \dot{\tau'} \rangle \Rrightarrow (C_3)}{\langle \dot{\Gamma} \vdash (\dot{t_1} : \dot{t_2}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, [\dot{\tau}])}\ (\textsc{Cons}^C)$$

$$\frac{\begin{array}{c}\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C_1, [\dot{\tau_1}]) \qquad \langle \dot{\Gamma} \vdash \dot{t_1} \rangle \Rrightarrow (C_2, \dot{\tau}) \\ \langle \dot{\Gamma}, x_1 :: \dot{\tau_1}, x_2 :: [\dot{\tau_1}] \vdash \dot{t_2} \rangle \Rrightarrow (C_3, \dot{\tau'}) \qquad \langle \dot{\tau} = \dot{\tau'} \rangle \Rrightarrow (C_4)\end{array}}{\langle \dot{\Gamma} \vdash (\mathbf{case}\ \dot{t}\ \mathbf{of}\ \{[\,] \to \dot{t_1}; x_1 : x_2 \to \dot{t_2}\}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3 \wedge C_4, \dot{\tau})}\ (\textsc{LCase}^C)$$

$$\frac{\langle \dot{\Gamma}, x :: \dot{\tau_1} \vdash \dot{t} \rangle \Rrightarrow (C_1, \dot{\tau_2}) \qquad \langle \cdot \preceq \dot{\tau_1} \rangle \Rrightarrow (C_2, \dot{\tau_1'})}{\langle \dot{\Gamma} \vdash (\lambda x :: \dot{\tau_1}.\dot{t}) \rangle \Rrightarrow (C_1 \wedge C_2, \dot{\tau_1'} \to^{\nu} \dot{\tau_2})}\ (\textsc{Abs}^C)$$

$$\frac{\langle \dot{\Gamma} \vdash \dot{t_1} \rangle \Rrightarrow (C_1, \dot{\tau_1} \to^{\nu} \dot{\tau_2}) \qquad \langle \dot{\Gamma} \vdash \dot{t_2} \rangle \Rrightarrow (C_2, \dot{\tau_1'}) \qquad \langle \dot{\tau_1} = \dot{\tau_1'} \rangle \Rrightarrow (C_3)}{\langle \dot{\Gamma} \vdash (\dot{t_1}\ \dot{t_2}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot{\tau_2})}\ (\textsc{App}^C)$$

$$\frac{\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C_1, \dot{\tau} \to^{\nu} \dot{\tau'}) \qquad \langle \dot{\tau} = \dot{\tau'} \rangle \Rrightarrow (C_2) \qquad \langle \dot{\tau} \preceq \cdot \rangle \Rrightarrow (C_3, \dot{\tau''})}{\langle \dot{\Gamma} \vdash (\mathbf{fix}\ \dot{t}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot{\tau''})}\ (\textsc{Fix}^C)$$

$$\frac{\langle \dot{\Gamma} \vdash \dot{\tau_1} \in \mathsf{Seqable} \rangle \Rrightarrow (C_1) \qquad \langle \dot{\Gamma} \vdash \dot{t_1} \rangle \Rrightarrow (C_2, \dot{\tau_1}) \qquad \langle \dot{\Gamma}, x :: \dot{\tau_1} \vdash \dot{t_2} \rangle \Rrightarrow (C_3, \dot{\tau_2})}{\langle \dot{\Gamma} \vdash (\mathbf{let!}\ x = \dot{t_1}\ \mathbf{in}\ \dot{t_2}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot{\tau_2})}\ (\textsc{SLet}^C)$$

Figure 5.11: Conditional typing rules of $\lambda^{\alpha}_{\mathbf{seq}^C}$

$$\frac{\alpha^{\nu} \in \dot{\Gamma}}{\langle \dot{\Gamma} \vdash \alpha \in \mathsf{Seqable} \rangle \Rrightarrow (\nu = \epsilon)}\ (\textsc{CS-Var}^C)$$

$$\langle \dot{\Gamma} \vdash [\dot{\tau}] \in \mathsf{Seqable} \rangle \Rrightarrow (\mathit{True})\ (\textsc{CS-List}^C) \qquad \langle \dot{\Gamma} \vdash \dot{\tau_1} \to^{\nu} \dot{\tau_2} \in \mathsf{Seqable} \rangle \Rrightarrow (\nu = \epsilon)\ (\textsc{CS-Arrow}^C)$$

Figure 5.12: Conditional class membership rules for Seqable in $\lambda^{\alpha}_{\mathbf{seq}^C}$

$$\langle \alpha \preceq \cdot \rangle \Rrightarrow (\mathit{True}, \alpha)\ (\textsc{S-Var}_1^C) \qquad \langle \cdot \preceq \alpha \rangle \Rrightarrow (\mathit{True}, \alpha)\ (\textsc{S-Var}_2^C)$$

$$\frac{\langle \cdot \preceq \dot{\tau_1} \rangle \Rrightarrow (C_1, \dot{\tau_1'}) \qquad \langle \dot{\tau_2} \preceq \cdot \rangle \Rrightarrow (C_2, \dot{\tau_2'})}{\langle \dot{\tau_1} \to^{\nu} \dot{\tau_2} \preceq \cdot \rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu' \leqslant \nu), \dot{\tau_1'} \to^{\nu'} \dot{\tau_2'})}\ (\textsc{S-Arrow}_1^C)$$

$$\frac{\langle \dot{\tau_1'} \preceq \cdot \rangle \Rrightarrow (C_1, \dot{\tau_1}) \qquad \langle \cdot \preceq \dot{\tau_2'} \rangle \Rrightarrow (C_2, \dot{\tau_2})}{\langle \cdot \preceq \dot{\tau_1'} \to^{\nu'} \dot{\tau_2'} \rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu' \leqslant \nu), \dot{\tau_1} \to^{\nu} \dot{\tau_2})}\ (\textsc{S-Arrow}_2^C)$$

$$\frac{\langle \dot{\tau} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau'})}{\langle [\dot{\tau}] \preceq \cdot \rangle \Rrightarrow (C, [\dot{\tau'}])}\ (\textsc{S-List}_1^C) \qquad \frac{\langle \cdot \preceq \dot{\tau'} \rangle \Rrightarrow (C, \dot{\tau})}{\langle \cdot \preceq [\dot{\tau'}] \rangle \Rrightarrow (C, [\dot{\tau}])}\ (\textsc{S-List}_2^C)$$

Figure 5.13: Conditional subtyping rules of $\lambda^{\alpha}_{\mathbf{seq}^C}$

$$\langle \alpha = \alpha \rangle \Rrightarrow (\mathit{True})\ (\text{E-VAR}^C) \qquad \frac{\langle \dot\tau = \dot\tau' \rangle \Rrightarrow (C)}{\langle [\dot\tau] = [\dot\tau'] \rangle \Rrightarrow (C)}\ (\text{E-LIST}^C)$$

$$\frac{\langle \dot\tau_1 = \dot\tau_1' \rangle \Rrightarrow (C_1) \qquad \langle \dot\tau_2 = \dot\tau_2' \rangle \Rrightarrow (C_2)}{\langle \dot\tau_1 \to^{\nu} \dot\tau_2 = \dot\tau_1' \to^{\nu'} \dot\tau_2' \rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu = \nu'))}\ (\text{E-ARROW}^C)$$

Figure 5.14: Conditional equality rules of $\lambda^{\alpha}_{\mathbf{seq}^C}$

DEFINITION 39 (annotation substitution)

A function ϱ that maps from a finite set of annotation variables into the set of concrete annotations $\{\circ, \epsilon\}$ is called an *annotation substitution*.

DEFINITION 40 (instantiation of a parametrized entity or a typing constraint)

Let $\dot\kappa$ a parametrized term, type or typing context, or a typing constraint. By $\dot\kappa\varrho$ we denote the *instantiation* of $\dot\kappa$ under ϱ, i.e., the replacement, in $\dot\kappa$, of all annotation variables ν that occur in $\dot\kappa$ and in the domain of ϱ by $\varrho(\nu)$.

Annotation substitutions enable us to reduce conditional typeability to concrete typeability such that it is equivalent to typeability in $\lambda^{\alpha}_{\mathbf{seq}+}$, and hence in $\lambda^{\alpha}_{\mathbf{seq}*}$. We define (concrete) typeability in $\lambda^{\alpha}_{\mathbf{seq}^C}$ as follows.

DEFINITION 41 (typeability in $\lambda^{\alpha}_{\mathbf{seq}^C}$)

A term t is typeable to τ under Γ in $\lambda^{\alpha}_{\mathbf{seq}^C}$, if there exist $\dot\Gamma$, $\dot t$, $\dot\tau$, C and ϱ, such that $\dot\Gamma\varrho = \Gamma$, $\dot t\varrho = t$, $\dot\tau\varrho = \tau$, $[\![C\varrho]\!] = \mathit{True}$, and $\langle \dot\Gamma \vdash \dot t \rangle \Rrightarrow (C, \dot\tau)$ valid in $\lambda^{\alpha}_{\mathbf{seq}^C}$.

By that notion of concrete typeability, we can, w.r.t. $\lambda^{\alpha}_{\mathbf{seq}+}$, prove syntactic soundness and completeness of the typing algorithm stated by the typing rules of $\lambda^{\alpha}_{\mathbf{seq}^C}$.

THEOREM 10

A term t is typeable to type τ under typing context Γ in $\lambda^{\alpha}_{\mathbf{seq}^C}$ iff it is typeable to the same τ under the same Γ in $\lambda^{\alpha}_{\mathbf{seq}+}$.

The proof of Theorem 10 proceeds basically as follows: We construct for every concrete valid typing judgment in $\lambda^{\alpha}_{\mathbf{seq}+}$ a conditional one that can be instantiated as shown in Definition 41 and thus shows that the original type statement is also valid in $\lambda^{\alpha}_{\mathbf{seq}^C}$.

Conversely, we show that every instance of a conditional typing judgment in $\lambda^{\alpha}_{\mathbf{seq}^C}$ that satisfies the typing constraint is a valid typing judgment in $\lambda^{\alpha}_{\mathbf{seq}+}$.

In the proof, we relate the respective subsystems of the typing rules, before we can prove the following two lemmas that, with Definition 41, yield Theorem 10.

LEMMA 26 If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{seq}+}$, then there exist parametrized $\dot{\Gamma}$, $\dot{t}$, $\dot{\tau}$, a typing constraint C, and an annotation substitution ϱ, such that $\dot{t}\varrho = t$, $\dot{\tau}\varrho = \tau$, $[\![C\varrho]\!] = \mathit{True}$, and $\langle\dot{\Gamma} \vdash \dot{t}\rangle \Rrightarrow (C, \dot{\tau})$ valid.

LEMMA 27 If $\langle\dot{\Gamma} \vdash \dot{t}\rangle \Rrightarrow (C, \dot{\tau})$ valid then for every ϱ, such that $dom(\varrho)$ includes the annotation variables that occur in $\dot{\Gamma}$, $\dot{t}$ or $\dot{\tau}$ and $[\![C\varrho]\!] = \mathit{True}$, we have $\dot{\Gamma}\varrho \vdash \dot{t}\varrho :: \dot{\tau}\varrho$ valid in $\lambda^{\alpha}_{\mathbf{seq}+}$.

The proof is found in Appendix A.2.

5.3.3 Finding Optimal Annotations

Figure 5.10 (page 132) provides an overview of the algorithm for refined typing. Up to now, we described how to get typing constraint and parametrized type. The next step is to solve the constraint. That is, when we have found a valid conditional typing judgment $\langle\dot{\Gamma} \vdash \dot{t}\rangle \Rrightarrow (C, \dot{\tau})$, we search for all annotation substitutions ϱ that have the annotation variables occurring in the conditional typing judgment as domain and satisfy $[\![C\varrho]\!] = \mathit{True}$. A way to find all possible concrete refinements is to consider all annotation substitutions that have the annotation variables occurring in C as domain, choose the ones with $C\varrho = \mathit{True}$, and extend these to obtain annotation substitutions with all annotation variables that occur in the typing judgment as domain.

Once we generated all possible concrete refinements, we can select the ones that provide maximally strong free theorems. But which are these? For annotations on type variables in the typing context it is clear that we prefer $\circ$-annotations instead of ϵ-annotations, because in the parametricity theorem (cf. Theorem 9) for $\circ$-annotated type variables a bottom-reflectingness restriction is dropped. Hence, a totality condition vanishes in the derived free theorem in which relations are specialized to functions. Concerning annotations in the type, i.e., at arrows, the types that allow the strongest free theorems are exactly the ones with minimal (in the sense of related values) logical relations. Intuitively, the connection is quite clear: The smaller the relation, the more we can say about the related items. Fortunately, Lemma 23 already provides a way to characterize the types that give rise to minimal logical relations. They are exactly the minimal types (w.r.t. the subtype relation). Hence, employing the subtyping rules from $\lambda^{\alpha}_{\mathbf{seq}*}$ (cf. Figure 5.6) and preferring $\circ$-annotations over ϵ-annotations in the typing context, we can identify the optimal refined typing judgments, in the sense that they provide for maximally strong free theorems.

By the next example, we show how the complete algorithm works.

EXAMPLE 27 We consider strict function application, i.e., the function

$$(\$!) = \lambda f :: \alpha \to \beta . \lambda x :: \alpha . \mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x'$$

It is typeable under $\Gamma = \alpha, \beta$ in $\lambda^{\alpha}_{\mathsf{seq}}$. To start the main algorithm, i.e., the algorithm stated by the typing rules of $\lambda^{\alpha}_{\mathsf{seq}C}$ and the corresponding subsystems, we annotate type variables in Γ and arrows in type annotations of (\$!) by pairwise distinct annotation variables. We feed the resulting tuple

$$\langle \alpha^{\nu_1}, \beta^{\nu_2} \vdash \lambda f :: \alpha \ \to^{\nu_3} \ \beta.\lambda x :: \alpha.\mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x' \rangle$$

to the main algorithm and get the derivation shown in Figure 5.15. It yields

$$\langle \alpha^{\nu_1}, \beta^{\nu_2} \vdash \lambda f :: \alpha \ \to^{\nu_3} \ \beta.\lambda x :: \alpha.\mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x' \rangle$$
$$\Rrightarrow ((\nu_1 = \epsilon) \wedge (\nu_4 \leqslant \nu_3) \wedge (\nu_3 \leqslant \nu_6), (\alpha \ \to^{\nu_6} \ \beta) \ \to^{\nu_7} \ \alpha \ \to^{\nu_5} \ \beta)$$

The conditional typing judgment contains the annotation variables $\nu_1, \ldots, \nu_7$ and we have to find all ϱ with the domain $D = \{\nu_1, \ldots, \nu_7\}$ that satisfy $[\![C\varrho]\!] = \mathit{True}$ for $C = (\nu_1 = \epsilon) \wedge (\nu_4 \leqslant \nu_3) \wedge (\nu_3 \leqslant \nu_6)$. Then we select the ones that yield optimal typing judgments.

We use the example to consider two extremes of annotation instantiations that we can choose. First, consider we choose all annotations ϵ. That means to create exactly the free theorem we had in $\lambda^{\alpha}_{\mathsf{seq}}$, i.e., to accept all extra conditions arising from possible uses of the strictness primitive. Hence, we expect the arising type to be valid, and indeed, $[\![C\varrho^{\epsilon}]\!] = \mathit{True}$ for ϱ^{ϵ} the constant function from D to ϵ. Second, consider we choose all annotations $\circ$. That means to create exactly the free theorems we had in $\lambda^{\alpha}_{\mathsf{fix}}$, i.e., to disregard all extra conditions arising from possible uses of the strictness primitive. Hence, aware that x of type α is to be evaluated strictly, we expect the arising type to be invalid, and indeed $[\![C\varrho^{\circ}]\!] = \mathit{False}$ for ϱ° the constant function from D to $\circ$.

So, what is the optimal typing judgment? For the example, we do a systematic search. First, we know that every ϵ-annotated type variable in the typing context enforces a constraint in the parametricity theorem, i.e., we try to prefer $\circ$-annotations. For ν_1 we are forced to map it to ϵ by the constraint C, but ν_2 does not appear in C at all. Hence, we can instantiate it by $\circ$. Second, concerning the annotations at arrows, the subtype relation tells that for top-level arrows ϵ is the best choice. The choice is natural concerning free theorems, because the relation for $\to^{\epsilon}$ tells more about related pairs of functions than the relation for $\to^{\circ}$, in particular that $\perp$ is only related to itself. With each nesting level of arrows to the left of other arrows, the preferable annotation changes, which is also quite natural, thinking of input and output positions and the contravariant behavior of function arguments. Top-level arrows describe the function we characterize, i.e., an output position, and therefore we want them as heavily restricted as possible. Arrows nested once to the left of a top-level arrow describe functions that are given as input, i.e., we want as few restrictions as possible on them. Summarizing the above thought, the best choice in our example would be to map ν_6 to $\circ$ and ν_7, as well as ν_5, to ϵ. Doing so, we fulfill the constraint when we fix ν_3 and ν_4 to $\circ$. Hence, we found the best possible concrete typing judgment

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\langle \alpha \preceq \cdot \rangle \\ \Rrightarrow (\mathit{True}, \alpha)\end{array}}{\begin{array}{c}\langle \dot{\Gamma}, x :: \alpha \vdash x \rangle \\ \Rrightarrow (\mathit{True}, \alpha)\end{array}} \quad \dfrac{\alpha^{\nu_1} \in \dot{\Gamma}_T}{\begin{array}{c}\langle \dot{\Gamma}_T \vdash \alpha \in \mathsf{Seqable} \rangle \\ \Rrightarrow (\nu_1 = \epsilon)\end{array}} \quad (1)}{\begin{array}{c}\langle \dot{\Gamma}, x :: \alpha \vdash \mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x' \rangle \\ \Rrightarrow ((\nu_1 = \epsilon) \wedge (\nu_4 \leqslant \nu_3), \beta)\end{array}} \quad \begin{array}{c}\langle \cdot \preceq \alpha \rangle \\ \Rrightarrow (\mathit{True}, \alpha)\end{array}}{\begin{array}{c}\langle \dot{\Gamma} \vdash \lambda x :: \alpha.\mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x' \rangle \\ \Rrightarrow ((\nu_1 = \epsilon) \wedge (\nu_4 \leqslant \nu_3), \alpha \ \to^{\nu_5}\ \beta)\end{array}} \quad \dfrac{\begin{array}{c}\langle \alpha \preceq \cdot \rangle \\ \Rrightarrow (\mathit{True}, \alpha)\end{array} \quad \begin{array}{c}\langle \cdot \preceq \beta \rangle \\ \Rrightarrow (\mathit{True}, \beta)\end{array}}{\begin{array}{c}\langle \cdot \preceq \alpha \ \to^{\nu_3}\ \beta \rangle \\ \Rrightarrow ((\nu_3 \leqslant \nu_6), \alpha \ \to^{\nu_6}\ \beta)\end{array}}}{\begin{array}{c}\langle \alpha^{\nu_1}, \beta^{\nu_2} \vdash \lambda f :: \alpha \ \to^{\nu_3}\ \beta.\lambda x :: \alpha.\mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x' \rangle \\ \Rrightarrow ((\nu_1 = \epsilon) \wedge (\nu_4 \leqslant \nu_3) \wedge (\nu_3 \leqslant \nu_6), (\alpha \ \to^{\nu_6}\ \beta) \ \to^{\nu_7}\ \alpha \ \to^{\nu_5}\ \beta)\end{array}}$$

where (1) is

$$\dfrac{\dfrac{\dfrac{\langle \cdot \preceq \alpha \rangle \Rrightarrow (\mathit{True}, \alpha) \qquad \langle \beta \preceq \cdot \rangle \Rrightarrow (\mathit{True}, \beta)}{\langle \alpha \ \to^{\nu_3}\ \beta \preceq \cdot \rangle \Rrightarrow ((\nu_4 \leqslant \nu_3), \alpha \ \to^{\nu_4}\ \beta)}}{\begin{array}{c}\langle \dot{\Gamma}, x :: \alpha, x' :: \alpha \vdash f \rangle \\ \Rrightarrow ((\nu_4 \leqslant \nu_3), \alpha \ \to^{\nu_4}\ \beta)\end{array}} \quad \dfrac{\langle \alpha \preceq \cdot \rangle \Rrightarrow (\mathit{True}, \alpha)}{\begin{array}{c}\langle \dot{\Gamma}, x :: \alpha, x' :: \alpha \vdash x' \rangle \\ \Rrightarrow (\mathit{True}, \alpha)\end{array}} \quad \begin{array}{c}\langle \alpha = \alpha \rangle \\ \Rrightarrow (\mathit{True})\end{array}}{\langle \dot{\Gamma}, x :: \alpha, x' :: \alpha \vdash f\ x' \rangle \Rrightarrow ((\nu_4 \leqslant \nu_3), \beta)}$$

and $\dot{\Gamma} = \alpha^{\nu_1}, \beta^{\nu_2}, f :: \alpha \ \to^{\nu_3}\ \beta$.

Figure 5.15: Example derivation with the typing rules of $\lambda^{\alpha}_{\mathbf{seq}^C}$

we can get in $\lambda^{\alpha}_{\mathbf{seq}^C}$, $\lambda^{\alpha}_{\mathbf{seq}+}$ and $\lambda^{\alpha}_{\mathbf{seq}*}$ w.r.t. the strength of free theorems:

$$\alpha^{\epsilon}, \beta^{\circ} \vdash (\lambda f :: \alpha \to^{\circ} \beta.\lambda x :: \alpha.\mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x') :: (\alpha \to^{\circ} \beta) \to^{\epsilon} \alpha \to^{\epsilon} \beta$$

From the refined type, we derive the following free theorem: For all types $\tau_1, \ldots, \tau_4$, every strict and total function $f :: \tau_1 \to \tau_2$ and strict function $g :: \tau_3 \to \tau_4$, we have[9]

$$\begin{array}{l}(((\$!)[\tau_1/\alpha, \tau_3/\beta] \equiv \bot) \Leftrightarrow ((\$!)[\tau_2/\alpha, \tau_4/\beta] \equiv \bot)) \\ \quad \wedge \forall p :: \tau_1 \to \tau_3, q :: \tau_2 \to \tau_4. \\ \qquad (\forall x :: \tau_1.\ g\ (p\ x) \equiv q\ (f\ x)) \\ \qquad\quad \Rightarrow (((p\ \$!) \equiv \bot) \Leftrightarrow ((q\ \$!) \equiv \bot)) \\ \qquad\qquad \wedge \forall y :: \tau_1.\ g\ (p\ \$!\ y) \equiv q\ \$!\ (f\ y)\end{array}$$

The theorem is a significant improvement compared to the standard free theorem (i.e., the one derivable in $\lambda^{\alpha}_{\mathbf{seq}}$). The standard theorem would additionally force g total and require $(p \equiv \bot_{\tau_1 \to \tau_3}) \Leftrightarrow (q \equiv \bot_{\tau_2 \to \tau_4})$.

[9]We employ some syntactic sugar, writing (\$!) as infix operator \$!, in particular we write $f\ \$!\ x$ instead of (\$!) $f\ x$ and (f \$!) instead of (\$!) f.

Having seen the above example, the following question may come into mind: Why do we speak of optimal typing judgments, and not of a best typing judgment? The answer is simple: There can be many, incomparable, valid, optimal typing judgments, i.e., there is no best one. Consider the following example.

Example 28

The identity function restricted to (endo-) functions can be defined as

$$(\lambda x :: \alpha \to \alpha.x) :: (\alpha \to \alpha) \to (\alpha \to \alpha)$$

The type is refineable to $(\alpha \to^{\circ} \alpha) \to^{\epsilon} (\alpha \to^{\circ} \alpha)$ or to $(\alpha \to^{\epsilon} \alpha) \to^{\epsilon} (\alpha \to^{\epsilon} \alpha)$ under $\Gamma = \alpha^{\circ}$. Both typing judgments are optimal in the set of valid refined statements. The refinement to $(\alpha \to^{\circ} \alpha) \to^{\epsilon} (\alpha \to^{\epsilon} \alpha)$ under $\Gamma = \alpha^{\circ}$, i.e., the refined statement that would yield the strongest free theorem, is invalid.

5.4 The Implemented Algorithm

The type refinement algorithm defined by the typing rules of $\lambda^{\alpha}_{\mathtt{seq}^C}$ and described in Section 5.3 is implemented and can be used via a web interface at http://www-ps.iai.uni-bonn.de/cgi-bin/polyseq.cgi . Also, the source code is available at Hackage: http://hackage.haskell.org/package/free-theorems-seq-1.0 (library plus command line interface) and http://hackage.haskell.org/package/free-theorems-seq-webui-1.0.0.2 (web interface).

The implementation extends the considered language. It allows type abstraction and application. It also includes Boolean values and integers. Furthermore, some syntactic sugar is added. As another change, possible due to the explicit type abstraction, we omit typing contexts (and thereby annotated type variables) in the interface to the algorithm. Instead, the annotation is attached to an explicit $\forall$-quantifier in the type syntax. Figure 5.16 gives an overview of the implementation's syntax. It also provides hints on how things are expressed with ASCII-characters. Particularly helpful to test the influence of strict evaluation at a certain point of the program is the easy switch of strict and nonstrict **let**-expressions: One can "turn on and off" strict evaluation by adding or removing the "!" right after **let**.

The algorithm, as accessible via the web interface, takes only closed terms. As already indicated, we do not need a typing context as input. We can explicitly abstract over all type variables occurring in the type annotations of the input term, or leave it to the implementation to add all occurring type variables to a, to the user invisible, typing context. The implementation does not only provide the optimal refined types, it also presents the respective free theorems for these types and the free theorem for the standard type. All parts of the theorems that are only necessary because of strict evaluation are highlighted. In general, the highlighted parts in the theorems for refined types are at most as many as in the standard theorem. Figure 5.17 shows a screenshot of the output for *foldl''*.

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid \mathit{Int} \mid \mathit{Bool} \mid [\tau] \mid \tau \to \tau \mid \tau \to^\circ \tau \mid \forall \alpha.\tau \mid \forall^\circ \alpha.\tau \\
t & ::= & x \mid n \mid \mathit{True} \mid \mathit{False} \mid [\,]_\tau \mid t : t \mid [t, \ldots, t]_\tau \mid t + t \mid \textbf{case}\ t\ \textbf{of}\ \{\mathit{True} \to t; \mathit{False} \to t\} \\
& \mid & \textbf{case}\ t\ \textbf{of}\ \{\mathit{False} \to t; \mathit{True} \to t\} \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t \mid \textbf{case}\ t\ \textbf{of}\ \{0 \to t; \_ \to t\} \\
& \mid & \textbf{case}\ t\ \textbf{of}\ \{[\,] \to t; x : xs \to t\} \mid \lambda x :: \tau.t \mid t\ t \mid \Lambda \alpha.t \mid t_\tau \mid \textbf{fix}\ t \\
& \mid & \textbf{let}\ x = t\ \textbf{in}\ t \mid \textbf{let!}\ x = t\ \textbf{in}\ t \mid \textbf{seq}\ t\ t
\end{array}$$

where ASCII representations for special characters are as in the following examples

pretty	ASCII	pretty	ASCII	pretty	ASCII
$\to$	`->`	$\forall$	`forall`	$\Lambda\alpha.t$	`/\a.t`
$\to^\circ$	`->^o`	$\forall^\circ$	`forall^o`	f_{Int}	`f_{Int}`

and variables can have any (non-keyword) alphanumeric string, starting with a letter.

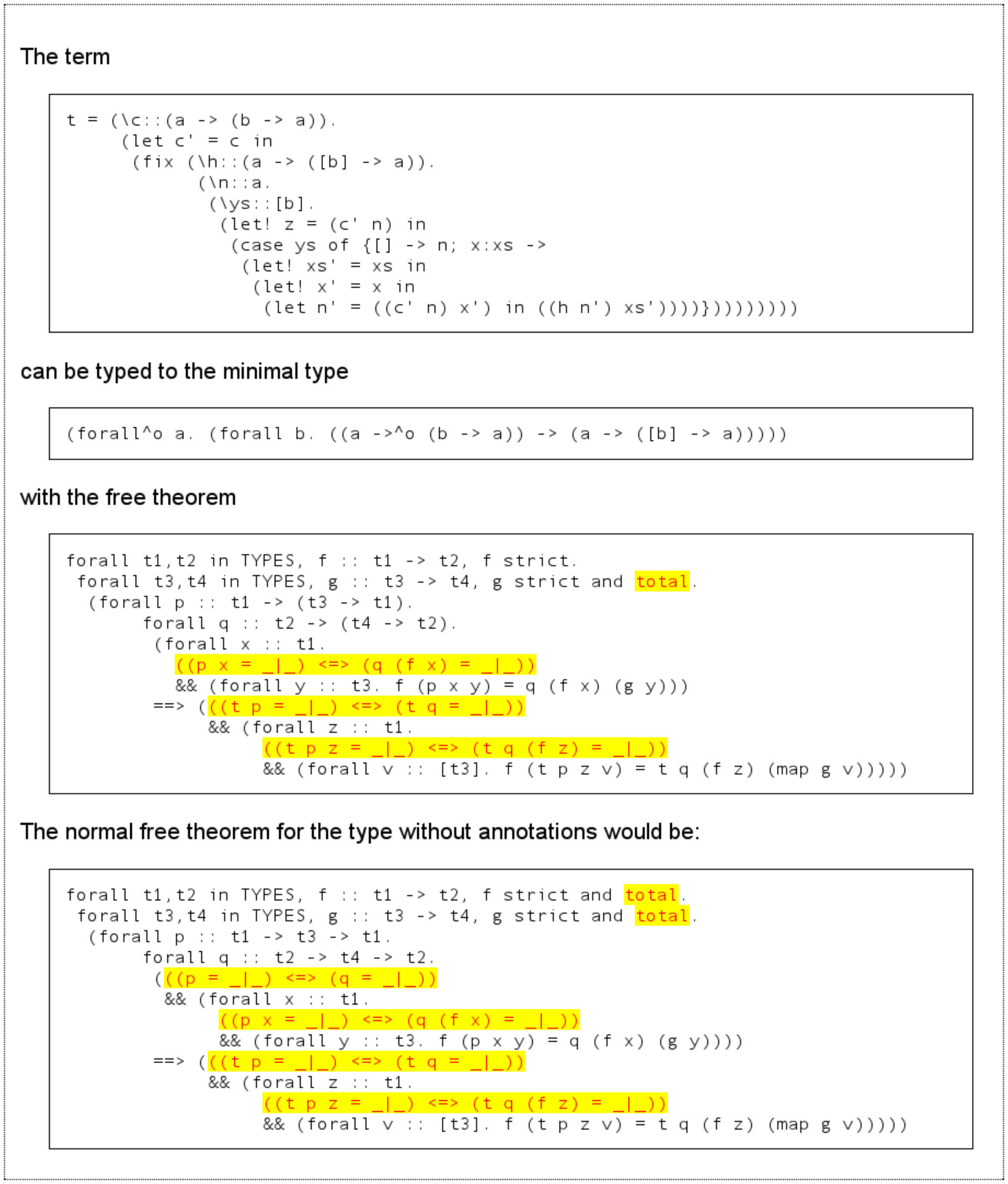
Figure 5.16: Syntax of the implemented version of the type refinement algorithm

5.5 Summary

We presented an annotated type system to localize the influence of selective strict evaluation on free theorems. Based on the insights of Johann and Voigtländer (2004) and inspired by the way Launchbury and Paterson (1996) localized the influence of general recursion, we developed a type system that expresses via annotations at type variables in typing contexts and at arrows, if in a term t the evaluation of a specific subterm (or a term given for a variable t abstracts over) is definitely *not forced* via a strictness primitive, or if it may be evaluated because of such a primitive even if not needed. The annotated type system leads to a stronger parametricity theorem, Theorem 9, that allows to drop totality restrictions and also to relax extra conditions on functions involved in the respective free theorems.

To derive refined types automatically, we reworked the original system of annotated typing rules and improved the algorithmic properties. In the end, we arrived at an algorithm for conditional typeability and employed it to obtain all optimally refined typing judgments, in the sense of maximally strong free theorems. The final algorithm is implemented and available online at http://www-ps.iai.uni-bonn.de/cgi-bin/polyseq.cgi .

Summarizing our work in terms of Nielson and Nielson (1999), we developed an annotated type system with shape conformant subtyping, showed it to be a conservative extension of the underlying type system, i.e., the type system of $\lambda^\alpha_{\mathbf{seq}}$, and developed the second stage of a two-stage type inference algorithm, i.e., an algorithm generating possible annotations to the already given type of the underlying type system. As described by Nielson and Nielson (1999) our algorithm "operates on a free algebra by restricting annotations to be annotation variables only [...] and by recording a set of constraints for the meaning of the annotation variables". We proved the algorithm syntactically sound and complete.



The term

```
t = (\c::(a -> (b -> a)).
     (let c' = c in
      (fix (\h::(a -> ([b] -> a)).
            (\n::a.
             (\ys::[b].
              (let! z = (c' n) in
               (case ys of {[] -> n; x:xs ->
                (let! xs' = xs in
                 (let! x' = x in
                  (let n' = ((c' n) x') in ((h n') xs'))))}))))))))
```

can be typed to the minimal type

```
(forall^o a. (forall b. ((a ->^o (b -> a)) -> (a -> ([b] -> a)))))
```

with the free theorem

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  (forall p :: t1 -> (t3 -> t1).
       forall q :: t2 -> (t4 -> t2).
        (forall x :: t1.
          ((p x = _|_) <=> (q (f x) = _|_))
          && (forall y :: t3. f (p x y) = q (f x) (g y)))
        ==> (((t p = _|_) <=> (t q = _|_))
             && (forall z :: t1.
                  ((t p z = _|_) <=> (t q (f z) = _|_))
                  && (forall v :: [t3]. f (t p z v) = t q (f z) (map g v)))))
```

The normal free theorem for the type without annotations would be:

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict and total.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  (forall p :: t1 -> t3 -> t1.
       forall q :: t2 -> t4 -> t2.
        (((p = _|_) <=> (q = _|_))
         && (forall x :: t1.
              ((p x = _|_) <=> (q (f x) = _|_))
              && (forall y :: t3. f (p x y) = q (f x) (g y))))
        ==> (((t p = _|_) <=> (t q = _|_))
             && (forall z :: t1.
                  ((t p z = _|_) <=> (t q (f z) = _|_))
                  && (forall v :: [t3]. f (t p z v) = t q (f z) (map g v)))))
```

Figure 5.17: Output of the web interface for *foldl''*

Interestingly, a work quite similar to ours, but taming selective strictness from a somehow opposite direction, has been published by Holdermans and Hage (2010). They incorporate explicit strictness annotations in the spirit of **seq** into a strictness analysis based on relevance typing. Their strategy is, dual to ours, to track where "a term is definitely strictly evaluated" and where "a term might or might not be strictly evaluated" by type annotations. Most other works on strictness analysis do not consider the possibility of strictness primitives. An exception is the algorithm SAL, a variant of Nöcker's abstract reduction based strictness analysis, presented by Schmidt-Schauß et al. (2008). Schmidt-Schauß et al. (2008) remark that, as for our considerations, it is not redundant to include a strictness primitive in a calculus, at least if considering an (at least weakly) typed calculus, as they do.

5.6 Outlook

At the moment we do not employ an (even close to) optimal strategy to solve the constraint that arises by the presented type inference algorithm. We might replace the current strategy by a more efficient one. Nielson and Nielson (1999) write that: "For efficiency the algorithmic techniques often involve the generation of constraint systems in a program independent representation". We already have such a representation, but only to describe valid refined typing judgments, not to gain only the ones that are optimal w.r.t. the strength of free theorems. We might enrich the constraint with further information about optimality and apply an advanced technique to solve the constraint.

Another question worth to be considered is the lack of a principal type in the current approach. As Example 28 shows, with only concrete annotations at types, we do not gain principal types in general. But what if we allow for polymorphic type annotations in the final typing judgments? At least in our simple example, we could quantify over the choice for the two type variables that are altered between the two solutions, to get a single, principal type. The extension to polymorphic annotations could also yield a more abstract formulation of free theorems.

Besides generalization of the annotations, we also might extend the underlying calculus' type structure. In Seidel and Voigtländer (2009c) we already consider type abstraction and instantiation but, to extend our work such that a refined typing mechanism can be implemented in a Haskell compiler, is a long and non-trivial way.

Nevertheless, we may benefit already now from the detailed insights about the influence of strict evaluation on free theorems. In particular, refined typing allows already in the current setting to check if and how exactly program transformations that rely on free theorems are influenced, and considerations about their validity, e.g. as performed by Johann and Voigtländer (2006) and Voigtländer (2008c), can be refined.

Last, but not least, the refined type system can serve as a starting point to develop a counterexample generator to free theorems that ignore necessary extra restrictions arising from forced strict evaluation. Refined types might be employed similarly to their role in the counterexample generator presented in Chapter 4, where the influence of general recursion was considered.

Chapter 6

Looking at Quantitative Aspects

[1] Free theorems imply the semantic equivalence (or at least approximation) of two expressions involving a polymorphic function. Hence, they validate program transformations and exactly this validation is, as already stated in Section 3.2, one of their main applications. Naturally the question arises whether the substitution of an expression by a semantically equal one speeds up the program execution or not. Concerning this question, we consider a very simple example: the program equivalence implied by the free theorem for a function f of the type $\alpha \rightarrow Nat$. When we ignore general recursion and selective strictness, we get for all types τ_1, τ_2, functions $g :: \tau_1 \rightarrow \tau_2$ and terms $x :: \tau_1$ the following equivalence:

$$f\ (g\ x) \equiv f\ x \tag{6.1}$$

Intuitively, the theorem states that every function $f :: \alpha \rightarrow Nat$ is a constant function. It ignores its argument and returns always the same natural number. Concerning evaluation costs, one is tempted to conjecture that the costs for the right-hand side expression never exceed, or are even below, the costs for the left-hand side expression. This conjecture will prove true. But, as simple as the example is, it is sufficient to reveal two main points when we compare the evaluation costs (and thus indirectly the runtime) of two expressions that are proved semantically equivalent via a free theorem:

- Only parametric polymorphism allows a (useful) statement on which expression has less evaluation costs than the other.
- The concrete evaluation strategy is essential for a correct comparison of evaluation costs.

[1]Results have been published in (Seidel and Voigtländer, 2011b)

To show that parametric polymorphism really is essential, imagine function f has type $Nat \rightarrow Nat$. Even if it returns the same value for all inputs it is not forced to have the same evaluation cost (in whatever measure). Say, for example, f always returns zero, but, dependent on the input, it calls itself differently often recursively. Clearly, then the evaluation cost depends on the input, and clearly we can give different g and x such that for one combination the left-hand side and for another the right-hand side of statement (6.1) is more efficient w.r.t. evaluation costs. Let us pin down this fact by a concrete example.

Example 29

Consider the function

$$\begin{aligned} &f :: Nat \rightarrow Nat \\ &f\ x = \textbf{if}\ x == 0\ \textbf{then}\ 0\ \textbf{else}\ f\ (x-1) \end{aligned}$$

and a function

$$\begin{aligned} &g :: Nat \rightarrow Nat \\ &g\ x = 1000 \end{aligned}$$

Regarding left- and right-hand side of statement (6.1), we have $f\ \ x$ less expensive than $f\ \ (g\ \ x)$ if and only if $x \leqslant 1000$ when we count function applications as evaluation costs. Also for other measures of evaluation costs we will find g and x where the left-hand side is less expensive and g and x where the right-hand side is less expensive than the respective other side.

call-by-name
call-by-need
call-by-value
lazy evaluation

strict / non-strict evaluation

Based on the example, we also elaborate on the influence of the evaluation strategy. In general, three different evaluation strategies are distinguished: *call-by-name*, *call-by-need* and *call-by-value*. Call-by-need is also known as *lazy evaluation* and for example applied by Haskell implementations. It delays evaluation as long as possible and that way omits unnecessary evaluation steps. Furthermore, it shares expressions and thereby omits multiple evaluations of the same expression. For example, if we evaluate the application of the function $double = \lambda x.x+x$ to $5+3$ then both x in the function body will first point to the expression $5+3$. If one x in the body needs to be evaluated, $5+3$ is evaluated to 8 and now, since the other x points to the same expression, it points to 8 as well. Hence, when the second x is requested, reevaluation is prevented. In contrast, call-by-name would copy $5+3$ to both x and this way evaluate the sum twice if both x are requested. Thus, call-by-name can be viewed as an "as late as possible" strategy without sharing while call-by-need can be viewed as an "as late as possible" strategy with sharing. Completely differently call-by-value, also called *strict evaluation* (call-by-name and call-by-need are both non-strict), evaluates all function arguments before the function body is evaluated. In the $double$-example we would evaluate $5+3$ and then copy the resulting 8 to both x in the function body. Now, how does the evaluation strategy influence the relative performance of the left-/right-hand side of statement (6.1)? While for call-by-value the right-hand side is more efficient than the left-hand side that

	$f :: \alpha \to Nat$	$f :: \alpha \to \alpha \to \alpha$	$f :: \alpha \to (\alpha, \alpha)$
	$f\ (g\ x) = f\ x$	$f\ (g\ x)\ (g\ y) = g\ (f\ x\ y)$	$f\ (g\ x) = mapPair\ (g, g)\ (f\ x)$
call-by-value	lhs > rhs	lhs > rhs	lhs < rhs
call-by-name	lhs = rhs	lhs = rhs	lhs $\leqslant$ rhs
call-by-need	lhs = rhs	lhs = rhs	lhs < rhs

Figure 6.1: Comparison of evaluation costs under different evaluation strategies

has to evaluate $g\ x$, for the other two strategies both sides are equally efficient because $g\ x$ is never evaluated (since f cannot employ its argument).

The table in Figure 6.1 gives an overview of the comparison of evaluation costs for left- and right-hand sides of free theorems for different types. For the comparison, function applications are counted as cost measure. We abbreviate left-hand side as *lhs* and right-hand side as *rhs*. The function $mapPair$ takes a pair of functions and a pair of arguments and applies the respective functions to the respective arguments. In Haskell we could define it for example as follows:

$$mapPair :: (\alpha \to \beta, \gamma \to \delta) \to (\alpha, \gamma) \to (\beta, \delta)$$
$$mapPair\ (f, g)\ (x, y) = (f\ x, g\ y)$$

Note that, dependent on the cost associated to the function $mapPair$ the relative efficiency of the different sides of the free theorem for $f :: \alpha \to (\alpha, \alpha)$ under call-by-name evaluation differs: Only if $mapPair$ causes costs, the right-hand side is less efficient, otherwise both sides are equally efficient.

The simple examples suggest that for strict evaluation only the number of occurrences of g on each side of the free theorem count. But, a general formal machinery for efficiency analysis via types should also handle more complicated cases. What for example can we say about the relative efficiency between the two sides of the free theorem for type $[\alpha] \to [\alpha]$ as given in statement (1.1) on page 5? Or what is the profit we gain when performing free theorem based program transformations, say short-cut fusion for example? We want to tackle these questions via a general theory and the gain is not only to assert that one program is faster than another, but also to quantify the runtime difference in an appropriately abstract measure (of evaluation cost).

For concrete investigations we have to decide on the evaluation strategy under which we consider costs. The two simple strategies to look at are call-by-name and call-by-value. Call-by-need is much more challenging since evaluation of one part of a program may influence evaluation costs of another, i.e., evaluation costs are not compositional. We concentrate on call-by-value. It is not the evaluation strategy of Haskell, but it is applied in many functional languages, e.g. for most ML dialects such as Standard ML (Milner et al., 1997) or OCaml (Leroy et al., 2010). Call-by-name is, at least from a practical point of view, less interesting than call-by-value. It is less efficient then call-by-need and therefore implementations of non-strict functional languages usually employ call-by-need as evaluation strategy.

What has to be done to incorporate evaluation costs into free theorems? Free theorems are semantic statements. Hence, if they shall incorporate runtime assertions a measure for runtime must be visible in the semantics, i.e., be an external property. From the standard denotational semantics as described in Chapter 2 evaluation costs are not observable. Hence, we need to adjust the semantics and externalize evaluation costs. Suitable adjustments of a denotational semantics are for example presented by Wadler (1988), Rosendahl (1989), Liu and Gómez (2001) and van Stone (2003). Also Sands (1995) worked on a time analysis for functional programs. He employed an operational semantics. We choose an easy to handle, straightforward extension of the standard denotational semantics. We extend it in the style of the instrumented semantics of Rosendahl (1989). Semantic domains do not consist of single values. Each value is paired with its evaluation cost, in our case an integer value. The simplest measure to count as evaluation cost in the lambda calculus is function application. So this will be the cost we count. Adding other costs, for example for constructor applications, is possible but unnecessarily complicates our study.

The work presented here can be summarized as follows. We

- set up an instrumented denotational semantics that externalizes evaluation costs in a very simple λ-calculus,
- develop a theory of relational parametricity incorporating costs based on this semantics,
- apply the theory to several simple examples and
- analyze the concrete performance gains achieved by short-cut deforestation (Gill et al., 1993).

The chapter is structured as follows. In Section 6.1 we introduce the syntax of the λ-calculus we study. It is an extension of the calculus λ^α presented in Section 2.1. In particular, as λ^α, it does not allow general recursion, but it provides primitives for structural recursion on lists and numbers. Section 6.2 introduces the instrumented semantics for the calculus and Section 6.3 presents an appropriate theory of parametricity. In Section 6.4 we apply the newly developed theory to several examples. Besides several small examples, we investigate the efficiency improvements gained via $foldr$ / $build$. The last two Sections (Sections 6.5 and 6.6) summarize the work and discuss directions of further research.

6.1 The Calculus

As the basis for our investigations we employ an extension of the calculus λ^α presented in Section 2.1. Since we do not consider general recursion, the standard denotational semantics presented in Section 2.1 correctly models

$$
\begin{array}{rcll}
\tau & ::= & \ldots & \\
 & | & (\tau, \tau) & \text{tuple type} \\
t & ::= & \ldots & \\
 & | & (t, t) & \text{tuple} \\
 & | & \textbf{case } t \textbf{ of } \{(x, x) \to t\} & \text{case expression for tuples} \\
 & | & \textbf{ifold}(t, t, t) & \text{structural recursion on naturals} \\
 & | & \textbf{lfold}(t, t, t) & \text{structural recursion on lists}
\end{array}
$$

Figure 6.2: Type and term syntax of $\lambda^{\alpha}_{\text{fold}}$, extended from Figure 2.1

$$
\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2) \qquad \Gamma, x_1 :: \tau_1, x_2 :: \tau_2 \vdash t_1 :: \tau}{\Gamma \vdash \textbf{case } t \textbf{ of } \{(x_1, x_2) \to t_1\} :: \tau}\ (\textsc{PCase})
$$

$$
\frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_2 \qquad \Gamma \vdash t_3 :: [\tau_1]}{\Gamma \vdash \textbf{lfold}(t_1, t_2, t_3) :: \tau_2}\ (\textsc{LFold})
$$

$$
\frac{\Gamma \vdash t_1 :: \tau \to \tau \qquad \Gamma \vdash t_2 :: \tau \qquad \Gamma \vdash t_3 :: \mathit{Nat}}{\Gamma \vdash \textbf{ifold}(t_1, t_2, t_3) :: \tau}\ (\textsc{NFold})
$$

Figure 6.3: Additional typing rules of $\lambda^{\alpha}_{\text{fold}}$, extended from Figure 2.2

$$
[\![(\tau_1, \tau_2)]\!]_\theta = [\![\tau_1]\!]_\theta \times [\![\tau_2]\!]_\theta
$$

Figure 6.4: Type semantics of $\lambda^{\alpha}_{\text{fold}}$, extended from Figure 2.3

strict and non-strict evaluation, so we only extend it here. We call the extended calculus $\lambda^{\alpha}_{\text{fold}}$. In Figure 6.2 the additional type and term syntax compared to λ^{α} is shown. The primitives $\textbf{ifold}(\cdot, \cdot, \cdot)$ and $\textbf{lfold}(\cdot, \cdot, \cdot)$ capture structural recursion. Their three arguments are: the step function, the base value and finally the "structure" that is folded over. For example, we can express the map-function over lists as follows.

$\lambda^{\alpha}_{\text{fold}}$

$$
\mathit{map} = \lambda g :: \alpha \to \beta . \lambda ys :: [\alpha] . \textbf{lfold}(\lambda x :: \alpha . \lambda xs :: [\beta] . (g\ x) : xs, []_\beta, ys)
$$

The typing rules of $\lambda^{\alpha}_{\text{fold}}$ (that are new compared to the rules of λ^{α}) are given in Figure 6.3. The standard semantics for the new constructs is given in Figures 6.4 and 6.5. The standard parametricity theorem, Theorem 1, still holds if we extend the logical relation by a lifting for tuples as given in Figure 6.6.

The first task on the way to cost-sensitive free theorems is to enrich the standard semantics with a cost measure. The next section presents a suitable enrichment.

$$\begin{aligned}
[\![(t_1, t_2)]\!]_\sigma &= ([\![t_1]\!]_\sigma, [\![t_2]\!]_\sigma) \\
[\![\textbf{case } t \textbf{ of } \{(x_1, x_2) \to t_1\}]\!]_\sigma &= [\![t_1]\!]_{\sigma[x_1 \mapsto \mathbf{v_1}, x_2 \mapsto \mathbf{v_2}]} \text{ with } [\![t]\!]_\sigma = (\mathbf{v_1}, \mathbf{v_2}) \\
[\![\textbf{lfold}(t_1, t_2, t_3)]\!]_\sigma &= [\![t_1]\!]_\sigma\ \mathbf{v_1}\ ([\![t_1]\!]_\sigma\ \mathbf{v_2} \ldots ([\![t_1]\!]_\sigma\ \mathbf{v_n}\ [\![t_2]\!]_\sigma) \ldots) \text{ with } [\![t_3]\!]_\sigma = [\mathbf{v_1}, \ldots, \mathbf{v_n}] \\
[\![\textbf{ifold}(t_1, t_2, t_3)]\!]_\sigma &= \underbrace{[\![t_1]\!]_\sigma\ ([\![t_1]\!]_\sigma \ldots ([\![t_1]\!]_\sigma}_{[\![t_3]\!]_\sigma \text{ times}}\ [\![t_2]\!]_\sigma) \ldots)
\end{aligned}$$

Figure 6.5: Term semantics of $\lambda^{\alpha}_{\mathbf{fold}}$, extended from Figure 2.4

$$\Delta_{(\tau_1,\tau_2),\rho} = \{((\mathbf{x_1}, \mathbf{x_2}), (\mathbf{y_1}, \mathbf{y_2})) \mid (\mathbf{x_1}, \mathbf{y_1}) \in \Delta_{\tau_1,\rho} \land (\mathbf{x_2}, \mathbf{y_2}) \in \Delta_{\tau_2,\rho}\}$$

Figure 6.6: Logical relation for $\lambda^{\alpha}_{\mathbf{fold}}$, extended from Figure 2.5

6.2 An Instrumented Semantics for Counting Costs

To set up free theorems that incorporate assertions about evaluation costs, we enrich the semantics presented in Section 6.1 with information about evaluation costs. A non-standard semantics that "lifts" typically intensional properties of expressions into the denotations is called an *instrumented semantics* (Jones and Nielson, 1995, Section 4.1.5). Rosendahl (1989) presents an instrumented semantics that externalizes evaluation costs. We adopt this semantics.

instrumented semantics

The basic idea for the instrumented semantics is to tuple the values returned by the standard term semantics (as presented in Figures 2.4 and 6.5) with a cost value that counts operations necessary to evaluate the original expression, i.e., the semantics of a term t is not a value $\mathbf{v}$ anymore, but a tuple $(\mathbf{v}, c)$ where c is some abstract cost measure. As cost measure we employ integers, i.e., we simply count (some) evaluation steps. It seems more natural to consider only non-negative evaluation costs, but the generalization to negative costs comes in handy later on.

Before we regard the actual instrumented term semantics, we clarify to which mathematical structures types are mapped. Because we regard a strict language, there is no need for nested costs in data structures, for example for costs on elements of a list. Strictness forces the list, if evaluated at all, to be fully evaluated and hence top-level costs are sufficient. Only the costs for function application must be nested. As long as a function is not provided with an argument, the cost of applying the function is not encountered and moreover the actual cost may depend on the concrete argument the function is applied to. Otherwise evaluation costs are independent of values. To highlight this fact, we decide to interpret types as sets, i.e., the same way the standard interpretation does, and define a cost-lifting for types that tuples values with costs. The concrete type semantics is given in Figure 6.7 where θ, as for the standard semantics, maps type variables to sets.

$$\begin{aligned}
[\![\alpha]\!]^{¢}_{\theta} &= \theta(\alpha) & [\![(\tau_1,\tau_2)]\!]^{¢}_{\theta} &= [\![\tau_1]\!]^{¢}_{\theta} \times [\![\tau_2]\!]^{¢}_{\theta}\\
[\![Nat]\!]^{¢}_{\theta} &= \mathbb{N} & [\![\tau_1 \to \tau_2]\!]^{¢}_{\theta} &= [\![\tau_1]\!]^{¢}_{\theta} \to \mathcal{C}([\![\tau_2]\!]^{¢}_{\theta})\\
[\![[\tau]]\!]^{¢}_{\theta} &= \{[\mathbf{x_1},\ldots,\mathbf{x_n}] \mid n \in \mathbb{N} \wedge \forall i \in \{1,\ldots,n\}.\ \mathbf{x_i} \in [\![\tau]\!]^{¢}_{\theta}\}
\end{aligned}$$

$$\text{where } \mathcal{C}(S) = \{(\mathbf{x}, c) \mid \mathbf{x} \in S \wedge c \in \mathbb{Z}\}$$

Figure 6.7: Instrumented type semantics with embedded costs

For better reference, we give names to values with and without top-level costs.

For each τ and θ with $\mathrm{UTV}(\tau) \subseteq dom(\theta)$, we call $\mathbf{v} \in [\![\tau]\!]^{¢}_{\theta}$ a *cost-free* value. For each cost-free value $\mathbf{v}$, a pair $\mathbf{x} = (\mathbf{v}, c)$ with $c \in \mathbb{Z}$ is called *cost-full* value. To extract the components of a cost-full value, we introduce the functions

$$val(\mathbf{x}) = \mathbf{v} \qquad\qquad cost(\mathbf{x}) = c$$

Definition 42 (cost-full, cost-free value, $cost(\cdot)$, $val(\cdot)$)

To state the instrumented term semantics, we first need to decide which evaluation steps entail costs. A simple possibility is to assign costs only to function applications. Of course, the choice does not reflect real evaluation costs, but the results we present are independent of the exact costs assigned to function applications, constructor applications or other high-level evaluation steps that correspond to a rule in the definition of the standard denotational semantics. Adjustment of the abstract costs w.r.t. real runtime is not the focus of this work, but Liu and Gómez (2001) show that the attachment of a cost measure to a denotational semantics provides for quite good cost prediction. Thus, potentially our approach is well suited to derive predictions about real runtimes.

The concrete cost-full term semantics is shown in Figure 6.8. Environment σ maps term variables to cost-free values. To express the term semantics in a convenient way, we define several auxiliary semantic functions. The function $c \rhd \mathbf{x}$ increases the cost component of $\mathbf{x}$ by c. The functions $(:^{¢})$, $(+^{¢})$, $(¢)$ and $(\cdot,\cdot)^{¢}$ are cost-sensitive variants of $(:)$, $(+)$, function application and $(\cdot,\cdot)$. They sum up the costs of their arguments. Costs arise only for function applications. We nest a cost of 1 in the body of semantic functions, i.e., via the definition of $[\![\lambda x :: \tau.t]\!]^{¢}_{\sigma}$ cost 1 is introduced. Alternatively, costs could have been added via $(¢)$. Note that the cost-handling for functions clearly corresponds to call-by-value evaluation: On the one hand costs that arise from the evaluation of a function argument are added to the overall costs of a function application directly via $¢$ when a function is applied to an argument. On the other hand, values that are stored in the term environment, i.e., had already been an argument somehow, do not cause costs (except of nested costs when applied to an argument) when they are employed during the evaluation.

To avoid parentheses, we introduce the following conventions.

$$
\begin{aligned}
\llbracket x \rrbracket^{¢}_{\sigma} &= (\sigma(x), 0) & \llbracket t_1\ t_2 \rrbracket^{¢}_{\sigma} &= \llbracket t_1 \rrbracket^{¢}_{\sigma} \mathbin{¢} \llbracket t_2 \rrbracket^{¢}_{\sigma} \\
\llbracket n \rrbracket^{¢}_{\sigma} &= (\mathbf{n}, 0) & \llbracket (t_1, t_2) \rrbracket^{¢}_{\sigma} &= (\llbracket t_1 \rrbracket^{¢}_{\sigma}, \llbracket t_2 \rrbracket^{¢}_{\sigma})^{¢} \\
\llbracket [\,]_{\tau} \rrbracket^{¢}_{\sigma} &= ([\,], 0) & \llbracket \lambda x :: \tau . t \rrbracket^{¢}_{\sigma} &= (\lambda \mathbf{v}. 1 \rhd \llbracket t \rrbracket^{¢}_{\sigma[x \mapsto \mathbf{v}]}, 0) \\
\llbracket t_1 : t_2 \rrbracket^{¢}_{\sigma} &= \llbracket t_1 \rrbracket^{¢}_{\sigma} :^{¢} \llbracket t_2 \rrbracket^{¢}_{\sigma} & \llbracket t_1 + t_2 \rrbracket^{¢}_{\sigma} &= \llbracket t_1 \rrbracket^{¢}_{\sigma} +^{¢} \llbracket t_2 \rrbracket^{¢}_{\sigma}
\end{aligned}
$$

$$
\llbracket \mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\} \rrbracket^{¢}_{\sigma} = \begin{cases} c \rhd \llbracket t_1 \rrbracket^{¢}_{\sigma} & \text{if } \llbracket t \rrbracket^{¢}_{\sigma} = (\mathbf{0}, c) \\ c \rhd \llbracket t_2 \rrbracket^{¢}_{\sigma} & \text{if } \llbracket t \rrbracket^{¢}_{\sigma} = (\mathbf{n}, c) \text{ where } \mathbf{n} > \mathbf{0} \end{cases}
$$

$$
\llbracket \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1; x_1 : x_2 \to t_2\} \rrbracket^{¢}_{\sigma} = \begin{cases} c \rhd \llbracket t_1 \rrbracket^{¢}_{\sigma} & \text{if } \llbracket t \rrbracket^{¢}_{\sigma} = ([\,], c) \\ c \rhd \llbracket t_2 \rrbracket^{¢}_{\sigma[x_1 \mapsto \mathbf{v_1}, x_2 \mapsto [\mathbf{v_2}, \ldots, \mathbf{v_n}]]} & \text{if } \llbracket t \rrbracket^{¢}_{\sigma} = ([\mathbf{v_1}, \ldots, \mathbf{v_n}], c) \\ & \text{where } \mathbf{n} > \mathbf{0} \end{cases}
$$

$$
\begin{aligned}
\llbracket \mathbf{case}\ t\ \mathbf{of}\ \{(x_1, x_2) \to t_1\} \rrbracket^{¢}_{\sigma} &= c \rhd \llbracket t_1 \rrbracket^{¢}_{\sigma[x_1 \mapsto \mathbf{v_1}, x_2 \mapsto \mathbf{v_2}]} \text{ with } \llbracket t \rrbracket^{¢}_{\sigma} = ((\mathbf{v_1}, \mathbf{v_2}), c) \\
\llbracket \mathbf{lfold}(t_1, t_2, t_3) \rrbracket^{¢}_{\sigma} &= (c_1 + c_3) \rhd ((\mathbf{g}\ \mathbf{v_1}) \mathbin{¢} ((\mathbf{g}\ \mathbf{v_2}) \mathbin{¢} \ldots ((\mathbf{g}\ \mathbf{v_n}) \mathbin{¢} \llbracket t_2 \rrbracket^{¢}_{\sigma}) \ldots)) \\
&\quad \text{with } \llbracket t_1 \rrbracket^{¢}_{\sigma} = (\mathbf{g}, c_1),\ \llbracket t_3 \rrbracket^{¢}_{\sigma} = ([\mathbf{v_1}, \ldots, \mathbf{v_n}], c_3) \\
\llbracket \mathbf{ifold}(t_1, t_2, t_3) \rrbracket^{¢}_{\sigma} &= (c_1 + c_3) \rhd (\underbrace{(\mathbf{g}, 0) \mathbin{¢} ((\mathbf{g}, 0) \mathbin{¢} \ldots ((\mathbf{g}, 0) \mathbin{¢}}_{\mathbf{n} \text{ times}} \llbracket t_2 \rrbracket^{¢}_{\sigma}) \ldots)) \\
&\quad \text{with } \llbracket t_1 \rrbracket^{¢}_{\sigma} = (\mathbf{g}, c_1),\ \llbracket t_3 \rrbracket^{¢}_{\sigma} = (\mathbf{n}, c_3)
\end{aligned}
$$

where

$$
\begin{aligned}
c \rhd (\mathbf{v}, c') &= (\mathbf{v}, c + c') \\
\mathbf{x} :^{¢} \mathbf{xs} &= ([\mathbf{v}, \mathbf{v_1}, \ldots, \mathbf{v_n}], c + c') \text{ with } \mathbf{x} = (\mathbf{v}, c),\ \mathbf{xs} = ([\mathbf{v_1}, \ldots, \mathbf{v_n}], c') \\
\mathbf{x_1} +^{¢} \mathbf{x_2} &= (\mathbf{n_1} + \mathbf{n_2}, c_1 + c_2) \text{ with } \mathbf{x_1} = (\mathbf{n_1}, c_1), \mathbf{x_2} = (\mathbf{n_2}, c_2) \\
(\mathbf{x_1}, \mathbf{x_2})^{¢} &= ((\mathbf{v_1}, \mathbf{v_2}), c + c') \text{ with } \mathbf{x_1} = (\mathbf{v_1}, c),\ \mathbf{x_2} = (\mathbf{v_2}, c') \\
\mathbf{f} \mathbin{¢} \mathbf{x} &= (c + c') \rhd (\mathbf{g}\ \mathbf{v}) \text{ with } \mathbf{f} = (\mathbf{g}, c),\ \mathbf{x} = (\mathbf{v}, c')
\end{aligned}
$$

Figure 6.8: Instrumented term semantics with costs

CONVENTION 13

- $\rhd$ and $:^{¢}$ are right-associative,
- $¢$ is left-associative,
- $\rhd$ has always the highest precedence.

The just defined semantics guarantees that every well-typed term is in the cost-lifting of the respective type's semantics.

LEMMA 28 If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fold}}$ then $\llbracket t \rrbracket^{¢}_{\sigma} \in \mathcal{C}(\llbracket \tau \rrbracket^{¢}_{\theta})$ for every mapping θ with $dom(\theta) \supseteq \Gamma_{\mathcal{T}}$ and σ with $\sigma(x) \in \llbracket \tau' \rrbracket^{¢}_{\theta}$ for every $x :: \tau'$ in Γ.

Since costs are mostly summed up and pushed through by the term semantics, shifting costs does not alter the semantics as stated by the following lemma and used (mostly silently) throughout.

Lemma 29

Let $\mathbf{x}$, $\mathbf{xs}$, $\mathbf{x_1}$, $\mathbf{x_2}$, $\mathbf{f}$ cost-full semantic values and c, c' costs, i.e., integers. The following equivalences hold.

- $c \rhd c' \rhd \mathbf{x} = (c + c') \rhd \mathbf{x}$
- $c \rhd (\mathbf{x} :^{¢} \mathbf{xs}) = c \rhd \mathbf{x} :^{¢} \mathbf{xs} = \mathbf{x} :^{¢} c \rhd \mathbf{xs}$
- $c \rhd (\mathbf{x_1}, \mathbf{x_2})^{¢} = (c \rhd \mathbf{x_1}, \mathbf{x_2})^{¢} = (\mathbf{x_1}, c \rhd \mathbf{x_2})^{¢}$
- $c \rhd (\mathbf{f} \; ¢ \; \mathbf{x}) = c \rhd \mathbf{f} \; ¢ \; \mathbf{x} = \mathbf{f} \; ¢ \; c \rhd \mathbf{x}$

Finally, let us show the instrumented semantics in action and calculate the semantics of a program that determines the length of a two element list.

Example 30

We express the length function in $\lambda^{\alpha}_{\mathbf{fold}}$ by

$$\mathit{length} = \lambda xs :: [\alpha].\mathbf{lfold}(\lambda x :: \alpha.\lambda y :: \mathit{Nat}.1 + y, 0, xs)$$

and calculate the semantics of

$$\mathit{length}[\mathit{Nat}/\alpha] \; (1 : 2 : [\,]_{\mathit{Nat}})$$

where type substitution is as described in Section 2.4.

$$\begin{aligned}
& [\![(\lambda xs :: [\mathit{Nat}].\mathbf{lfold}(\lambda x :: \mathit{Nat}.\lambda y :: \mathit{Nat}.1 + y, 0, xs)) \; (1 : 2 : [\,]_{\mathit{Nat}})]\!]^{¢}_{\emptyset} \\
= \; & [\![\lambda xs :: [\mathit{Nat}].\mathbf{lfold}(\lambda x :: \mathit{Nat}.\lambda y :: \mathit{Nat}.1 + y, 0, xs)]\!]^{¢}_{\emptyset} \; ¢ \; [\![1 : 2 : [\,]_{\mathit{Nat}}]\!]^{¢}_{\emptyset} \\
= \; & (\boldsymbol{\lambda}\mathbf{v}.1 \rhd [\![\mathbf{lfold}(\lambda x :: \mathit{Nat}.\lambda y :: \mathit{Nat}.1 + y, 0, xs)]\!]^{¢}_{[xs \mapsto \mathbf{v}]}, 0) \; ¢ \; ([\mathbf{1}, \mathbf{2}], 0) \\
= \; & 1 \rhd [\![\mathbf{lfold}(\lambda x :: \mathit{Nat}.\lambda y :: \mathit{Nat}.1 + y, 0, xs)]\!]^{¢}_{[xs \mapsto [\mathbf{1},\mathbf{2}]]} \\
= \; & 1 \rhd (((\boldsymbol{\lambda}\mathbf{x}.(\boldsymbol{\lambda}\mathbf{y}.(\mathbf{1} + \mathbf{y}, 1), 1)) \; \mathbf{1}) \; ¢ \; (((\boldsymbol{\lambda}\mathbf{x}.(\boldsymbol{\lambda}\mathbf{y}.(\mathbf{1} + \mathbf{y}, 1), 1)) \; \mathbf{2}) \; ¢ \; (\mathbf{0}, 0))) \\
= \; & 1 \rhd ((\boldsymbol{\lambda}\mathbf{y}.(\mathbf{1} + \mathbf{y}, 1), 1) \; ¢ \; 1 \rhd (\mathbf{1} + \mathbf{0}, 1)) \\
= \; & 1 \rhd (1 + 1 + 1) \rhd ((\boldsymbol{\lambda}\mathbf{y}.(\mathbf{1} + \mathbf{y}, 1)) \; (\mathbf{1} + \mathbf{0})) \\
= \; & (\mathbf{2}, 5)
\end{aligned}$$

Exactly the five required beta-reductions (one for $\lambda xs :: [\mathit{Nat}]$ and two for each $\lambda x :: \mathit{Nat}.\lambda y :: \mathit{Nat}$) have been counted. Note that the cost is linearly dependent on the list length. Each additional list entry causes an additional cost of two function applications.

6.3 Parametricity Theory Involving Costs

The key to a cost-sensitive theory of relational parametricity is a suitable relational type interpretation, i.e., a logical relation over the instrumented type

$$
\begin{aligned}
\Delta'_{\alpha,\rho} &= \rho(\alpha)\\
\Delta'_{Nat,\rho} &= id_{\mathbb{N}}\\
\Delta'_{[\tau],\rho} &= \{([\mathbf{v_1},\ldots,\mathbf{v_n}],[\mathbf{v'_1},\ldots,\mathbf{v'_n}]) \mid n\in\mathbb{N}\wedge\forall i\in\{1,\ldots,n\}.\,(\mathbf{v_i},\mathbf{v'_i})\in\Delta'_{\tau,\rho}\}\\
\Delta'_{(\tau_1,\tau_2),\rho} &= \{((\mathbf{v_1},\mathbf{v_2}),(\mathbf{v'_1},\mathbf{v'_2})) \mid (\mathbf{v_1},\mathbf{v'_1})\in\Delta'_{\tau_1,\rho}\wedge(\mathbf{v_2},\mathbf{v'_2})\in\Delta'_{\tau_2,\rho}\}\\
\Delta'_{\tau_1\to\tau_2,\rho} &= \{(\mathbf{f},\mathbf{g}) \mid \forall(\mathbf{v},\mathbf{v'})\in\Delta'_{\tau_1,\rho}.\,(\mathbf{f}\ \mathbf{v},\mathbf{g}\ \mathbf{v})\in\mathcal{C}(\Delta'_{\tau_2,\rho})\}
\end{aligned}
$$

where

$$\mathcal{C}(\mathcal{R}) = \{((\mathbf{v},c),(\mathbf{v}',c)) \mid (\mathbf{v},\mathbf{v}')\in\mathcal{R}\wedge c\in\mathbb{Z}\}$$

Figure 6.9: Logical relation with embedded costs

semantics. To establish such a relation, we consider the difference between the standard and the instrumented type interpretation (cf. Figures 2.3, 6.4 and 6.7). The essential difference is the use of a cost-lifting. Hence, if we find an appropriate cost-lifting for relations, we might be able to give a cost-sensitive version of the standard logical relation presented in Figures 2.5 and 6.6. Alike the cost-sensitive standard type interpretation, we define the relational interpretation on cost-free values. Consequently, the semantic interpretations of terms are related if they are related by the cost-lifting of the logical relation.

But how should the cost lifting for relations look like? Of course, cost-full values should only be related if their value components are. Regarding costs, different choices seem possible. To allow assertions about the difference between evaluation costs, we cannot allow values with arbitrarily different costs to be related. We get precise assertions about the cost difference for two values if we force the cost components to be equal whenever the values are related. This restriction on costs seems to prevent us from getting useful theorems at all. But, we can add artificial costs to values to enforce relatedness. The artificial costs essentially express that the evaluation of a value causes less costs than the evaluation of the value it is related to. The cost-sensitive logical relation is given in Figure 6.9. It allows the following parametricity theorem.

Theorem 11 (parametricity theorem)

If $\Gamma\vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fold}}$, then for every ρ, σ_1, σ_2 such that

- for every α in Γ, $\rho(\alpha)\in Rel$, and
- for every $x :: \tau'$ in Γ, $(\sigma_1(x),\sigma_2(x))\in\Delta'_{\tau',\rho}$,

we have $([\![t]\!]^{¢}_{\sigma_1},[\![t]\!]^{¢}_{\sigma_2})\in\mathcal{C}(\Delta'_{\tau,\rho})$.

The proof of Theorem 11 employs a characterization of the logical relation's lifting for function types that does only rely on cost-full values. We give it via the next lemma, before we prove Theorem 11.

Lemma 30

For all types τ_1, τ_2 and ρ, such that all type variables in τ_1 or τ_2 are in its domain, we have

$$(\mathbf{f},\mathbf{g}) \in \mathcal{C}(\Delta'_{\tau_1\to\tau_2,\rho})$$
$$\Leftrightarrow$$
$$cost(\mathbf{f}) = cost(\mathbf{g}) \wedge \forall(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}).\ (\mathbf{f} ¢\ \mathbf{x}, \mathbf{g} ¢\ \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$$

Proof. See Appendix A.3. □

Proof (Theorem 11). The proof is by induction over the type derivation, i.e., we have to consider the derivation rules in Figures 2.2 and 6.3. In the proof we use the same names for terms and types as in the figures with the typing rules; we name environments as in Theorem 11 (i.e., ρ, σ_1, σ_2) and we assume the conditions on them that are given in Theorem 11 to be satisfied.

We state only selected cases of the proof. A complete version is found in Appendix A.3.

(Sum)

$([\![t_1+t_2]\!]^{¢}_{\sigma_1}, [\![t_1+t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{Nat,\rho})$

$\Leftrightarrow$ { term semantics, definition of $(+^{¢})$ }

$((\mathbf{n_1}+\mathbf{n_2}, c_1+c_2), (\mathbf{n'_1}+\mathbf{n'_2}, c'_1+c'_2)) \in \mathcal{C}(\Delta'_{Nat,\rho})$

where $(\mathbf{n_1},c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1},c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2},c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2},c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftrightarrow$ { definition of cost-lifting and $\Delta'_{Nat,\rho}$ }

$\mathbf{n_1}+\mathbf{n_2} = \mathbf{n'_1}+\mathbf{n'_2} \wedge c_1+c_2 = c'_1+c'_2$

where $(\mathbf{n_1},c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1},c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2},c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2},c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftarrow$ { sum of equal addends is equal }

$\mathbf{n_1} = \mathbf{n'_1} \wedge \mathbf{n_2} = \mathbf{n'_2} \wedge c_1 = c'_1 \wedge c_2 = c'_2$

where $(\mathbf{n_1},c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1},c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2},c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2},c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftrightarrow$ { definition of $\Delta'_{Nat,\rho}$ and cost-lifting }

$((\mathbf{n_1},c_1),(\mathbf{n'_1},c'_1)) \in \mathcal{C}(\Delta'_{Nat,\rho}) \wedge ((\mathbf{n_2},c_2),(\mathbf{n'_2},c'_2)) \in \mathcal{C}(\Delta'_{Nat,\rho})$

where $(\mathbf{n_1},c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1},c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2},c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2},c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftrightarrow$ { term semantics }

$([\![t_1]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{Nat,\rho}) \wedge ([\![t_2]\!]^{¢}_{\sigma_1}, [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{Nat,\rho})$

The last statement is true by the induction hypotheses.

(NCASE)

The proof is by case distinction on the value of $[\![t]\!]^{¢}_{\sigma_1}$. First, we assume $[\![t]\!]^{¢}_{\sigma_1} = (\mathbf{0}, c)$ for an arbitrary $c \in \mathbb{Z}$. Taking the induction hypothesis from the first premise and the definition of the logical relation for type Nat into account, we know $[\![t]\!]^{¢}_{\sigma_2} = (\mathbf{0}, c)$, too. Hence, evaluating the semantics of the case expression reduces to evaluating the first branch of the semantics of the case expression:

$$([\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_1}, [\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

$\Leftrightarrow$ { term semantics with choice of the first branch }

$$(c \rhd [\![t_1]\!]^{¢}_{\sigma_1}, c \rhd [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$([\![t_1]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

The last statement is by the induction hypothesis from the second premise. Second, we take $[\![t]\!]^{¢}_{\sigma_1} = (\mathbf{n}, c)$ for $\mathbf{n} > \mathbf{0}$ and $c \in \mathbb{Z}$ arbitrary. Again, using the induction hypothesis from the first premise we have $[\![t]\!]^{¢}_{\sigma_2} = (\mathbf{n}, c)$, too. Hence, we can reason as follows:

$$([\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_1}, [\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

$\Leftrightarrow$ { term semantics with choice of the second branch }

$$(c \rhd [\![t_2]\!]^{¢}_{\sigma_1}, c \rhd [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$([\![t_2]\!]^{¢}_{\sigma_1}, [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

The last statement is true by the induction hypothesis from the third premise.

(ABS)

$$([\![\lambda x :: \tau_1 . t]\!]^{¢}_{\sigma_1}, [\![\lambda x :: \tau_1 . t]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_1 \to \tau_2,\rho})$$

$\Leftrightarrow$ { term semantics }

$$((\lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, 0), (\lambda \mathbf{v}'.1 \rhd [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}, 0)) \in \mathcal{C}(\Delta'_{\tau_1 \to \tau_2,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$(\lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, \lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}]}) \in \Delta'_{\tau_1 \to \tau_2,\rho}$$

$\Leftrightarrow$ { definition of $\Delta'_{\tau_1 \to \tau_2,\cdot}$ }

$$\forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1,\rho}.((\lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]})\ \mathbf{v}, (\lambda \mathbf{v}.1 \rhd [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}]})\ \mathbf{v}') \in \mathcal{C}(\Delta'_{\tau_2,\rho})$$

$\Leftrightarrow$ { function application }

$$\forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1,\rho}.(1 \rhd [\![t]\!]^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, 1 \rhd [\![t]\!]^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$\forall(\mathbf{v},\mathbf{v}') \in \Delta'_{\tau_1,\rho}.([\![t]\!]^{¢}_{\sigma_1[x\mapsto\mathbf{v}]}, [\![t]\!]^{¢}_{\sigma_2[x\mapsto\mathbf{v}']}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

The last statement is true by the induction hypothesis.

(App)

$([\![t_1\ t_2]\!]^{¢}_{\sigma_1}, [\![t_1\ t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

$\Leftrightarrow$ { term semantics }

$([\![t_1]\!]^{¢}_{\sigma_1}\ ¢\ [\![t_2]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}\ ¢\ [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

$\Leftarrow$ { induction hypothesis from the second premise }

$\forall(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}).([\![t_1]\!]^{¢}_{\sigma_1}\ ¢\ \mathbf{x}, [\![t_1]\!]^{¢}_{\sigma_2}\ ¢\ \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

$\Leftarrow$ { Lemma 30 }

$([\![t_1]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_1\to\tau_2,\rho})$

The last statement is true by the induction hypothesis from the first premise.

(LFold)

We take $[\![t_1]\!]^{¢}_{\sigma_1} = (\mathbf{f}, c_1)$ and thus have $[\![t_1]\!]^{¢}_{\sigma_2} = (\mathbf{f}', c_1)$ with $(\mathbf{f},\mathbf{f}') \in \Delta'_{\tau_1\to\tau_2\to\tau_2,\rho}$ by the induction hypothesis from the first premise. By the induction hypothesis from the third premise, taking $[\![t_3]\!]^{¢}_{\sigma_1} = ([\mathbf{v_1},\ldots,\mathbf{v_n}], c_3)$, we have $[\![t_3]\!]^{¢}_{\sigma_2} = ([\mathbf{v'_1},\ldots,\mathbf{v'_n}], c_3)$ for the same n and c_3 and with $\{(\mathbf{v_1},\mathbf{v'_1}),\ldots,(\mathbf{v_n},\mathbf{v'_n})\} \subseteq \Delta'_{\tau_1,\rho}$. We reason as follows:

$([\![\mathbf{lfold}(t_1,t_2,t_3)]\!]^{¢}_{\sigma_1}, [\![\mathbf{lfold}(t_1,t_2,t_3)]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

$\Leftrightarrow$ { term semantics }

$((c_1+c_3) \rhd ((\mathbf{f}\ \mathbf{v_1})\ ¢\ (\ldots((\mathbf{f}\ \mathbf{v_n})\ ¢\ [\![t_2]\!]^{¢}_{\sigma_1}))),$
$\quad (c_1+c_3) \rhd ((\mathbf{f}'\ \mathbf{v'_1})\ ¢\ (\ldots((\mathbf{f}'\ \mathbf{v'_n})\ ¢\ [\![t_2]\!]^{¢}_{\sigma_2})))) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

$\Leftrightarrow$ { definition of cost-lifting }

$(((\mathbf{f}\ \mathbf{v_1})\ ¢\ (\ldots((\mathbf{f}\ \mathbf{v_n})\ ¢\ [\![t_2]\!]^{¢}_{\sigma_1}))),$
$\quad ((\mathbf{f}'\ \mathbf{v'_1})\ ¢\ (\ldots((\mathbf{f}'\ \mathbf{v'_n})\ ¢\ [\![t_2]\!]^{¢}_{\sigma_2})))) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

$\Leftarrow$ { side conditions on $\mathbf{f}$, $\mathbf{f}'$, $\mathbf{v_i}$, $\mathbf{v'_i}$, cost-lifting, definition of $\Delta'_{\tau_1\to\tau_2,\rho}$ }

$([\![t_2]\!]^{¢}_{\sigma_1}, [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

The last statement is true by the induction hypothesis from the second premise.

$\square$

When we employ Theorem 11 to establish cost-sensitive free theorems, reasoning is complicated by the interaction between the cost-lifted and the unlifted version of the logical relation. On first sight, the way we presented the logical relation was reasonable because it reflects the alteration from the standard to the instrumented type interpretations and it also highlights the places where costs are of interest. Nevertheless, for reasoning we prefer a completely cost-lifted logical relation. Lemma 30 already provides a characterization for the function lifting of $\mathcal{C}(\Delta'_{\cdot,\cdot})$. For all other liftings, we establish similar characterizations. The proofs for the following lemmas are found in Appendix A.3.

Lemma 31 Let τ_1, τ_2 types and ρ such that all type variables in τ_1 and τ_2 are in its domain. Then

$$(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{(\tau_1,\tau_2),\rho})$$
$$\Leftrightarrow$$
$$\exists (\mathbf{p}, \mathbf{q}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}), (\mathbf{p}', \mathbf{q}') \in \mathcal{C}(\Delta'_{\tau_2,\rho}).\ (\mathbf{x}, \mathbf{y}) = ((\mathbf{p}, \mathbf{p}')^{¢}, (\mathbf{q}, \mathbf{q}')^{¢})$$

Using Lemma 31 we can directly express a relation between cost-full pairs based on the relations between their cost-full components.

Corollary 2

$$\mathcal{C}(\Delta'_{(\tau_1,\tau_2),\rho}) = \{((\mathbf{x_1}, \mathbf{x_2})^{¢}, (\mathbf{y_1}, \mathbf{y_2})^{¢}) \mid (\mathbf{x_1}, \mathbf{y_1}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}) \land (\mathbf{x_2}, \mathbf{y_2}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})\}$$

$[\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}$

In the sequel we abbreviate $\mathbf{x_1} :^{¢} \ldots :^{¢} \mathbf{x_n} :^{¢} ([\,], 0)$ by $[\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}$.

Lemma 32 Let τ type and ρ such that all type variables in τ are in its domain. Then

$$(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{[\tau],\rho})$$
$$\Leftrightarrow$$
$$\exists n \in \mathbb{N}, (\mathbf{x_1}, \mathbf{y_1}), \ldots, (\mathbf{x_n}, \mathbf{y_n}) \in \mathcal{C}(\Delta'_{\tau,\rho}), c \in \mathbb{Z}.$$
$$(\mathbf{x}, \mathbf{y}) = (c \triangleright [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}, c \triangleright [\mathbf{y_1}, \ldots, \mathbf{y_n}]^{¢})$$

Using Lemma 32 we can define a relation between cost-full lists based on cost-full components.

Corollary 3 For all types τ and ρ such that all type variables in τ are in its domain it holds that

$$\mathcal{C}(\Delta'_{[\tau],\rho}) = \{(c \triangleright [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}, c \triangleright [\mathbf{y_1}, \ldots, \mathbf{y_n}]^{¢}) \mid c \in \mathbb{Z} \land n \in \mathbb{N} \land (\mathbf{x_1}, \mathbf{y_1}) \ldots (\mathbf{x_n}, \mathbf{y_n}) \in \mathcal{C}(\Delta'_{\tau,\rho})\}$$

$$\begin{array}{llll}
\Delta^{¢}_{\alpha,\rho} & = \mathcal{C}(\rho(\alpha)) & \Delta^{¢}_{[\tau],\rho} & = \mathit{lift}^{¢}_{[]}(\Delta^{¢}_{\tau,\rho}) \\
\Delta^{¢}_{Nat,\rho} & = id_{\mathcal{C}(\mathbb{N})} & \Delta^{¢}_{(\tau_1,\tau_2),\rho} & = \mathit{lift}^{¢}_{(,)}(\Delta^{¢}_{\tau_1,\rho}, \Delta^{¢}_{\tau_2,\rho})
\end{array}$$

$$\Delta^{¢}_{\tau_1\to\tau_2,\rho} = \{(\mathbf{f},\mathbf{g}) \mid \mathit{cost}(\mathbf{f}) = \mathit{cost}(\mathbf{g}) \wedge \forall (\mathbf{x},\mathbf{y}) \in \Delta^{¢}_{\tau_1,\rho}.\ (\mathbf{f}\ ¢\ \mathbf{x}, \mathbf{g}\ ¢\ \mathbf{y}) \in \Delta^{¢}_{\tau_2,\rho}\}$$

where

$$\mathit{lift}^{¢}_{[]}(\mathcal{C}(\mathcal{R})) = \{(c \rhd [\mathbf{x_1},\ldots,\mathbf{x_n}]^{¢}, c \rhd [\mathbf{y_1},\ldots,\mathbf{y_n}]^{¢}) \mid c \in \mathbb{Z} \wedge n \in \mathbb{N} \wedge (\mathbf{x_1},\mathbf{y_1}) \ldots (\mathbf{x_n},\mathbf{y_n}) \in \mathcal{C}(\mathcal{R})\}$$

$$\mathit{lift}^{¢}_{(,)}(\mathcal{C}(\mathcal{R}_1),\mathcal{C}(\mathcal{R}_2)) = \{((\mathbf{x_1},\mathbf{x_2})^{¢}, (\mathbf{y_1},\mathbf{y_2})^{¢}) \mid (\mathbf{x_1},\mathbf{y_1}) \in \mathcal{C}(\mathcal{R}_1) \wedge (\mathbf{x_2},\mathbf{y_2}) \in \mathcal{C}(\mathcal{R}_2)\}$$

Figure 6.10: Fully cost-lifted logical relation

The alternative descriptions of the liftings of the cost-lifted logical relation allow us to establish an alternative definition of this relation that only relies on cost-full values. We call the resulting relation $\Delta^{¢}_{\cdot,\cdot}$ and give its definition in Figure 6.10. The following lemma shows that the newly defined relation $\Delta^{¢}_{\cdot,\cdot}$ really is the same relation as the cost-lifting of $\Delta'_{\cdot,\cdot}$.

Lemma 33 For all τ and ρ, $\mathcal{C}(\Delta'_{\tau,\rho}) = \Delta^{¢}_{\tau,\rho}$.

Proof. By Lemma 30, Corollaries 2 and 3, definitions of $\Delta^{¢}_{\cdot,\cdot}$ and $\Delta'_{\cdot,\cdot}$, and cost-lifting. □

The benefits of the characterization become apparent in the next section where we regard concrete efficiency assertions based on free theorems. Employing the characterization, the parametricity theorem can be stated slightly differently from Theorem 11.

Corollary 4 (parametricity theorem) If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{fold}}$, then for every ρ, σ_1, σ_2 such that

- for every α in Γ, $\rho(\alpha) \in \mathit{Rel}$, and
- for every $x :: \tau'$ in Γ, $((\sigma_1(x), 0), (\sigma_2(x), 0)) \in \Delta^{¢}_{\tau',\rho}$

we have $([\![t]\!]^{¢}_{\sigma_1}, [\![t]\!]^{¢}_{\sigma_2}) \in \Delta^{¢}_{\tau,\rho}$.

6.4 The Parametricity Theory at Work

In the previous section we established a theory of parametricity that incorporates evaluation costs in the sense that the number of function applications is counted. Based on the cost-lifted version of the logical relation presented in

Figure 6.10 we want to derive free theorems that, besides equivalence in terms of the standard denotational semantics, also provide a statement about the cost difference between two expressions.

The general scheme to derive a free theorem that states the semantic equivalence of two expressions from the parametricity theorem is to unfold the logical relation and to specialize the relations chosen via environment ρ to functions that can be expressed as the semantics of some term. Let us recapitulate the procedure for the type $\alpha \to Nat$ w.r.t. the standard semantics, i.e., without costs. The parametricity theorem (Theorem 1) states that for all sets S_1, S_2, relations $\mathcal{R} \subseteq S_1 \times S_2$ and terms $f :: \alpha \to Nat$ we have

$$([\![f]\!]_\emptyset, [\![f]\!]_\emptyset) \in \Delta_{\alpha\to Nat,[\alpha\mapsto\mathcal{R}]}$$

We unfold the logical relation and gain

$$\forall(\mathbf{x}, \mathbf{y}) \in \mathcal{R}.\ ([\![f]\!]_\emptyset\ \mathbf{x} = [\![f]\!]_\emptyset\ \mathbf{y})$$

As a next step we specialize the relation $\mathcal{R}$ (and thereby the sets S_1 and S_2) to (the graph of) the semantics of a term $g :: \tau_1 \to \tau_2$ with τ_1 and τ_2 arbitrary types. We gain (with the definition of the term semantics):

$$\forall \tau_1, \tau_2, g :: \tau_1 \to \tau_2, x :: \tau_1.\ ([\![f\ x]\!]_\emptyset = [\![f\ (g\ x)]\!]_\emptyset)$$

Unfortunately, if we try to reuse the same scheme starting from the cost-sensitive parametricity theorem (Corollary 4), we encounter difficulties with the specialization of the relation $\mathcal{R}$. The environment ρ maps to cost-free relations. Hence, we cannot directly specialize relation $\mathcal{R}$ to a function that corresponds to the semantics of some term, because the semantics of a term will be a cost-full value. To bridge the gap between the cost-full semantics and the cost-free relation, we define for each cost-full function $\mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2))$ a corresponding cost-free relation based on the $val(\cdot)$-parts of the function. Because it will allow stronger assertions about cost differences (as we will see from the examples to come), we consider mostly partial function graphs.

Definition 43 **(cost-free graph)**

Let S_1, S_2 sets and $\mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2))$. We define by

$$\mathcal{R}^{\mathbf{g}} = \{(val(\mathbf{x}), val(\mathbf{g} \not{c}\ \mathbf{x})) \mid \mathbf{x} \in \mathcal{C}(S_1)\}$$

the *total cost-free graph* of $\mathbf{g}$ and for any given $\mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathcal{C}(S_1)$ with $n \in \mathbb{N} \setminus \{0\}$ we define by

$$\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}} = \{(val(\mathbf{x_1}), val(\mathbf{g} \not{c}\ \mathbf{x_1})), \ldots, (val(\mathbf{x_n}), val(\mathbf{g} \not{c}\ \mathbf{x_n}))\}$$

the *partial cost-free graph* of $\mathbf{g}$ on $\{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$.

To derive free theorems, costs that arise during function application play a central role and we abbreviate these costs as follows.

Definition 44 ($appCost(\mathbf{f},\mathbf{x})$)

Let S_1, S_2 sets, $\mathbf{f} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2))$ and $\mathbf{x} \in \mathcal{C}(S_1)$. The costs that arise when $\mathbf{f}$ is applied to $\mathbf{x}$ are as follows:

$$appCost(\mathbf{f},\mathbf{x}) = cost(\mathbf{f} \not{c}\, \mathbf{x}) - cost(\mathbf{x})$$

Definitions 43 and 44 allow us to formulate the following lemmas and the corollary. These are handy in the actual derivation of cost-sensitive free theorems. The respective proofs are given in Appendix A.3.

Lemma 34

For all S_1, S_2 sets, $\mathbf{f} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2))$ and $\mathbf{x_1}, \mathbf{x_2} \in \mathcal{C}(S_1)$ we have

$$val(\mathbf{x_1}) = val(\mathbf{x_2}) \Rightarrow appCost(\mathbf{f},\mathbf{x_1}) = appCost(\mathbf{f},\mathbf{x_2})$$

Lemma 35

Let $\mathbf{x} \in \mathcal{C}(S_1)$, $\mathbf{y} \in \mathcal{C}(S_2)$ and $n \in \mathbb{N}^+$. Then $(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}})$ if and only if there exist $i \in \{1,\ldots,n\}$ and $c \in \mathbb{Z}$ such that

$$\mathbf{x} = c \triangleright appCost(\mathbf{g},\mathbf{x_i}) \triangleright \mathbf{x_i} \text{ and } \mathbf{y} = c \triangleright (\mathbf{g} \not{c}\, \mathbf{x_i})$$

Corollary 5

Let $\mathbf{x} \in \mathcal{C}(S_1)$, $\mathbf{y} \in \mathcal{C}(S_2)$ and $n \in \mathbb{N}^+$. If $(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}})$, then there exists $i \in \{1,\ldots,n\}$ such that

$$\mathbf{g} \not{c}\, \mathbf{x} = appCost(\mathbf{g},\mathbf{x_i}) \triangleright \mathbf{y}$$

A further property we employ, but without explicitly mentioning it, is that $(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\mathcal{R})$ implies $(c \triangleright \mathbf{x}, c \triangleright \mathbf{y}) \in \mathcal{C}(\mathcal{R})$ for every $c \in \mathbb{Z}$.

By the additional statements just given we derive cost-sensitive free theorems from Corollary 4. As a first example, we derive the theorem for $f :: \alpha \to Nat$.

The cost-sensitive parametricity theorem states

$$\forall \mathcal{R} \in Rel.\ (\llbracket f \rrbracket^{\not{c}}_{\emptyset}, \llbracket f \rrbracket^{\not{c}}_{\emptyset}) \in \Delta^{\not{c}}_{\alpha \to Nat, [\alpha \mapsto \mathcal{R}]}$$

Via unfolding of the logical relation we obtain

$$\forall \mathcal{R} \in Rel, (\mathbf{x},\mathbf{y}) \in \mathcal{C}(\mathcal{R}).\ \llbracket f \rrbracket^{\not{c}}_{\emptyset} \not{c}\, \mathbf{x} = \llbracket f \rrbracket^{\not{c}}_{\emptyset} \not{c}\, \mathbf{y}$$

Now we specialize $\mathcal{R}$ to $\mathcal{R}^{\mathbf{g}}_{\mathbf{x}}$ for some $\mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2))$ and $\mathbf{x} \in \mathcal{C}(S_1)$. Applying Lemma 35 we get

$$\begin{aligned} &\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x} \in \mathcal{C}(S_1). \\ &\quad \llbracket f \rrbracket^{\not{c}}_{\emptyset} \not{c}\, appCost(\mathbf{g},\mathbf{x}) \triangleright \mathbf{x} = \llbracket f \rrbracket^{\not{c}}_{\emptyset} \not{c}\, (\mathbf{g} \not{c}\, \mathbf{x}) \end{aligned}$$

By Lemma 29 we move the extra costs to the top level, get

$$\begin{aligned}&\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x} \in \mathcal{C}(S_1).\\ &\qquad appCost(\mathbf{g}, \mathbf{x}) \triangleright ([\![f]\!]^{¢}_{\emptyset} \text{ ¢ } \mathbf{x}) = [\![f]\!]^{¢}_{\emptyset} \text{ ¢ } (\mathbf{g} \text{ ¢ } \mathbf{x})\end{aligned}$$

and finally, by the specialization of the sets S_1 and S_2 to the semantics of types and the specialization of the function $\mathbf{g}$ to the semantics of a term, and furthermore by the definition of the term semantics, we obtain

$$\begin{aligned}&\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, x :: \tau_1.\\ &\qquad appCost([\![g]\!]^{¢}_{\emptyset}, [\![x]\!]^{¢}_{\emptyset}) \triangleright [\![f\ x]\!]^{¢}_{\emptyset} = [\![f\ (g\ x)]\!]^{¢}_{\emptyset}\end{aligned}$$

Regarding our semantics, the costs that arise when the semantic interpretation of one term is applied to the semantic interpretation of another term are always positive. Whenever a function that is the semantics of a term is applied to an argument, the application has at least cost 1. Hence, the final specialization of our free theorem implies that: If f has type $\alpha \to Nat$ and g is an arbitrary term of function type, then for every appropriately typed x, the terms $f\ x$ and $f\ (g\ x)$ evaluate to the same cost-free semantic value, but $f\ x$ can be calculated with $appCost([\![g]\!]^{¢}_{\emptyset}, [\![x]\!]^{¢}_{\emptyset})$, which is at least 1, less costs than $f\ (g\ x)$. To express this implication in an easy to read manner, we introduce the following relation.

Definition 45 ($\sqsubseteq^{¢}$, $\sqsubset^{¢}$)

Let S set and $\mathbf{x}, \mathbf{y} \in \mathcal{C}(S)$. We say that $\mathbf{x}$ is *as least as efficient as* $\mathbf{y}$, written $\mathbf{x} \sqsubseteq^{¢} \mathbf{y}$, if there exists $c \in \mathbb{N}$ with $c \triangleright \mathbf{x} = \mathbf{y}$. We say $\mathbf{x}$ is *more efficient than* $\mathbf{y}$, written $\mathbf{x} \sqsubset^{¢} \mathbf{y}$, if there exists $c \in \mathbb{N}^{+}$ such that $c \triangleright \mathbf{x} = \mathbf{y}$. We use the notation also to compare (closed) terms: $t_1 \sqsubset^{¢} t_2$ means $[\![t_1]\!]^{¢}_{\emptyset} \sqsubset^{¢} [\![t_2]\!]^{¢}_{\emptyset}$.

With Definition 45 we express a (slightly weaker version of) the assertion we derived above as:

$$f\ x \sqsubset^{¢} f\ (g\ x)$$

In the next subsection, we regard the cost-sensitive free theorems for the remaining types shown in Figure 6.1 and also for the type $[\alpha] \to [\alpha]$. The examples are simple, but useful to show how our theory works and to introduce some extra lemmas to handle pairs and lists. Subsection 6.4.2 shows a more complex application: We investigate the speed-up that is gained via short-cut fusion (Gill et al., 1993).

6.4.1 Simple Examples

Let us consider the cost-sensitive free theorem for type $\alpha \to \alpha \to \alpha$. Therefor, the machinery developed up to now is completely sufficient.

Example 31 $(\alpha \to \alpha \to \alpha)$

Starting from Corollary 4 we derive the cost-sensitive free theorem for $f :: \alpha \to \alpha \to \alpha$.

$$
\begin{array}{l}
\forall \mathcal{R} \in \mathit{Rel}.\ (\llbracket f \rrbracket^{¢}_{\emptyset}, \llbracket f \rrbracket^{¢}_{\emptyset}) \in \Delta^{¢}_{\alpha\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}]} \\
\quad \Rightarrow \quad \{\ \text{definition of } \Delta^{¢}_{\cdot,\cdot}\ (\text{cf. Figure 6.10})\ \} \\
\forall \mathcal{R} \in \mathit{Rel}, (\mathbf{x},\mathbf{y}), (\mathbf{x}',\mathbf{y}') \in \mathcal{C}(\mathcal{R}).\ (\llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ \mathbf{x}\ ¢\ \mathbf{x}', \llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ \mathbf{y}\ ¢\ \mathbf{y}') \in \mathcal{C}(\mathcal{R}) \\
\quad \Rightarrow \quad \{\ \text{specialization of } \mathcal{R}\ \} \\
\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x_1}, \mathbf{x_2} \in \mathcal{C}(S_1). \\
\quad \forall (\mathbf{x},\mathbf{y}), (\mathbf{x}',\mathbf{y}') \in \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\mathbf{x_2}}).\ (\llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ \mathbf{x}\ ¢\ \mathbf{x}', \llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ \mathbf{y}\ ¢\ \mathbf{y}') \in \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\mathbf{x_2}}) \\
\quad \Rightarrow \quad \{\ \text{Lemma 35}\ \} \\
\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x_1}, \mathbf{x_2} \in \mathcal{C}(S_1). \\
\quad (\llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ (\mathit{appCost}(\mathbf{g},\mathbf{x_1}) \rhd \mathbf{x_1})\ ¢\ (\mathit{appCost}(\mathbf{g},\mathbf{x_2}) \rhd \mathbf{x_2}), \\
\qquad \llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_1})\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_2})) \in \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\mathbf{x_2}}) \\
\quad \Rightarrow \quad \{\ \text{Corollary 5}\ \} \\
\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x_1}, \mathbf{x_2} \in \mathcal{C}(S_1).\ \exists i \in \{1,2\}. \\
\quad \mathbf{g}\ ¢\ (\llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ (\mathit{appCost}(\mathbf{g},\mathbf{x_1}) \rhd \mathbf{x_1})\ ¢\ (\mathit{appCost}(\mathbf{g},\mathbf{x_2}) \rhd \mathbf{x_2})) \\
\qquad = \mathit{appCost}(\mathbf{g},\mathbf{x_i}) \rhd (\llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_1})\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_2})) \\
\quad \Leftrightarrow \quad \{\ \text{simplification employing Lemma 29}\ \} \\
\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x_1}, \mathbf{x_2} \in \mathcal{C}(S_1). \\
\quad \exists c \in \{\mathit{appCost}(\mathbf{g},\mathbf{x_1}), \mathit{appCost}(\mathbf{g},\mathbf{x_2})\}. \\
\qquad c \rhd (\mathbf{g}\ ¢\ (\llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ \mathbf{x_1}\ ¢\ \mathbf{x_2})) = \llbracket f \rrbracket^{¢}_{\emptyset}\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_1})\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_2}) \\
\quad \Rightarrow \quad \{\ \text{specialization of g, type and term semantics}\ \} \\
\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t_1 :: \tau_1, t_2 :: \tau_1. \\
\quad \exists c \in \{\mathit{appCost}(\llbracket g \rrbracket^{¢}_{\emptyset}, \llbracket t_1 \rrbracket^{¢}_{\emptyset}), \mathit{appCost}(\llbracket g \rrbracket^{¢}_{\emptyset}, \llbracket t_2 \rrbracket^{¢}_{\emptyset})\}. \\
\qquad c \rhd \llbracket g\ (f\ t_1\ t_2) \rrbracket^{¢}_{\emptyset} = \llbracket f\ (g\ t_1)\ (g\ t_2) \rrbracket^{¢}_{\emptyset}
\end{array}
$$

This certainly means that the evaluation of $g\ (f\ t_1\ t_2)$ is more efficient than the evaluation of $f\ (g\ t_1)\ (g\ t_2)$ and as simplified statement we get:

$$g\ (f\ t_1\ t_2) \sqsubset^{¢} f\ (g\ t_1)\ (g\ t_2)$$

As a second example, we consider $f :: \alpha \to (\alpha, \alpha)$. We should get a very similar result compared to Example 31. But we have to handle pairs and thus the pair-lifting of the logical relation. Therefore, we need similar statements as Lemma 35 and Corollary 5, but concerned with pairs. To set up such statements, we define $\mathbf{mapPair} = \llbracket \mathit{mapPair} \rrbracket^{¢}_{\emptyset}$ for a reasonable implementation of

$mapPair$ in our calculus, as for example

$$\begin{array}{l} mapPair :: (\alpha \to \beta, \gamma \to \delta) \to (\alpha, \gamma) \to (\beta, \delta) \\ mapPair = \lambda fp :: (\alpha \to \beta, \gamma \to \delta).\lambda p :: (\alpha, \gamma). \\ \quad \textbf{case}\ fp\ \textbf{of}\ \{(f, g) \to \textbf{case}\ p\ \textbf{of}\ \{(x, y) \to (f\ x, g\ y)\}\} \end{array}$$

The semantics of $mapPair$ is

$$(\lambda(\mathbf{f}, \mathbf{g}).(\lambda(\mathbf{x}, \mathbf{y}).((\mathit{val}(\mathbf{f}\ \mathbf{x}), \mathit{val}(\mathbf{g}\ \mathbf{y})), \mathit{cost}(\mathbf{f}\ \mathbf{x}) + \mathit{cost}(\mathbf{g}\ \mathbf{y}) + 1), 1), 0)$$

Now we are prepared to state the next lemma and corollary. Up to the end of this subsection we omit the proofs since they are straightforward but lengthy calculations.

LEMMA 36

Let $\mathbf{p} \in \mathcal{C}(S_1 \times S_3)$, $\mathbf{q} \in \mathcal{C}(S_2 \times S_4)$ and $n, m \in \mathbb{N}^+$. Then

$$(\mathbf{p}, \mathbf{q}) \in \mathit{lift}^{¢}_{(,)}(\mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}), \mathcal{C}(\mathcal{R}^{\mathbf{h}}_{\mathbf{y_1},\ldots,\mathbf{y_m}}))$$

if and only if there exist $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots m\}$, and $c \in \mathbb{Z}$ such that

$$\mathbf{p} = c \rhd \mathit{appCost}(\mathbf{mapPair}\ ¢\ (\mathbf{g}, \mathbf{h})^{¢}, (\mathbf{x_i}, \mathbf{y_j})^{¢}) \rhd (\mathbf{x_i}, \mathbf{y_j})^{¢}$$

and

$$\mathbf{q} = c \rhd (\mathbf{mapPair}\ ¢\ (\mathbf{g}, \mathbf{h})^{¢}\ ¢\ (\mathbf{x_i}, \mathbf{y_j})^{¢})$$

COROLLARY 6

Let $\mathbf{p} \in \mathcal{C}(S_1 \times S_3)$, $\mathbf{q} \in \mathcal{C}(S_2 \times S_4)$ and $n, m \in \mathbb{N}^+$. If $(\mathbf{p}, \mathbf{q}) \in \mathit{lift}^{¢}_{(,)}(\mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}), \mathcal{C}(\mathcal{R}^{\mathbf{h}}_{\mathbf{y_1},\ldots,\mathbf{y_m}}))$, then there exist $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ such that

$$\mathbf{mapPair}\ ¢\ (\mathbf{g}, \mathbf{h})^{¢}\ ¢\ \mathbf{p} = \mathit{appCost}(\mathbf{mapPair}\ ¢\ (\mathbf{g}, \mathbf{h})^{¢}, (\mathbf{x_i}, \mathbf{y_j})^{¢}) \rhd \mathbf{q}$$

EXAMPLE 32 ($\alpha \to (\alpha, \alpha)$)

Starting from Corollary 4 we derive the cost-sensitive free theorem for $f :: \alpha \to (\alpha, \alpha)$.

$$\begin{array}{l} \forall \mathcal{R} \in \mathit{Rel}.\ ([\![f]\!]^{¢}_{\emptyset}, [\![f]\!]^{¢}_{\emptyset}) \in \Delta^{¢}_{\alpha \to (\alpha,\alpha),[\alpha \mapsto \mathcal{R}]} \\ \Rightarrow \quad \{ \text{ definition of } \Delta^{¢}_{\cdot,\cdot} \text{ (cf. Figure 6.10) and specialization of } \mathcal{R} \ \} \\ \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x_1} \in \mathcal{C}(S_1). \\ \quad \forall (\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1}}).\ ([\![f]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{x}, [\![f]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{y}) \in \mathit{lift}^{¢}_{(,)}(\mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1}}), \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1}})) \\ \Rightarrow \quad \{ \text{ Lemma 35 } \} \\ \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x_1} \in \mathcal{C}(S_1). \\ \quad ([\![f]\!]^{¢}_{\emptyset}\ ¢\ (\mathit{appCost}(\mathbf{g}, \mathbf{x_1}) \rhd \mathbf{x_1}), [\![f]\!]^{¢}_{\emptyset}\ ¢\ (\mathbf{g}\ ¢\ \mathbf{x_1})) \in \mathit{lift}^{¢}_{(,)}(\mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1}}), \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1}})) \end{array}$$

$$\Rightarrow \left\{ \begin{array}{l} \text{Corollary 6, choice of relations,} \\ \text{definition of } \mathit{lift}^{¢}_{(,)}(\cdot,\cdot)\text{, Lemma 34} \end{array} \right\}$$

$$\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), \mathbf{x_1} \in \mathcal{C}(S_1).$$
$$\mathbf{mapPair} \ ¢ \ (\mathbf{g},\mathbf{g})^{¢} \ ¢ \ ([\![f]\!]^{¢}_{\emptyset} \ ¢ \ (\mathit{appCost}(\mathbf{g},\mathbf{x_1}) \rhd \mathbf{x_1}))$$
$$= \mathit{appCost}(\mathbf{mapPair} \ ¢ \ (\mathbf{g},\mathbf{g})^{¢}, (\mathbf{x_1},\mathbf{x_1})^{¢}) \rhd ([\![f]\!]^{¢}_{\emptyset} \ ¢ \ (\mathbf{g} \ ¢ \ \mathbf{x_1}))$$

$$\Rightarrow \{ \mathit{appCost}(\mathbf{mapPair} \ldots) - \mathit{appCost}(\mathbf{g},\mathbf{x_1}) > 0 \}$$
$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t :: \tau_1.\ f\ (g\ t) \sqsubset^{¢} \mathit{mapPair}\ (g,g)\ (f\ t)$$

Finally, we consider types involving lists. We give two examples, one very simple where the relative efficiency between the two expressions is clear, and one just a bit more complicated, where additional restrictions are necessary to clearly assert one of the compared expressions more efficient than the other. Of course, to handle lists we need to establish the analogues of Lemma 35 and Corollary 5 w.r.t. the list-lifting. We call the appropriate semantic lifting **mapList** and take it to be the semantics of the already known map, given on page 149.

Hence, **mapList** is defined as

$$[\![\lambda g :: \alpha \to \beta.\lambda ys :: [\alpha].\mathbf{lfold}(\lambda x :: \alpha.\lambda xs :: [\beta].(g\ x) : xs, [\,]_{\beta}, ys)]\!]^{¢}_{\emptyset}$$
$$= (\lambda \mathbf{g}.(\lambda \mathbf{ys}.((\mathbf{v_f}\ \mathbf{v_1}) \ ¢ \ ((\mathbf{v_f}\ \mathbf{v_2}) \ ¢ \ \ldots ((\mathbf{v_f}\ \mathbf{v_n}) \ ¢ \ ([\,],1)))),1),0)$$

for $\mathbf{ys} = [\mathbf{v_1}, \ldots, \mathbf{v_n}]$ as input and with $\mathbf{v_f}$ an abbreviation for

$$\mathit{val}([\![\lambda x :: \alpha.\lambda xs :: [\beta].(g\ x) : xs]\!]^{¢}_{[g \mapsto \mathbf{g}, ys \mapsto \mathbf{ys}]}) = \lambda \mathbf{x}.(\lambda \mathbf{xs}.1 \rhd (\mathbf{g}\ \mathbf{x} :^{¢} (\mathbf{xs},0)),1)$$

Lemma 37

We have $(\mathbf{xs},\mathbf{ys}) \in \mathit{lift}^{¢}_{[]}(\mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}))$ if and only if there exist $m \in \mathbb{N}$, $i_1, \ldots, i_m \in \{1, \ldots, n\}$, and $c \in \mathbb{Z}$ such that

$$\mathbf{xs} = c \rhd \mathit{appCost}(\mathbf{mapList} \ ¢ \ \mathbf{g}, [\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢}) \rhd [\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢}$$

and

$$\mathbf{ys} = c \rhd (\mathbf{mapList} \ ¢ \ \mathbf{g} \ ¢ \ [\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢})$$

Corollary 7

If $(\mathbf{xs},\mathbf{ys}) \in \mathit{lift}^{¢}_{[]}(\mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}))$, then there exist $m \in \mathbb{N}$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$ such that

$$\mathbf{mapList} \ ¢ \ \mathbf{g} \ ¢ \ \mathbf{xs} = \mathit{appCost}(\mathbf{mapList} \ ¢ \ \mathbf{g}, [\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢}) \rhd \mathbf{ys}$$

and

$$\mathit{val}([\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢}) = \mathit{val}(\mathbf{xs})$$

Now, we are prepared to state the two examples that involve lists.

Example 33 ($[\alpha] \to \mathit{Nat}$)

Starting from Corollary 4 we derive the cost-sensitive free theorem for $f :: [\alpha] \to \mathit{Nat}$. The definition of $\Delta^{¢}_{\cdot,\cdot}$, the specialization of $\mathcal{R}$ to $\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}$ and Lemma 37 for the specialized relation lead to the following assertion.

$$\begin{aligned}
&\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), n \in \mathbb{N}, \mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathcal{C}(S_1).\\
&[\![f]\!]^{¢}_{\emptyset} \ ¢ \ \mathit{appCost}(\mathbf{mapList} \ ¢ \ \mathbf{g}, [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}) \rhd [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}\\
&\quad = [\![f]\!]^{¢}_{\emptyset} \ ¢ \ (\mathbf{mapList} \ ¢ \ \mathbf{g} \ ¢ \ [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢})
\end{aligned}$$

With specialization of $\mathbf{g}$ to $[\![g]\!]^{¢}_{\emptyset}$ and similar arguments as in the previous examples we obtain:

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \to \tau_2, t :: [\tau_1]. \ f \ t \sqsubset^{¢} f \ (\mathit{map} \ g \ t)$$

Example 34 ($[\alpha] \to [\alpha]$)

Starting from Corollary 4 we derive the cost-sensitive free theorem for $f :: [\alpha] \to [\alpha]$. By the definition of the logical relation, Lemma 37 and Corollary 7 for $\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\ldots,\mathbf{x_n}}$, plus simplification via cost shifting, we get:

$$\begin{aligned}
&\forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2)), n \in \mathbb{N}, \mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathcal{C}(S_1).\\
&\exists m \in \mathbb{N}, i_1, \ldots, i_m \in \{1, \ldots, n\}.\\
&\quad \mathit{appCost}(\mathbf{mapList} \ ¢ \ \mathbf{g}, [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}) \rhd\\
&\qquad\quad (\mathbf{mapList} \ ¢ \ \mathbf{g} \ ¢ \ ([\![f]\!]^{¢}_{\emptyset} \ ¢ \ [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}))\\
&\quad\ = \mathit{appCost}(\mathbf{mapList} \ ¢ \ \mathbf{g}, [\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢}) \rhd\\
&\qquad\quad ([\![f]\!]^{¢}_{\emptyset} \ ¢ \ (\mathbf{mapList} \ ¢ \ \mathbf{g} \ ¢ \ [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}))\\
&\quad \wedge \mathit{val}([\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢}) = \mathit{val}([\![f]\!]^{¢}_{\emptyset} \ ¢ \ [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢})
\end{aligned}$$

In order to continue and derive a statement about the relative efficiency of $\mathit{map} \ g \ (f \ t)$ and $f \ (\mathit{map} \ g \ t)$, for types τ_1, τ_2, function $g :: \tau_1 \to \tau_2$, and list $t :: [\tau_1]$, we would need further information about $\mathit{appCost}(\mathbf{mapList} \ ¢ \ \mathbf{g}, [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢})$ and $\mathit{appCost}(\mathbf{mapList} \ ¢ \ \mathbf{g}, [\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}]^{¢})$. This cannot be provided generally, but a number of useful observations is possible. For example, we know that the elements $\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_m}}$ form a subset of $\{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$, and hence that evaluation of $\mathit{map} \ g \ (f \ t)$ does not incur g-costs on elements other than those already encountered during evaluation of $f \ (\mathit{map} \ g \ t)$, though of course a different selection and multiplicities are possible. Moreover, if we assume that g is equally costly on every element of t, or indeed on every term of type τ_1, then we can reduce the question about the relative efficiency of $\mathit{map} \ g \ (f \ t)$ and $f \ (\mathit{map} \ g \ t)$ to one about the relative length of t and $f \ t$, to which an answer might be known statically by some separate analysis. Also, note that with some extra effort it would even be possible to explicitly get our hands at the existentially quantified m and $i_1, \ldots, i_m$, namely to establish that $[\mathbf{i_1}, \ldots, \mathbf{i_m}] = \mathit{val}([\![f]\!]^{¢}_{\emptyset} \ ¢ \ ([\mathbf{1}, \ldots, \mathbf{n}], 0))$.

6.4.2 Considering *foldr* / *build*

We now consider the first complex example. We obtain a perhaps slightly surprising result, but unfortunately also uncover a weak point of our theory of parametricity. Gill et al. (1993) present a program transformation named short-cut fusion or *foldr* / *build*-rule (cf. Section 3.2).

In our calculus, the *foldr* / *build*-rule says that for arbitrary monotypes τ, τ', every function $g :: (\tau \to \alpha \to \alpha) \to \alpha \to \alpha$, function $k :: \tau \to \tau' \to \tau'$ and term $z :: \tau'$, the value parts of $[\![\mathbf{lfold}(k, z, g\ (:)\ [\,]_\tau)]\!]^{¢}_{\emptyset}$ and $[\![g\ k\ z]\!]^{¢}_{\emptyset}$ are equal. Here $(:)$ describes the cons-operator as a function:

$$(:) = \lambda x :: \alpha.\lambda xs :: [\alpha].x : xs$$

In the sequel, we write $(:^{¢})$ to denote $[\![(:)]\!]^{¢}_{\emptyset}$. The question if the evaluation of $g\ k\ z$ is always as least as efficient as the evaluation of $\mathbf{lfold}(k, z, g\ (:)\ [\,]_\tau)$ arises, and for an answer we can derive a specialization of the cost-sensitive free theorem of g's type, i.e., for type $(\tau \to \alpha \to \alpha) \to \alpha \to \alpha$ with τ a fixed monotype. As long as a type is not polymorphic, the logical relation is the identity relation. Consequently, the logical relation on type τ is fixed to the identity. Additionally to fixing τ, we fix τ', $k :: \tau \to \tau' \to \tau'$ with semantics $[\![k]\!]^{¢}_{\emptyset} = \mathbf{k} = (\mathbf{v_k}, c_k)$ and $z :: \tau'$ with semantics $[\![z]\!]^{¢}_{\emptyset} = \mathbf{z} = (\mathbf{v_z}, c_z)$.

$(:^{¢})$

As in the examples in Subsection 6.4.1, we start from the type specific statement of Corollary 4, i.e., in the current case:

$$\forall \mathcal{R} \in Rel.\ ([\![g]\!]^{¢}_{\emptyset}, [\![g]\!]^{¢}_{\emptyset}) \in \Delta^{¢}_{(\tau\to\alpha\to\alpha)\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}]}$$

and unfold the logical relation to gain

$$\begin{aligned}&\forall \mathcal{R} \in Rel.\ \forall (\mathbf{f}, \mathbf{f}') \in \Delta^{¢}_{\tau\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}]}.\\ &\quad \forall (\mathbf{x}, \mathbf{x}') \in \mathcal{C}(\mathcal{R}).\ ([\![g]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{f}\ ¢\ \mathbf{x}, [\![g]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{f}'\ ¢\ \mathbf{x}') \in \mathcal{C}(\mathcal{R})\end{aligned}$$

The first essential step now is to find an appropriate instantiation for $\mathcal{R}$. Basically we need to introduce $\mathbf{lfold}$, k and z; and hence, in analogy to the reasoning by Gill et al. (1993, Section 3.4), where a partial application of *foldr* is chosen, we choose

$$\begin{aligned}\mathbf{fkz} &= [\![\lambda xs :: [\tau].\mathbf{lfold}(k, z, xs)]\!]^{¢}_{\emptyset}\\ &= (\lambda[\mathbf{v_1}, \ldots, \mathbf{v_n}].1 \rhd ((\mathbf{v_k}\ \mathbf{v_1})\ ¢\ ((\mathbf{v_k}\ \mathbf{v_2}) \ldots ((\mathbf{v_k}\ \mathbf{v_n})\ ¢\ \mathbf{z}) \ldots)), c_k)\end{aligned}$$

fkz

to specialize $\mathcal{R}$ to $\mathcal{R}^{\mathbf{fkz}}$ and obtain, also specializing $\mathbf{x}$ to $[\,]^{¢} = ([\,], 0)$, via Lemma 35 and Corollary 5, that

$$\begin{aligned}&\forall (\mathbf{f}, \mathbf{f}') \in \Delta^{¢}_{\tau\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}^{\mathbf{fkz}}]}.\\ &appCost(\mathbf{fkz}, [\,]^{¢}) \rhd (\mathbf{fkz}\ ¢\ ([\![g]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{f}\ ¢\ [\,]^{¢}))\\ &\ = appCost(\mathbf{fkz}, [\![g]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{f}\ ¢\ [\,]^{¢}) \rhd ([\![g]\!]^{¢}_{\emptyset}\ ¢\ \mathbf{f}'\ ¢\ (\mathbf{fkz}\ ¢\ [\,]^{¢}))\end{aligned}$$

For the application of $\mathbf{fkz}$ to $[]^{¢}$ we get $(c_k + 1) \triangleright \mathbf{z}$ as result and $c_k + c_z + 1$ as application costs. Hence, we can simplify the free theorem to

$$\begin{aligned}
&\forall (\mathbf{f}, \mathbf{f}') \in \Delta^{¢}_{\tau\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}^{\mathbf{fkz}}]}.\\
&c_z \triangleright (\mathbf{fkz} \; ¢ \; ([\![g]\!]^{¢}_{\emptyset} \; ¢ \; \mathbf{f} \; ¢ \; []^{¢}))\\
&\quad = appCost(\mathbf{fkz}, [\![g]\!]^{¢}_{\emptyset} \; ¢ \; \mathbf{f} \; ¢ \; []^{¢}) \triangleright ([\![g]\!]^{¢}_{\emptyset} \; ¢ \; \mathbf{f}' \; ¢ \; \mathbf{z})
\end{aligned}$$

A last and a bit tricky problem must still be solved: What to choose for $\mathbf{f}$ and $\mathbf{f}'$? The answer for $\mathbf{f}'$ is quite clear, it has to be $\mathbf{k}$. The answer for $\mathbf{f}$ seems also quite clear, it has to be $(:^{¢})$. But here we encounter the problem that $((:^{¢}), \mathbf{k}) \in \Delta^{¢}_{\tau\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}^{\mathbf{fkz}}]}$ does not necessarily hold — differently from the cost-free case proved by Gill et al. (1993). The reason is that $\mathbf{k}$ and $(:^{¢})$ might cause different costs.

Let us examine the problem in depth and unfold $\Delta^{¢}_{\tau\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}^{\mathbf{fkz}}]}$. Up to now, unfolding the function-lifting of the logical relation, we dropped the cost equivalence of the functions and thus achieved only an implication instead of an equivalence. If we recall the function-lifting of $\Delta^{¢}_{\cdot,\cdot}$ (cf. Figure 6.10) the simplification becomes clear and as long as function types were only on result positions, and not the type of an argument, the implication was sufficient to establish the kind of assertion about evaluation costs we aim for. Unfortunately, now we have a function of type $\tau \to \alpha \to \alpha$ as input and therefore we need an exact characterization, derived as follows.

$$\begin{aligned}
&(\mathbf{f}, \mathbf{f}') \in \Delta^{¢}_{\tau\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}^{\mathbf{fkz}}]}\\
\Leftrightarrow\quad &\left\{ \text{definition of } \Delta^{¢}_{\cdot,\cdot} \text{ with } \Delta^{¢}_{\tau,\emptyset} = id_{[\![\tau]\!]^{¢}_{\emptyset}} \right\}\\
&cost(\mathbf{f}) = cost(\mathbf{f}') \wedge \forall \mathbf{u} \in \mathcal{C}([\![\tau]\!]^{¢}_{\emptyset}).\\
&\quad cost(\mathbf{f} \; ¢ \; \mathbf{u}) = cost(\mathbf{f}' \; ¢ \; \mathbf{u}) \wedge\\
&\qquad \forall (\mathbf{v}, \mathbf{v}') \in \mathcal{C}(\mathcal{R}^{\mathbf{fkz}}).\ (\mathbf{f} \; ¢ \; \mathbf{u} \; ¢ \; \mathbf{v}, \mathbf{f}' \; ¢ \; \mathbf{u} \; ¢ \; \mathbf{v}') \in \mathcal{C}(\mathcal{R}^{\mathbf{fkz}})\\
\Leftrightarrow\quad &\left\{ \text{definition of } \mathcal{R}^{\mathbf{fkz}}\text{, Lemma 35 and Corollary 5} \right\}\\
&cost(\mathbf{f}) = cost(\mathbf{f}') \wedge \forall \mathbf{u} \in \mathcal{C}([\![\tau]\!]^{¢}_{\emptyset}).\\
&\quad cost(\mathbf{f} \; ¢ \; \mathbf{u}) = cost(\mathbf{f}' \; ¢ \; \mathbf{u}) \wedge \forall \mathbf{v} \in \mathcal{C}([\![[\tau]]\!]^{¢}_{\emptyset}).\\
&\qquad appCost(\mathbf{fkz}, \mathbf{v}) \triangleright (\mathbf{fkz} \; ¢ \; (\mathbf{f} \; ¢ \; \mathbf{u} \; ¢ \; \mathbf{v}))\\
&\qquad = appCost(\mathbf{fkz}, \mathbf{f} \; ¢ \; \mathbf{u} \; ¢ \; \mathbf{v}) \triangleright (\mathbf{f}' \; ¢ \; \mathbf{u} \; ¢ \; (\mathbf{fkz} \; ¢ \; \mathbf{v}))
\end{aligned}$$

To obtain an assertion about the *foldr* / *build*-rule, we cannot vary $\mathbf{k}$, since it should be an arbitrary function. Also, an alteration of $(:^{¢})$ seems to be inappropriate. But to obtain relatedness, we must alter one of the functions. Our choice is to adjust the costs of $(:^{¢})$ in accordance with the (top-level) costs and the nested costs of $\mathbf{k}$. Hence, provisionally, we obtain functions $(:^{¢^{\mathbf{k}}})$ that, on the value part, behave like $(:^{¢})$, but that cause costs like $\mathbf{k}$.

Definition 46 **($(:^{¢\mathbf{k}})$)**

Let S_1, S_2, S_3 sets and $\mathbf{k} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2 \to \mathcal{C}(S_3)))$. We define

$$\begin{aligned}(:^{¢\mathbf{k}}) = (\lambda \mathbf{v_x}.(\lambda \mathbf{v_{xs}}.&\\ &(\mathbf{v_x} : \mathbf{v_{xs}}, cost(val(\mathbf{v_k}\ \mathbf{v_x})\ val(val(\mathbf{fkz})\ \mathbf{v_{xs}}))) \\ &, cost(\mathbf{v_k}\ \mathbf{v_x})), c_k)\end{aligned}$$

Lemma 38

The function $(:^{¢\mathbf{k}})$ fulfills the following two assertions

(a) $\forall \mathbf{x} \in \mathcal{C}(\llbracket \tau \rrbracket^{¢}_{\emptyset}), \mathbf{xs} \in \mathcal{C}(\llbracket [\tau] \rrbracket^{¢}_{\emptyset}).\ val(\mathbf{x} :^{¢\mathbf{k}} \mathbf{xs}) = val((:^{¢})\ ¢\ \mathbf{x}\ ¢\ \mathbf{xs})$

(b) $((:^{¢\mathbf{k}}), \mathbf{k}) \in \Delta^{¢}_{\tau \to \alpha \to \alpha, [\alpha \mapsto \mathcal{R}^{\mathbf{fkz}}]}$

Proof. See Appendix A.3. □

If we instantiate $\mathbf{f}$ by $(:^{¢\mathbf{k}})$ and $\mathbf{f}'$ by $\mathbf{k}$ we obtain

$$\begin{aligned}&c_z \rhd (\mathbf{fkz}\ ¢\ (\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢\mathbf{k}})\ ¢\ [\,]^{¢})) \\ &= appCost(\mathbf{fkz}, \llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢\mathbf{k}})\ ¢\ [\,]^{¢}) \rhd (\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ \mathbf{k}\ ¢\ \mathbf{z})\end{aligned}$$

Because of Lemma 38 *(a)* we can replace $(:^{¢\mathbf{k}})$ by $(:^{¢})$ if we appropriately adjust the costs and obtain

$$\begin{aligned}&(cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢\mathbf{k}})\ ¢\ [\,]^{¢}) - cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})\ ¢\ [\,]^{¢})) \rhd \\ &\quad c_z \rhd (\mathbf{fkz}\ ¢\ (\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})\ ¢\ [\,]^{¢})) \\ &= appCost(\mathbf{fkz}, \llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})\ ¢\ [\,]^{¢}) \rhd (\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ \mathbf{k}\ ¢\ \mathbf{z})\end{aligned}$$

The cost expressions are still quite complicated and cannot straightforwardly be summarized to a single cost that is always non-negative. Obviously the artificial costs on the right-hand side, i.e., the costs of applying **fkz** to a list depend mainly on the length of the list. For the artificial costs on the left-hand side we take the cost difference between two applications of $\llbracket g \rrbracket^{¢}_{\emptyset}$ only differing in the cost behavior of the first argument. Thus, the artificial costs on the left-hand side depend on how often $\llbracket g \rrbracket^{¢}_{\emptyset}$ employs $(:^{¢\mathbf{k}})$ or $(:^{¢})$ (partially or fully). Intuitively, this is reflected in the list length again. Our aim is to separate the costs that g produces into the costs that arise because of (partial or full) applications of its first argument and the costs that arise otherwise (that, because g is polymorphic, cannot rely on a particular first argument).

Via the separation the costs hopefully can be summarized in a way that a simple estimation of the cost difference between left- and right-hand side of the free theorem is possible. In particular, we aim for these simplifications of costs:

- The costs $\mathbf{g}$ causes independently of its first argument are the same for the application to $(:^{\not{c}\mathbf{k}})$ and to $(:^{\not{c}})$. Thus, these costs vanish on the left-hand side.
- The costs $\mathbf{g}$ causes when applied to $(:^{\not{c}\mathbf{k}})$ because of the applications of $(:^{\not{c}\mathbf{k}})$ are intuitively the same as the application costs of $\mathbf{fkz}$ to the list produced by $\mathbf{g}$ (minus the costs of $\mathbf{z}$). Thus, these costs from the left- and the right-hand side should somehow vanish as well.
- The costs $\mathbf{g}$ causes when applied to $(:^{\not{c}})$ because of the applications of $(:^{\not{c}})$ are on the first sight linear in the length of the list $\mathbf{g}$ produces.[2] Because of polymorphism, $\mathbf{g}$ seems to have no other possibility than applying $(:^{\not{c}})$ n-times to produce its output list. Thus, we might leave these costs as last costs in the equation and get extra costs linear in the list length for the expression on the left-hand side of the free theorem (i.e., the expression not transformed via *foldr* / *build*).

To separate the costs $\mathbf{g}$ causes by (applications of) its first argument and other costs, we introduce costless versions of functions. These versions of functions behave on the value side exactly as the original function, but cause no costs at all.

Definition 47 (costless function version)

Let $\mathbf{f} \in \mathcal{C}(S_1 \to \ldots \to \mathcal{C}(S_n))$. The function

$$\mathbf{f}^{0\ldots0} = (\lambda\mathbf{v_1}.(\ldots(\lambda\mathbf{v_n}.(val(val(\ldots val(val(\mathbf{f})\ \mathbf{v_1})\ldots)\ \mathbf{v_n}),0),\ldots,0)$$

where $\mathbf{f}$ is indexed by n zeros, is called a *costless version* of $\mathbf{f}$.

Because $cost(\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}\mathbf{k}})^{000} \not{c}\ [\,]^{\not{c}})$ is equal to $cost(\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}})^{000} \not{c}\ [\,]^{\not{c}})$ we can rewrite the free theorem as follows.

$$\begin{aligned}
&(cost(\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}\mathbf{k}}) \not{c}\ [\,]^{\not{c}}) - cost(\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}\mathbf{k}})^{000} \not{c}\ [\,]^{\not{c}})\\
&- (cost(\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}}) \not{c}\ [\,]^{\not{c}}) - cost(\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}})^{000} \not{c}\ [\,]^{\not{c}}))) \rhd\\
&\quad c_z \rhd (\mathbf{fkz} \not{c}\ (\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}}) \not{c}\ [\,]^{\not{c}}))\\
&= appCost(\mathbf{fkz}, \llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ (:^{\not{c}}) \not{c}\ [\,]^{\not{c}}) \rhd (\llbracket g\rrbracket^{\not{c}}_{\emptyset} \not{c}\ \mathbf{k} \not{c}\ \mathbf{z})
\end{aligned}$$

Next, we want to compare the first cost difference on the left-hand side and the application costs of $\mathbf{fkz}$ on the right-hand side. Therefore, without loss of generality, we consider $\llbracket g\rrbracket^{\not{c}} \not{c}\ (:^{\not{c}}) \not{c}\ [\,]^{\not{c}}$ to produce a list of length n.

Convention 14

Up to the end of this subsection, we assume, for arbitrary, but fixed $n \in \mathbb{N}$ and $c_l \in \mathbb{Z}$,

$$\llbracket g\rrbracket^{\not{c}} \not{c}\ (:^{\not{c}}) \not{c}\ [\,]^{\not{c}} = ([\mathbf{v_1},\ldots,\mathbf{v_n}], c_l)$$

[2] Actually, polymorphism does not guarantee the costs to be linear in the length of the list. Hence, our intuition will prove false. We discuss the problem later on.

Before we regard the costs that **g** causes depending on the costs of its first argument, we calculate the application costs of **fkz** that are artificially added on the right-hand side of the free theorem. As obvious from the semantics of **fkz** given on page 167, for the list $([\mathbf{v_1}, \ldots, \mathbf{v_n}], c_l)$ the application costs of **fkz** are

$$1 + c_k + c_z + \sum_{i=1}^{n} cost(\mathbf{v_k}\ \mathbf{v_i}) + \sum_{i=1}^{n} cost(\mathbf{v'_i}\ \mathbf{v''_i}) \tag{6.2}$$

where

$$\mathbf{v'_i} = \mathbf{v_k}\ \mathbf{v_i} \text{ and } \mathbf{v''_i} = cost(val(\mathbf{v_k}\ \mathbf{v_i})\ val((\mathbf{v_k}\ \mathbf{v_{i+1}}) \ ¢\ (\ldots \ ¢\ ((\mathbf{v_k}\ \mathbf{v_n}) \ ¢\ \mathbf{z}))))$$

To set off the first cost difference on the left-hand side against the just calculated application costs, we have to verify how often **g** applies its first argument (partially or fully) to build up the list. On the first look, it should apply it as often as **fkz** applies **k**. The reason seems to be obvious, but is actually faulty: Because **g** is polymorphic, it can, given $(:^{¢\mathbf{k}})$ and $[\,]^{¢}$ as arguments, only employ the given constructors to build a list. To build a list of length n it has to apply the given $(:^{¢\mathbf{k}})$ n times. Applications that do not directly contribute to the generation of the output list are useless, because **g** cannot inspect the (in **g**'s view) polymorphic output. But, unfortunately, nothing hinders **g** to enforce useless applications.

EXAMPLE 35

Regard the function

$$g = \lambda k :: Nat \to \alpha \to \alpha . \lambda z :: \alpha . \mathbf{case}\ [k\ 42\ z]\ \mathbf{of}\ \{[\,] \to z; x : xs \to k\ 1\ z\}$$

If we apply g to $(:)$ and $[\,]_\alpha$ the function $(:)$ is applied twice, even if only once to produce the final result. The first application does not contribute to the overall result, but artificially causes extra costs.

EXAMPLE 36

Consider the function

$$\begin{aligned} &g :: (Int \to \alpha \to \alpha) \to \alpha \to \alpha \\ &g = \lambda k . \lambda z . \mathbf{case}\ [k\ 42]\ \mathbf{of}\ \{[\,] \to z; (\lambda x . x\ (x\ (x\ z)))\ (k\ 1)\} \end{aligned}$$

We get $\mathbf{g}\ ¢\ (:^{¢})\ ¢\ [\,]^{¢} = ([\mathbf{1}, \mathbf{1}, \mathbf{1}], c)$ for some c, but **k** is applied unnecessarily to an element not appearing in the list.

The above examples show that we cannot generally offset the application costs of **fkz** against the costs that $(:^{¢\mathbf{k}})$ enforces when applied inside of **g**. Example 35 might give the impression that it suffices to limit the applications of $(:^{¢\mathbf{k}})$ in **g** to the length of the list **g** produces. Example 36 disproves the impression: Only two of the partial applications of $(:^{¢\mathbf{k}})$ produce a list of length three and one still is an unnecessary application. We can try to ignore unnecessary applications as long as $(:^{¢\mathbf{k}})$ is not employed more than n times. But this handling is only valid if the application costs for $(:^{¢\mathbf{k}})$ are independent of the argument given to it. If for example $k\ 42$ is way more expensive than $k\ 1$ in Example 36, we cannot estimate the cost difference we want in our free theorem.

The above discussion has shown that in general *foldr* / *build* does not necessarily speed up a program. Hence, we must restrict g to guarantee a speed up. Fortunately, in practice the restriction will not matter because it essentially says "g is not allowed to apply its first argument unnecessarily". Thus, we call sufficiently restricted functions "reasonable". We provide two definitions of reasonable. The first definition (weakly reasonable) requires the first argument to g to cause equal costs when applied to different arguments. The second definition (strongly reasonable) relaxes that restriction, but in return complicates the conditions on g. Note, that both definitions establish an upper and a lower bound for the costs. The lower bound is eligible, because the polymorphism of g intuitively guarantees that the list can only be built up using the operator given to g as first argument. Unfortunately, we were not able to state a somehow abstract definition for "reasonable", thus we will really just require g to cause costs that are easy to handle when we further simplify the free theorem.

DEFINITION 48
(weakly reasonable)

The function $g :: (\tau \to \alpha \to \alpha) \to \alpha \to \alpha$ with $val(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})\ ¢\ []^{¢}) = [\mathbf{v_1}, \ldots, \mathbf{v_n}]$ is *weakly reasonable* if for every $\mathbf{k} \in \mathcal{C}(\llbracket \tau \rrbracket^{¢}_{\emptyset} \to \mathcal{C}(S \to \mathcal{C}(S)))$ with

- $\exists c.\ \forall \mathbf{x} \in \llbracket \tau \rrbracket^{¢}_{\emptyset}.\ appCost(\mathbf{k}, \mathbf{x}) = c$
- $\exists c.\ \forall \mathbf{x} \in \llbracket \tau \rrbracket^{¢}_{\emptyset}.\ \forall \mathbf{y} \in \mathcal{C}(S).\ appCost(\mathbf{k}\ ¢\ \mathbf{x}, \mathbf{y}) = c$

there exists an $i \in \{0, \ldots, n\}$ such that for arbitrary $\mathbf{x} = (\mathbf{v}, c) \in \llbracket \tau \rrbracket^{¢}_{\emptyset}$,

$$\begin{aligned} &cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢\mathbf{k}})\ ¢\ []^{¢}) - cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})^{000}\ ¢\ []^{¢}) \\ &= cost(\mathbf{k}) + i \cdot cost(val((:^{¢\mathbf{k}}))\ \mathbf{v}) + n \cdot cost(val((:^{¢\mathbf{k}})\ ¢\ \mathbf{x})\ []) \end{aligned}$$

DEFINITION 49
(strongly reasonable)

The function $g :: (\tau \to \alpha \to \alpha) \to \alpha \to \alpha$ with $val(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})\ ¢\ []^{¢}) = [\mathbf{v_1}, \ldots, \mathbf{v_n}]$ is *strongly reasonable* if for every $\mathbf{k} \in \mathcal{C}(\llbracket \tau \rrbracket^{¢}_{\emptyset} \to \mathcal{C}(S \to \mathcal{C}(S)))$,

- $$\begin{aligned} &cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢\mathbf{k}})\ ¢\ []^{¢}) - cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})^{000}\ ¢\ []^{¢0}) \\ &\leqslant cost(\mathbf{k}) + cost((val((:^{¢\mathbf{k}}))\ \mathbf{v_1})\ ¢\ (\ldots\ ((val((:^{¢\mathbf{k}}))\ \mathbf{v_n})\ ¢\ []^{¢}))) \end{aligned}$$
- $cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:)\ ¢\ []^{¢}) - cost(\llbracket g \rrbracket^{¢}_{\emptyset}\ ¢\ (:^{¢})^{000}\ ¢\ []^{¢0}) > n$

For reasonable functions g we can simplify the already known free theorem further. Either assuming g to be weakly reasonable and $\mathbf{k}$ such that its application costs are independent of the arguments given to it, or assuming g to be strongly reasonable and not constraining $\mathbf{k}$, we can simplify the already known assertion

$$\begin{aligned}
&(cost(\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢\mathbf{k}}) \;¢\; [\,]^{¢}) - cost(\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢\mathbf{k}})^{000} \;¢\; [\,]^{¢})\\
&- (cost(\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢}) \;¢\; [\,]^{¢}) - cost(\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢})^{000} \;¢\; [\,]^{¢}))) \rhd\\
&\quad c_z \rhd (\mathbf{fkz} \;¢\; (\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢}) \;¢\; [\,]^{¢}))\\
&= appCost(\mathbf{fkz}, \llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢}) \;¢\; [\,]^{¢}) \rhd (\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; \mathbf{k} \;¢\; \mathbf{z})
\end{aligned}$$

By Definition 48 or 49 we overestimate the first cost difference on the left-hand side by the application costs of $\mathbf{fkz}$ to a list of length n (minus the costs of $\mathbf{z}$). Furthermore, by the same definitions, we underestimate the second cost difference on the left-hand side by n. Thereby we obtain

$$\begin{aligned}
&(appCost(\mathbf{fkz}, \llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:) \;¢\; [\,]^{¢}) - n) \rhd (\mathbf{fkz} \;¢\; (\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢}) \;¢\; [\,]^{¢}))\\
\sqsupseteq^{¢}\; &appCost(\mathbf{fkz}, \llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢}) \;¢\; [\,]^{¢}) \rhd (\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; \mathbf{k} \;¢\; \mathbf{z})
\end{aligned}$$

Finally, we cancel the remaining artificial application costs and move n to the right-hand side to get the cost estimation

$$(\mathbf{fkz} \;¢\; (\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; (:^{¢}) \;¢\; [\,]^{¢})) \sqsupseteq^{¢} n \rhd (\llbracket g\rrbracket^{¢}_{\emptyset} \;¢\; \mathbf{k} \;¢\; \mathbf{z})$$

So, in the end we gain that under appropriate restrictions, $foldr$ / $build$ enables a linear speed up in the length of the list $\mathbf{g} \;¢\; (:^{¢}) \;¢\; [\,]^{¢}$ produces. The result is quite promising w.r.t. the applicability of our theory, but the restrictions on g are formulated very brute force. Let us rethink what we want to express. Essentially, we want to state that g applies its first argument only to produce the final result. Since g is polymorphic, parametricity should tell us that the production of a list of length n needs at least n total applications of g's first argument and at least one partial such application. Furthermore, not more than n partial and n total applications should be necessary. Hence, at least for weakly reasonable functions it should suffice to count applications of g's first argument. A possible way to count applications might be "tick"-versions of functions, an extension of the concept of costless functions.

Definition 50 (tick-version)

Let $\mathbf{f} \in \mathcal{C}(S_1 \to \ldots \to \mathcal{C}(S_n))$ and $c_1, \ldots, c_n \in \{0,1\}$. The function

$$\mathbf{f}^{c_1 \ldots c_n} = (\lambda\mathbf{v_1}.(\ldots(\lambda\mathbf{v_n}.(val(val(\ldots val(val(\mathbf{f})\;\mathbf{v_1})\ldots)\;\mathbf{v_n}), c_n), \ldots, c_1)$$

is called a *tick-version* of $\mathbf{f}$.

The concrete usage of tick-versions is illustrated best by example.

Example 37 Let $addApps :: (Nat \to Nat) \to Nat \to Nat$ with

$$addApps = \lambda f :: Nat \to Nat.\lambda x :: Nat.f\ x + f\ x$$

as semantics of $addApps$ we get

$$\mathbf{addApps} = (\lambda \mathbf{v_f}.(\lambda \mathbf{v_x}.((\mathbf{v_f}\ \mathbf{v_x}) +^{¢} (\mathbf{v_f}\ \mathbf{v_x})), 1), 0)$$

We can calculate how often a function $\mathbf{f} \in \mathcal{C}(\mathbb{N} \to \mathcal{C}(\mathbb{N}))$ given to $\mathbf{addApps}$ as first argument is applied in $\mathbf{addApps}$ for a specific second input $\mathbf{x}$ by

$$cost(\mathbf{addApps}\ ¢\ \mathbf{f}^{01}\ ¢\ \mathbf{x}) - cost(\mathbf{addApps}\ ¢\ \mathbf{f}^{00}\ ¢\ \mathbf{x})$$

For the example, we take $\mathbf{f} = (\lambda \mathbf{x}.(\mathbf{5}, 3), 1)$ and $\mathbf{x} = (\mathbf{3}, 0)$. As tick-versions we obtain

$$\mathbf{f}^{00} = (\lambda \mathbf{x}.(\mathbf{5}, 0), 0)$$
$$\mathbf{f}^{01} = (\lambda \mathbf{x}.(\mathbf{5}, 1), 0)$$

and thereby we have

$$cost(\mathbf{addApps}\ ¢\ \mathbf{f}^{01}\ ¢\ \mathbf{x}) = 3$$
$$cost(\mathbf{addApps}\ ¢\ \mathbf{f}^{00}\ ¢\ \mathbf{x}) = 1$$

which means that our concrete $\mathbf{f}$ is applied twice when we hand it to $\mathbf{addApps}$ as first argument and give $\mathbf{x}$ as second argument.

Note that in general the number of applications really depends on the concrete functions, even if not so in Example 37. Consider for example

$$h = \lambda f :: Nat \to Nat.\lambda x :: Nat.\mathbf{case}\ x\ \mathbf{of}\ \{0 \to 0; \_ \to f\ x + f\ x\}$$

or

$$h = \lambda f :: Nat \to Nat.\lambda x :: Nat.\mathbf{case}\ f\ 1\ \mathbf{of}\ \{0 \to 0; \_ \to f\ x + f\ x\}$$

That the number of applications is independent of the concrete input holds only for sufficiently polymorphic functions, as g is. Furthermore, counting is only correct if the function, that applies the function we count, properly propagates application costs. All functions that arise as the semantics of a term do so.

That said, parametricity should provide the tools to prove the next proposition that much more directly corresponds to a pure counting of function applications than our original definition.

Let $\mathbf{g} = [\![g]\!]^{¢}_{\emptyset}$ with $g :: (\tau \to \alpha \to \alpha) \to \alpha \to \alpha$ where τ arbitrary but fixed and $\mathbf{g} ¢ (:^{¢}) ¢ []^{¢} = [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}$ for $n \in \mathbb{N}$. If

$$cost(\mathbf{g} ¢ (:^{¢})^{010} ¢ []^{¢}) - cost(\mathbf{g} ¢ (:^{¢})^{000} ¢ []^{¢}) \leqslant n$$
$$cost(\mathbf{g} ¢ (:^{¢})^{001} ¢ []^{¢}) - cost(\mathbf{g} ¢ (:^{¢})^{000} ¢ []^{¢}) = n$$

then g is weakly reasonable.

Proposition 1 (weakly reasonable function)

Unfortunately, our theory is not capable to prove Proposition 1. The problem is that we do not get a grip on nested costs. Thus, with the current theory of parametricity, we must stick to the original definition of a weakly reasonable function. For strongly reasonable functions a similar statement as Proposition 1 should be possible, but again our theory is too weak to prove it.

Irrespective of the restriction we encountered, our theory was able to reveal cases where *foldr* / *build* slows down program execution. By informal arguments we also found out, that those adverse instances of the transformation never occur for functions that do not calculate values they never use for their output, hence, hopefully for all functions that appear in practice.

6.5 Summary

We investigated in how far free theorems can be enriched to allow assertions about evaluation costs. For the study, we chose a very simple λ-calculus without general recursion, but with structural recursion on lists and natural numbers. As semantics, we employed an instrumented denotational semantics that externalized evaluation costs. In particular, we counted function applications. The semantics is similar to the one of Rosendahl (1989) and cost handling corresponds to call-by-value evaluation.

Based on the instrumented semantics, we set up a theory of relational parametricity. In essence we lifted the standard theory of relational parametricity to cost-full expressions in the way that only expressions that cause equal costs are related. The logical relation we set up allows the parametricity theorem given as Theorem 11.

To increase usability of our theory, we reformulated the logical relation and the parametricity theorem in a fully cost-lifted style (cf. Corollary 4). We employed the fully cost-lifted formulation of our theory to investigate in how far we can set up useful assertions about evaluation costs via free theorems. For simple examples the results were quite promising.

Finally, we applied our theory to explore the speed up of a program when it becomes transformed via the *foldr* / *build*-rule. We found out that there are situations where the transformed program might run even slower than the original one. But those cases, though they are possible not only in a call-by-value calculus but also in Haskell with seq, do not matter in practice. They require the forced evaluation of an expression that certainly does not influence the overall result of the program. In all practically relevant cases, $foldr$ / $build$ guarantees a speed up that is linear in the length of the list whose construction is omitted by the program transformation.

Unfortunately, the $foldr$ / $build$ example also revealed a weakness of our theory of parametricity. We were not able to give a simple criterion for the requirements we needed to guarantee a speed up. In particular, the theory seems to be of limited help when functions with different application costs need to be put in relation. Thus, the handling of higher-order functions is not satisfactory.

6.6 Outlook

There are several ways to extend or modify the presented results. Firstly, we might head for a more realistic cost measure. We can add costs to evaluation steps different than function application. Liu and Gómez (2001) show that already within the possibilities of a denotational semantics as ours, suitable choices for costs allow realistic runtime approximations. Secondly, we could extend our theory to a more powerful calculus, e.g. we could add general recursion. For such an extension, the work of van Stone (2003) might provide useful ideas. Thirdly, we might revise our theory to allow a better handling of higher-order functions. In particular, we could head for a theory that allows to relate functions with different application costs (for related arguments). At the moment, we do not see how such a theory should look like, but intuition says that parametricity should allow stronger statements than the ones we can derive at the moment. In particular, Proposition 1 should be provable.

Besides developing our theory further, we could automatize the derivation of cost-sensitive free theorems. We could try to extend the theorem generator of Böhme (2007) that is available at http://www-ps.iai.uni-bonn.de/ft/ to generate cost-sensitive free theorems.

Last, but not least, we could take a look at other free theorem based program transformations than $foldr$ / $build$ and investigate in how far and under which conditions these transformations speed up program execution.

We could also consider a different evaluation strategy. But while an investigation for call-by-name is of very limited interest (the strategy is usually not employed in implementations of programming languages), an investigation for call-by-need will be much more complicated than the one for call-by-value. In a call-by-need setting the instrumented semantics cannot be completely compositional. We must somehow propagate if an expression is already evaluated or not. For example, we might try to encode the state of evaluation of expressions in a parameter that is attached to the semantic function $[\![\cdot]\!]_{\cdot,\cdot}$.

Part III

Conclusion

Chapter 7

Conclusion

The main goal of this thesis was to *extend the theory of parametricity to boost its usability w.r.t. real-world programming languages*. In particular, our interest was to increase applicability of free theorems.

To reach this goal,

- we considered the impact of typical programming features on parametricity results
- we extended the theory of parametricity to incorporate runtime assertions,
- we provided two web interfaces to make (parts of) our theory applicable virtually for free.

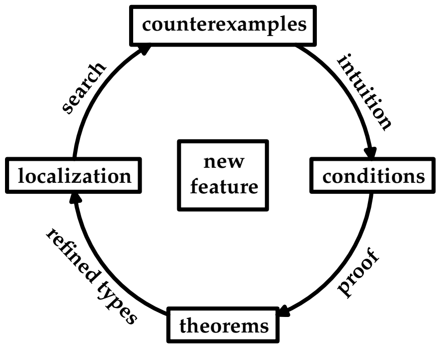
7.1 Consideration of Programming Features

As already explained in the introduction and described closer in the formal background section, free theorems were originally considered in a very simple λ-calculus and do not directly carry over to the calculi that real-world programming languages rely on. Our focus was in particular on functional programming languages, even more concrete on the functional language Haskell. Here in particular

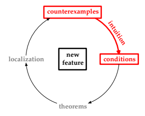
- general recursion and
- forced strict evaluation

were the features of interest.

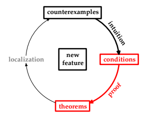
As already explained in the introduction (Section 1.2), the influence of a new programming feature on free theorems can be investigated in four phases. We picture the phases by a circle:



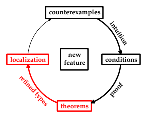
The different steps are described as follows.



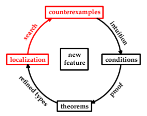
manual search for counterexamples (intuition) We consider a calculus where a theory of parametricity already is established and extend this calculus by the programming feature we want to investigate. Having done so, we try to find examples where the original theory yields incorrect theorems. Those counterexamples to the theory serve as a starting point to extract suitable conditions or alterations that lead the way to an enhanced theory.



formalization of the enhanced theory (proof) Having an idea about how a correct theory of parametricity might look like in the calculus with the new feature, we formally prove whether our intuition is correct. Doing so, we establish an appropriate parametricity theorem.



localization of the influence (refined types) The new feature we consider will weaken the assertions we can derive via the parametricity theorem. The restrictions are only necessary if we really use the feature. Hence, we can relax restrictions if we are able to incorporate knowledge about use or disuse of the new feature in the theory. Since the theory is based only on types, we need to refine typing.



automatic generation of counterexamples (search) Although, after the first three steps we have a fully fledged formal theory of parametricity for the calculus that includes the newly investigated feature, and we already reduced the influence of this feature via localization, still one step seems beneficial. For practical use it is interesting and helpful to see why extra conditions on free theorems arise. This knowledge may increase understanding of the theory. Moreover, for some types, restrictions might be superfluous. This could be discovered by an exhaustive search for examples in which the restrictions are really necessary. Those examples, i.e., free theorems that are incorrect if one of the newly introduced restrictions is relaxed, we call (according to the nomenclature in phase one) counterexamples.

According to the four phase scheme, we pushed the development one phase further for the feature of general recursion and the feature of forced strict

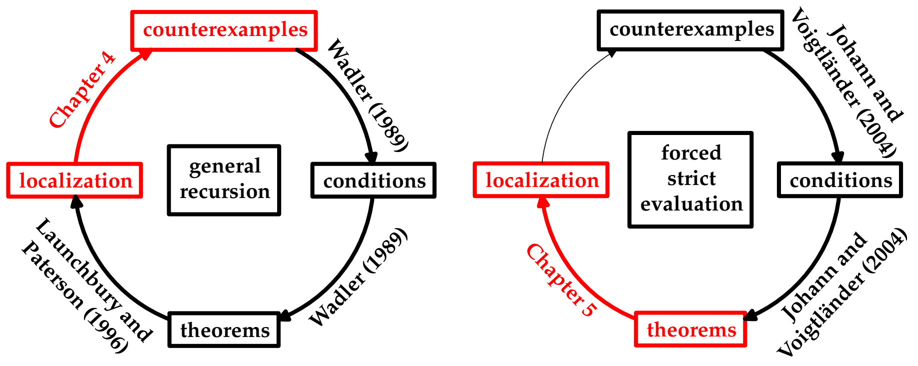


Figure 7.1: Investigation of program features (contributions of this thesis in red)

evaluation. Figure 7.1 graphically summarizes the progress we made. It also points to preceding works.

With the theoretical developments, we hope to supply good tools for formal proofs of program transformations. Free theorems act as verification tool of equational reasoning steps in many contexts. Investigating the restrictions on the theorems in the settings they actually are applied in, and hinting to the pitfalls and corner cases, will help to correctly use free theorems. The additional supply with automatic theorem generators (as we provide it for a calculus with forced strict evaluation and refined types) and counterexample generators (as we provide it for a calculus with general recursion), eases the application of free theorems and guarantees correctness of the results on a formal basis.

7.2 Parametricity Enables Efficiency Assertions

Aside from investigating free theorems in lambda calculi with additional features that are available in real-world functional programming languages, our interest was to widen the view on the kind of assertion free theorems may yield. For that purpose, we studied how to incorporate runtime assertions into free theorems. Those assertions are in particular of interest concerning program transformations that are verified via free theorems. It would be beneficial to know if a program transformation, that may be implemented in a compiler, speeds up a program. For a very simple calculus, we enriched the theory of parametricity such that free theorems with statements about evaluation costs can be derived. Employing the theory we were able to formally consider simple examples. We also employed the theory for a detailed investigation of the *foldr* / *build* program transformation that is implemented in the Glasgow Haskell Compiler (GHC). At least in a call-by-value setting, we found criteria where a speed-up is guaranteed and we could also argue that the speed-up is linear in the length of the list, whose construction is omitted.

Certainly, the theory can be improved and considerations should be made in calculi with more features. Nevertheless, we showed that extra information about evaluation costs can be derived via free theorems. As already mentioned, this information is particularly helpful to estimate the speed-up a program transformation may bring.

Part IV

Appendix

Appendix A

Proofs

In this chapter, we present proofs that are omitted in the thesis' main part or only sketched there. We always present the entire proofs, in particular the parts presented already in the thesis' main part are repeated here.

A.1 Proofs from Chapter 4

Proof of Theorem 6 on page 81 (correctness of $\mathrm{TermFind}$**)**

Proof. The proof splits into two parts. First we show that $\Gamma \vdash t :: \tau$ does not hold in $\lambda^{\alpha}_{\mathbf{fix}^*}$. Therefor, we prove that every term t returned by $\mathrm{TermFind}$ contains a subterm $\bot_{\tau'}$ with $\Gamma_{\mathcal{T}} \vdash \tau' \notin \mathsf{Pointed}$. If this is the case we have immediately that $\Gamma \vdash t :: \tau$ does not hold, since (Fix) is the only rule introducing **fix**, and hence $\bot_{\tau'}$, and we can only use it if τ' is pointed. Since $\Gamma_{\mathcal{T}}$ does not change during a type derivation, pointedness of types does not change during a derivation.

If $\mathrm{TermFind}$ terminates successfully it employs (Bottom) that introduces $\bot_{\tau'}$ on an unpointed type τ', or, if not, the algorithm uses (Bottom$\rightarrow$') at some step switching to the second phase and that way $\bot_{\tau'}$ for τ' unpointed is introduced, too. Knowing that at least one of (Bottom) and (Bottom$\rightarrow$') is applied whenever the algorithm returns a term, it suffices to show that the subterm $\bot_{\tau'}$ introduced by one of the mentioned rules is propagated by every rule of the first phase. Propagation is easily seen because the term at the right-hand side of every rule remains either unchanged as part of the new term on the right-hand side or with some variables substituted. Since $\bot_{\tau'}$ does not contain any variable it is always propagated unchanged.

The second part of the proof is to show that $\Gamma^* \vdash t :: \tau$ holds. We relate every rule of $\mathrm{TermFind}$ to a rule or a rule sequence of the typing rules of $\lambda^{\alpha}_{\mathbf{fix}^*}$ that performs the identical term transformation under the assumption that all types are pointed. The pointedness assumption is immediate because during the algorithm type variables in the context never change, and since we start with Γ^* all type variables are always annotated by *. Thus, regarding the definition of the type class Pointed (cf. Figure 4.5), we have pointedness for all types.

The rules (UPDROP), (NDROP) and (UDROP) perform only context extensions and hence can be skipped. (ABS), (LEFT) and (RIGHT) are present in both rule systems. In the case (WRAP) we use (CONS) with the additional premise (NIL) as corresponding rule. For all following rules we take $\Gamma \vdash \bot_\tau :: \tau$ as additional premise, since we can derive it for every τ under all contexts Γ:

$$\cfrac{\cfrac{\Gamma, x :: \tau \vdash x :: \tau}{\Gamma \vdash (\lambda x :: \tau . x) :: \tau \to \tau}}{\Gamma \vdash \bot_\tau :: \tau}$$

The additional premise enables us to relate (BOTTOM) with (FIX) and (PAIR$_1$), (PAIR$_2$) both with (PAIR).

In the third phase (PAIR′) corresponds to (PAIR) and (EITHER′) to (LEFT), both with $\bot_\tau$ of the appropriate type as a premise. The remaining rules of the third phase correspond to the axioms of the original typing rules. In the second phase (NAT′), (UNIT′), (LIST′), (PAIR′) and (EITHER′) fit to the respective case-rules with (VAR) as additional premise.

The remaining rules all substitute variables in the term t on the right-hand side. If the substituted variables do not appear in t, the proof is immediate. If they appear, they must have been brought into t by some rule already used in the derivation tree. Since all rules only distinguish types, but not term structures, it suffices to generate the term t_v that is substituted for the term variable v, only using the term variables known in the context Γ in the conclusion of the currently examined rule. Thus, in the following cases we focus on the derivation of the t_v in question[1].

(WRAP→′)

$$\cfrac{\cfrac{\Gamma, f :: [\tau_1] \to \tau_2 \vdash f :: [\tau_1] \to \tau_2 \qquad \cfrac{\Gamma, y :: \tau_1 \vdash y :: \tau_1 \qquad \Gamma, y :: \tau_1 \vdash [\,]_{\tau_1} :: [\tau_1]}{\Gamma, y :: \tau_1 \vdash (y : [\,]_{\tau_1}) :: [\tau_1]}}{\Gamma, f :: [\tau_1] \to \tau_2, y :: \tau_1 \vdash (f\ (y : [\,]_{\tau_1})) :: \tau_2}}{\Gamma, f :: [\tau_1] \to \tau_2 \vdash (\lambda y :: \tau_1 . f\ (y : [\,]_{\tau_1})) :: \tau_1 \to \tau_2}$$

(HEAD)

We use the rule (LCASE) with $\Gamma, l :: [\tau_1] \vdash l :: [\tau_1]$, $\Gamma \vdash \bot_{\tau_1} :: \tau_1$ and $\Gamma, x_1 :: \tau_1 \vdash x_1 :: \tau_1$ as premises. They are all three immediate.

[1]During the proof we omit uninteresting typing context entries in type derivations to save space.

(PAIR→)

$$\dfrac{\dfrac{\dfrac{\Gamma, f :: (\tau_1, \tau_2) \to \tau_3 \vdash f :: (\tau_1, \tau_2) \to \tau_3 \qquad \dfrac{\Gamma, x :: \tau_1 \vdash x :: \tau_1 \qquad \Gamma, y :: \tau_2 \vdash y :: \tau_2}{\Gamma, x :: \tau_1, y :: \tau_2 \vdash (x, y) :: (\tau_1, \tau_2)}}{\Gamma, f :: (\tau_1, \tau_2) \to \tau_3, x :: \tau_1, y :: \tau_2 \vdash (f\ (x, y)) :: \tau_3}}{\Gamma, f :: (\tau_1, \tau_2) \to \tau_3, x :: \tau_1 \vdash (\lambda y :: \tau_2 . f\ (x, y)) :: \tau_2 \to \tau_3}}{\Gamma, f :: (\tau_1, \tau_2) \to \tau_3 \vdash (\lambda x :: \tau_1 . \lambda y :: \tau_2 . f\ (x, y)) :: \tau_1 \to \tau_2 \to \tau_3}$$

(PROJ)

We have to consider two substitutions. The term derivation is both times the same style. We use (PCASE′) with $\Gamma, p :: (\tau_1, \tau_2) \vdash p :: (\tau_1, \tau_2)$ and either $\Gamma, x :: \tau_1 \vdash x :: \tau_1$ or $\Gamma, y :: \tau_2 \vdash y :: \tau_2$ as premises.

(EITHER→)

We also consider two substitutions. The term constructions are again of similar style. The first one is constructed as follows:

$$\dfrac{\dfrac{\Gamma, f :: \mathit{Either}\ \tau_1\ \tau_2 \to \tau_3 \vdash f :: \mathit{Either}\ \tau_1\ \tau_2 \to \tau_3 \qquad \dfrac{\Gamma, x :: \tau_1 \vdash x :: \tau_1}{\Gamma, x :: \tau_1 \vdash \mathbf{Left}_{\tau_2}\ x :: \mathit{Either}\ \tau_1\ \tau_2}}{\Gamma, f :: \mathit{Either}\ \tau_1\ \tau_2 \to \tau_3, x :: \tau_1 \vdash (f\ (\mathbf{Left}_{\tau_2}\ x)) :: \tau_3}}{\Gamma, f :: \mathit{Either}\ \tau_1\ \tau_2 \to \tau_3 \vdash (\lambda x :: \tau_1 . f\ (\mathbf{Left}_{\tau_2}\ x)) :: \tau_1 \to \tau_3}$$

The second derivation is similar, but with $y :: \tau_2$ and $\mathbf{Right}_{\tau_1}\ y$ instead of $x :: \tau_1$ and $\mathbf{Left}_{\tau_2}\ x$, respectively.

(DIST$_1$)

We use the rule (ECASE′) with the premises $\Gamma, e :: \mathit{Either}\ \tau_1\ \tau_2 \vdash e :: \mathit{Either}\ \tau_1\ \tau_2$, $\Gamma, x :: \tau_1 \vdash x :: \tau_1$ and $\Gamma \vdash \bot_{\tau_2} :: \tau_2$.

(DIST$_2$)

Change the last two premises from case (DIST$_1$) to $\Gamma \vdash \bot_{\tau_1} :: \tau_1$ and $\Gamma, x :: \tau_2 \vdash x :: \tau_2$.

(BOTTOM→′)

$$\dfrac{\Gamma, f :: \tau_1 \to \tau_2 \vdash f :: \tau_1 \to \tau_2 \qquad \Gamma \vdash \bot_{\tau_1} :: \tau_1}{\Gamma, f :: \tau_1 \to \tau_2 \vdash (f\ \bot_{\tau_1}) :: \tau_2}$$

(ARROW→′)

$$\dfrac{\dfrac{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \vdash f :: (\tau_1 \to \tau_2) \to \tau_3 \qquad \dfrac{\Gamma, u :: \tau_1, z :: \tau_2 \vdash z :: \tau_2}{\Gamma, z :: \tau_2 \vdash (\lambda u :: \tau_1 . z) :: \tau_1 \to \tau_2}}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3, z :: \tau_2 \vdash (f\ (\lambda u :: \tau_1 . z)) :: \tau_3}}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \vdash (\lambda z :: \tau_2 . f\ (\lambda u :: \tau_1 . z)) :: \tau_2 \to \tau_3}$$

to gain a substitute for g. The substitution of w by x is just a renaming of a bound variable occurrence and as substitute for y we have

$$\dfrac{\dfrac{\Gamma, g :: \ldots, f :: \ldots \vdash f :: (\tau_1 \to \tau_2) \to \tau_3 \qquad \dfrac{\dfrac{\Gamma, g :: \tau_2 \to \tau_3, w :: \tau_1 \vdash t_1 :: \tau_2}{\Gamma, g :: \tau_2 \to \tau_3, x :: \tau_1 \vdash t_1[x/w] :: \tau_2}}{\Gamma, g :: \ldots \vdash (\lambda x :: \tau_1 . t_1[x/w]) :: \tau_1 \to \tau_2}}{\Gamma, g :: \tau_2 \to \tau_3, f :: (\tau_1 \to \tau_2) \to \tau_3 \vdash (f\ (\lambda x :: \tau_1 . t_1[x/w])) :: \tau_3}}{\Gamma, f :: (\tau_1 \to \tau_2) \to \tau_3 \vdash (f\ (\lambda x :: \tau_1 . t_1[\lambda z :: \tau_2 . f\ (\lambda u :: \tau_1 . z)/g, x/w]) :: \tau_3} \ (*)$$

where (*), i.e., removing the substitution of g, is by the first derivation tree.

(BOTTOM→)

Similar to (BOTTOM→′)

(BOTTOM→°)

Similar to (BOTTOM→′)

(APP′°)

We use the rule (APP) with the premises $\Gamma, x :: \tau_1 \vdash x :: \tau_1$ and $\Gamma, f :: \tau_1 \to \tau_2 \vdash f :: \tau_1 \to \tau_2$, and have $\Gamma, x :: \tau_1, f :: \tau_1 \to \tau_2 \vdash (f\ x) :: \tau_2$.

All remaining rules are $\circ$-annotated and similar to the unannotated cases. □

Proof of Theorem 7 on page 83 (termination of TermFind)

Proof. To state the termination of TermFind we introduce a termination order that decreases with every (backward) rule application during the construction of the derivation tree and thus reaches its least element after finitely many rule applications.

Consider, we have an arbitrary external input $(\Gamma; \tau)$. We define the termination order on a summed up complexity measure over τ and over all types that are assigned to term variables in Γ, i.e., all types in Γ_V. We regard tuples $(\mathit{Either}_n, \ldots, \mathit{Either}_1, (\cdot,\cdot), [\cdot], \to, \#\text{var})$ where n is the maximal nesting level of a structural[2] subtype $\mathit{Either}\ \tau_1\ \tau_2$ in τ and in the types in Γ_V together. Nesting level is made precise below. The entries in the tuple are the number of

[2]By *structural* we want to emphasize that we distinguish two subtypes by their position in a type, even if they are syntactically equal. This is not very explicit in the proof, but could easily

occurrences of $Either$ in different nesting levels, the number of occurrences of pairs, of lists and of arrows in the regarded types; and finally the number of term variables in Γ_v. As order on those tuples we first compare the maximal nesting level of $Either$ and if the tuples are of the same size we take the lexicographical order on them, comparing each component separately, starting with the left-most. The nesting level of a structural subtype τ'' of a type τ' in τ is defined recursively by the function $nl\ (\tau'', \tau', r)$, where τ'' is the structural subtype we search the nesting level for, τ' the currently examined structural subtype of τ and r the nesting level of τ' in τ. We call $nl\ (\tau'', \tau, 0)$ to get the nesting level of τ'' in τ. The function nl is defined as follows[3]:

```
$nl\ (\tau'', \tau', r) =$
  **if** $\tau' \equiv \tau''$ **then** $r$
                  **else let** $rec\ \tau_1\ \tau_2 =$ **if**   $nl\ (\tau'', \tau_1, r+1) \not\equiv None$
                                  **then** $nl\ (\tau'', \tau_1, r+1)$
                                  **else** $nl\ (\tau'', \tau_2, r + nlmax\ \tau_1 + 1)$
                    **in case** $\tau'$ **of**
                         $[\tau_1]$              $\rightarrow nl\ (\tau'', \tau_1, r+1)$
                         $(\tau_1, \tau_2)$        $\rightarrow rec\ \tau_1\ \tau_2$
                         $Either\ \tau_1\ \tau_2$ $\rightarrow rec\ \tau_1\ \tau_2$
                         $\tau_1 \rightarrow \tau_2$        $\rightarrow rec\ \tau_1\ \tau_2$
                         _                $\rightarrow None$
```

where $nlmax\ \tau$ returns the maximal nesting level of τ, i.e., the maximal result of $nl\ (\tau', \tau, 0)$ where τ' ranges over all structural subtypes of τ. For every call $nl\ (\tau'', \tau, 0)$ with τ'' a structural subtype of τ the function nl returns the nesting level of τ'' in τ (i.e., not $None$) and hence is well-defined.

Having the definition of nesting level we examine whether the measure really decreases with each rule's application. Rules without premise can be disregarded, because the algorithm terminates immediately whenever they are applied. Also rules with a pointedness check as only premise can be disregarded, because pointedness checks are also obviously terminating if we consider the rules in Figure 4.5. Regarding (UPDROP), (NDROP), (UDROP), (NAT°), (UNIT°), (LIST°), (PAIR°) and (EITHER°) there is only a term variable removed from the term context, thus none of the values in our measure increases, but at least the number of variables in the term context decreases.

For (ABS) an arrow is eliminated, nothing new is introduced and there is no chance to deepen the nesting level of any $Either$. Hence the measure decreases.

Regarding (WRAP→') a list is eliminated and the nesting levels of τ_1 and τ_2 are reduced.

be made explicit by annotating (sub)types with unique identifiers. We refrain from annotations, because they blur the presentation further without adding significant details, not captured in this footnote.

[3]The definition is in Haskell-like syntax. $None$ is a special integer value. Correct Haskell code could use $Maybe$ with $Nothing$ as $None$, but consequently must pack/unpack r in $Just$.

For (PAIR$\rightarrow$) the nesting level of τ_1 decreases by one and the levels of τ_2 and τ_3 remain. Because the elimination of the pair is weighted higher than the introduction of an arrow, the termination order decreases.

The other rules behave harmless in similar ways. We only need to check that no *Either* is nested deeper than it was in the conclusion and that some structural element is removed. Regarding (ARROW$\rightarrow'$) we have to consider each premise separately.

The only exception in just removing something is (EITHER$\rightarrow$). The premise of (EITHER$\rightarrow$) duplicates τ_3. Hence, we can have (limited) growth in the number of all structural elements. What we know is that the nesting level of τ_1, τ_2 and τ_3 and consequently the nesting level of all their subtypes decreases. Here, the distinction between differently leveled *Either* in the measure comes in. Assume we have a second *Either* besides the eliminated one, then it is moved to a lower level and the termination order decreases. If we have no other *Either*, we have decreased the number of *Either*s to zero and hence decreased the termination measure, too. □

Proof of Lemma 15 on page 98

Proof. We refer to the notation in Definition 26 and prove Lemma 15 over the different cases in Definition 26 and by induction on the length of l. Keep in mind that functions are related iff their results for related arguments are, and note that the first element of l if written as $(\mathbf{v}, \mathbf{v}')$ in Definition 26 is already determined by the type τ as we see from the definition of a disrelator. Furthermore, remember that τ' is pointed under Γ_τ and thus $(\bot, \bot) \in \Delta^{\mathsf{fix}}_{\tau'}$.

First, we prove $(\mathbf{g_1}, \mathbf{g_2}) \in \Delta^{\mathsf{fix}}_{\tau\rightarrow\tau'}$. If τ is an unpointed type variable α, we check the required properties for all argument pairs in $\Delta^{\mathsf{fix}}_{\alpha} = \{(\bot, ()), ((), ())\}$ and see that the values of $\mathbf{g_1}$ and $\mathbf{g_2}$ applied to the components of these pairs, respectively, are related. For all other cases, except $\tau = \tau_1 \rightarrow \tau_2$, $\bot$ is only related to itself by the definition of the logical relation (cf. Figures 4.4 and 2.10). Thus $(\mathbf{g_1}, \mathbf{g_2}) \in \Delta^{\mathsf{fix}}_{\tau\rightarrow\tau'}$ is either immediate, or by the induction hypothesis that it holds for every (structural) subtype of τ. In the cases with $\tau = \tau_1 \rightarrow \tau_2$ the induction hypothesis suffices, too. This is because $(\mathbf{p}^+_{\tau_1}, \mathbf{p}^+_{\tau_1})$ and $(\mathbf{v}, \mathbf{v}')$ are both related, respectively.

Second, we regard $(\mathbf{g_1}\ \mathbf{t_1}, \mathbf{g_2}\ \mathbf{t_2}) = (\mathbf{t}'_1, \bot)$. For the case $\tau = \alpha$ and unpointed, we have $(\mathbf{t_1}, \mathbf{t_2}) = (\bot, \bot)$ for it is the only unrelated pair for type α, and we are done. For all other cases where $l = []$, except for $\tau = \tau_1 \rightarrow \tau_2$, we know from Definition 25 that $(\mathbf{t_1}, \mathbf{t_2}) = (\mathbf{t_1}, \bot)$ and $\mathbf{t_1} \neq \bot$ because $\Gamma \vdash \tau \in \mathsf{Pointed}$. Taking the components of this pair as arguments for the respective $\mathbf{g_1}$ and $\mathbf{g_2}$, the proof is immediate. For $\tau = \tau_1 \rightarrow \tau_2$ and $l = []$ the proof relies on $\mathbf{t_1}$ a constant function (cf. Definition 25) and the induction hypothesis. For the cases with $l \neq []$ the induction hypotheses suffice again. Regarding $(\mathbf{g_1}\ \mathbf{t_1}, \mathbf{g_2}\ \mathbf{t_2})$

we apply the head of l to $(\mathbf{t_1}, \mathbf{t_2})$. Hence, for the result the former disrelator with l replaced by its tail is a disrelator, and therefore the recursive call of the construction is well defined. □

Proof of Lemma 16 on page 106

Proof. The proof is by induction on the length of the derivation tree generated by the second and third phase of ExFind. Hence, we consider separately each rule of phase II and III as root rule of the derivation tree, yielding the final environment. We assume that for the premises we can already construct the respective result environments. The environment definitions in Definition 30 ensure directly the domain restrictions for a result environment. Hence, it remains to check that $(\sigma_1(x), \bot) \notin \Delta^{\mathsf{fix}}_\tau$ and $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathsf{fix}}_\tau$ for the appropriate type. By Lemma 14 (2) we get $(\sigma_1(x), \bot) \notin \Delta^{\mathsf{fix}}_\tau$ and by Lemma 14 (1) we get $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathsf{fix}}_\tau$ for all rules of the third phase, all rules whose premises are calls to the third phase, and for ($\textsc{Var}^\circ$). For the rule ($\textsc{App}'^\circ$) Lemma 15 and the induction hypothesis are sufficient. For all other rules of the second phase $(\sigma_1(x), \bot) \notin \Delta^{\mathsf{fix}}_\tau$ as well as $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathsf{fix}}_\tau$ is directly by the definition of the logical relation and the induction hypothesis for the substituted variable. □

Proof of Lemma 17 on page 106

Proof. As the proof of Lemma 16, this proof is by induction on the length of the derivation tree generated by the second and third phase of ExFind. In the remainder of the proof, we denote common parts of the term environments of premise and conclusion of a rule by σ_1 and σ_2 respectively. We give altered entries explicitly, where the value a variable is mapped to is symbolized by the variable's name and the respective environment index. By this notation, we refer to the value generated for the respective environment entry by Definition 30. Furthermore, we denote the term in the premise of a rule by t^p and the term in the conclusion by t^c. For axioms the term is also denoted by t^c.

For the rules of the third phase it suffices to show $(\llbracket t^c \rrbracket^{\mathsf{fix}}_{\sigma_1}, \bot) \notin \Delta^{\mathsf{fix}}_\tau$. For ($\textsc{Var}'$) the term is not related to $\bot$ as a consequence of Lemma 16. For all other rules of the third phase, $(\llbracket t^c \rrbracket^{\mathsf{fix}}_{\sigma_1}, \bot) \notin \Delta^{\mathsf{fix}}_\tau$ follows directly from the definition of the logical relation.

For the rules of the second phase, we show that term t in the conclusion is a strong result term. For ($\textsc{Var}^\circ$) Lemma 16 is sufficient to guarantee x a strong result term.

For rule ($\textsc{Nat}^\circ$), we show that $\llbracket t^c \rrbracket^{\mathsf{fix}}_{\sigma_2^\bot}[x \mapsto \bot] = \bot$. Since t^c is a case expression with x as scrutinee, the scrutinee and hence the whole case expression evaluates to $\bot$. Furthermore, we have to show that $(\llbracket t^c \rrbracket^{\mathsf{fix}}_{\sigma_1[x \mapsto \mathbf{p}^+_{Nat}]}, \bot) \notin \Delta^{\mathsf{fix}}_\tau$. Since

$\mathbf{p}_{Nat}^{+} = \mathbf{0}$ the semantics of t^c is equal to the semantics of t^p under $\sigma_1[x \mapsto \mathbf{p}_{Nat}^{+}]$. And hence, the induction hypothesis suffices for the proof. All other rules that yield case expressions as terms are proved similarly.

For the remaining rules, we prove that the semantic interpretations of the result term remain unchanged. Because for all rules the proofs are similar, we give the details only for (BOTTOM$\rightarrow^\circ$).

For the rule (BOTTOM$\rightarrow^\circ$) we have

$$[\![t]\!]^{\mathsf{fix}}_{\sigma_1[y \mapsto \mathbf{y_1}]} = [\![t]\!]^{\mathsf{fix}}_{\sigma_1[y \mapsto [\![f\ \bot_{\tau_1}]\!]^{\mathsf{fix}}_{\sigma_1[f \mapsto \mathbf{f_1}]}]} = [\![t[f\ \bot_{\tau_1}/y]]\!]^{\mathsf{fix}}_{\sigma_1[f \mapsto \mathbf{f_1}]}$$

and we reason similarly for $\sigma_2^{\bot}$, where always the first equality is by the definition of the environment entries for f in Definition 30 and the second equality by (the appropriate adaptation of) Lemma 4. □

Proof of Lemma 18 on page 106

Proof. The proof proceeds by induction on the length of the derivation tree. To prove (σ_1, σ_2) a result environment, we show only $(\sigma_1(x), \sigma_2(x)) \in \Delta^{\mathsf{fix}}_{\tau}$ for each variable x of type τ introduced to the typing context during a rule application. All other properties required by the definition of result environment (Definition 21) are easy to check looking at Definition 30. To prove the constructed term t a result term and the constructed ϖ a disrelator to it, we state the interesting cases only.

During the proof we denote the term in the premise of the currently considered rule by t^p and the term in the conclusion by t^c. Similarly all environments and also the disrelators are indexed by p and c for premise and conclusion, respectively. Concerning rule (ARROW$\rightarrow^*$), which has two premises (that are no pointedness checks), the indices for the premises are additionally enriched by a number, i.e., we write p_1 and p_2. Note that an axiom is a rule without a premise, i.e., we regard it as a conclusion.

For (BOTTOM) we have (σ_1^c, σ_2^c) a result environment by Lemma 14 (1). By Lemma 13 (2) t^c is a result term. The constructed $\varpi^c = (\tau, [\,])$ also meets Definition 25 as a disrelator for $([\![t]\!]^{\mathsf{fix}}_{\sigma_1^c}, [\![t]\!]^{\mathsf{fix}}_{\sigma_2^c})$.

For the rules (UPDROP), (NDROP) and (UDROP) by Lemma 14 (1) the term environments form a result environment. Since t and ϖ remain unchanged, there is nothing else to prove.

For the rule (ABS) the term environments remain unchanged and since we have $([\![t^c]\!]^{\mathsf{fix}}_{\sigma_1^c}\ \sigma_1^p(x), [\![t^c]\!]^{\mathsf{fix}}_{(\sigma_2^c)^{\bot}}\ (\sigma_2^p)^{\bot}(x)) = ([\![t^p]\!]^{\mathsf{fix}}_{\sigma_1^p}, [\![t^p]\!]^{\mathsf{fix}}_{(\sigma_2^p)^{\bot}}) \notin \Delta^{\mathsf{fix}}_{\tau_2}$, the term t^c is a result term: By the definition of the logical relation for function types we have $([\![t^c]\!]^{\mathsf{fix}}_{\sigma_1^c}, [\![t^c]\!]^{\mathsf{fix}}_{(\sigma_2^c)^{\bot}}) \notin \Delta^{\mathsf{fix}}_{\tau_1 \rightarrow \tau_2}$. Applying the first element of the list in ϖ^c to

$([\![t^c]\!]^{\mathsf{fix}}_{\sigma_1^c}, [\![t^c]\!]^{\mathsf{fix}}_{(\sigma_2^c)^\perp})$ we get $([\![t^p]\!]^{\mathsf{fix}}_{\sigma_1^p}, [\![t^p]\!]^{\mathsf{fix}}_{(\sigma_2^p)^\perp})$ to which the tail of the list combined with the type in ϖ^c is a disrelator by the induction hypothesis. Consequently, ϖ^c is a disrelator for $([\![t^c]\!]^{\mathsf{fix}}_{\sigma_1^c}, [\![t^c]\!]^{\mathsf{fix}}_{(\sigma_2^c}{}^{)\perp})$. Similar arguments work for (WRAP), (PAIR$_1$), (PAIR$_2$), (LEFT) and (RIGHT).

In all remaining rules ϖ is not changed. Hence, if we show that the term environments in the conclusion form a result environment and the values of t^c in the different term environments of the conclusion are equal to the values of t^p in the respective term environments of the premise, we are done.

In the case (WRAP→'), regarding the induction hypothesis $(\sigma_1^p(g), \sigma_2^p(g)) \in \Delta^{\mathsf{fix}}_{\tau_1 \to \tau_2}$, the definition of the logical relation for list types and $(\perp, \perp) \in \Delta^{\mathsf{fix}}_{\tau_2}$ (because $\Gamma \vdash \tau_2 \in \mathsf{Pointed}$ is required as premise), we have $(\sigma_1^c(f), \sigma_2^c(f)) \in \Delta^{\mathsf{fix}}_{[\tau_1] \to \tau_2}$ and hence (σ_1^c, σ_2^c) a result environment. Furthermore,

$$[\![t]\!]^{\mathsf{fix}}_{\sigma_1[g \mapsto \mathbf{g_1}]} = [\![t]\!]^{\mathsf{fix}}_{\sigma_1[g \mapsto [\![\lambda y :: \tau_1 . f\ (y : [\,])]\!]^{\mathsf{fix}}_{\sigma_1[f \mapsto \sigma_1^c(f)]}]} = [\![t[\lambda y :: \tau_1 . f\ (y : [\,]) / g]]\!]^{\mathsf{fix}}_{\sigma_1[f \mapsto \sigma_1^c(f)]}$$

and we reason similarly for the term environment σ_2^c. Both times, the first equivalence is by the choice for $\sigma_1^c(f)$ and $\sigma_2^c(f)$ in Definition 30, and the second equivalence by (the appropriate version of) Lemma 4.

For all remaining rules the reasoning is similar and we point only to additional difficulties. Regarding (BOTTOM→') and (ARROW→*) the fact that $(\sigma_1^c(f), \sigma_2^c(f)) \in \Delta^{\mathsf{fix}}_\tau$ for appropriate τ is by Lemma 15. For (BOTTOM→') Lemma 16 together with $(\tau, [\,])$ a disrelator for every strong result term of type τ suffices to verify that we meet the conditions of Lemma 15. For (ARROW→*) the induction hypothesis for t^{p_1} and ϖ^{p_1}, and the result from Lemma 16, as well as the induction hypothesis for the environments of the first premise are sufficient to verify that we meet the conditions of Lemma 15.

For (ARROW→*) the proof is still incomplete after reasoning as above: The tuple of term environments (σ_1^p, σ_2^p) found by mergeEnv (cut to the entries in the respective typing context in the premises) has to be a result environment, such that t^{p_1} is a result term and t^{p_2} a strong result term w.r.t. (σ_1^p, σ_2^p). We know that the result environments $(\sigma_1^{p_1}, \sigma_2^{p_1})$ and $(\sigma_1^{p_2}, \sigma_2^{p_2})$, belonging to the two premises respectively, fit one to t^{p_1} and the other to t^{p_2}. The proof that (σ_1^p, σ_2^p) with the appropriate domain restrictions forms respective result environments is accomplished by induction over the different histories of variables. This is the proof idea: Since the same history tag corresponds to equal term environment manipulations, variables with the same history have the same value and thus can be taken from either environment. If some history ends by Leaf it means that the choice of the values was close to arbitrary and has just to be non-$\perp$ (and in the respective domain). Therefore, such an entry can always be substituted by an entry for a term variable with the same type, but a different history. This is not possible the other way round. The algorithm mergeEnv performs exactly the admissible substitutions when comparing the histories belonging to the two result environments given as input. □

A.2 Proofs from Chapter 5

Proof of Lemma 23 on page 127

Proof. Induction on the structure of the subtype derivation. Hence, we regard each of the rules in Figure 5.6 separately.

For (S-VAR) the assertion of Lemma 23 is immediate.

For $(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu'\in\{\circ,\epsilon\},\nu'\leqslant\nu}$ we regard all rules of the family separately. For the rule with $\nu = \nu' = \circ$ we reason:

$$\begin{array}{l}
\Delta^{\mathbf{seq}*}_{\tau_1 \to^\circ \tau_2,\rho} \subseteq \Delta^{\mathbf{seq}*}_{\tau_1' \to^\circ \tau_2',\rho} \\
\Leftrightarrow \quad \{ \text{ logical relation } \} \\
\{(\mathbf{f},\mathbf{g} \mid \forall (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_1,\rho}.\ (\mathbf{f}\ \$\ \mathbf{a}, \mathbf{g}\ \$\ \mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_2,\rho})\} \\
\qquad \subseteq \{(\mathbf{f},\mathbf{g} \mid \forall (\mathbf{a},\mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_1',\rho}.\ (\mathbf{f}\ \$\ \mathbf{a}, \mathbf{g}\ \$\ \mathbf{b}) \in \Delta^{\mathbf{seq}*}_{\tau_2',\rho})\} \\
\Leftarrow \quad \{ \text{ property of functions } \} \\
\Delta^{\mathbf{seq}*}_{\tau_1',\rho} \subseteq \Delta^{\mathbf{seq}*}_{\tau_1,\rho} \wedge \Delta^{\mathbf{seq}*}_{\tau_2,\rho} \subseteq \Delta^{\mathbf{seq}*}_{\tau_2',\rho} \\
\Leftarrow \quad \{ \text{ induction hypotheses } \} \\
\tau_1' \preceq \tau_1 \wedge \tau_2 \preceq \tau_2'
\end{array}$$

For $\nu = \nu' = \epsilon$ we reason the same way, but exchange the definition of the logical relation. For $\nu = \epsilon$ and $\nu' = \circ$ reasoning is also similar, again with adjusted definitions of the logical relation, due to the different function types.

Finally, for (S-LIST) the definition of the logical relation for list types and the induction hypothesis suffice for the proof. □

Proof of Lemma 24 on page 130

Proof. Reflexivity is proved inductively over the type structure. The different cases are obvious by the rules in Figure 5.6.

Because subtyping is shape conformant (cf. Lemma 21), we can prove transitivity also on the structure of the type. Consider the three types τ, τ' and τ'' with $\tau \preceq \tau'$ and $\tau' \preceq \tau''$. We prove $\tau \preceq \tau''$ inductively over the structure of τ.

$\tau = \alpha$

$\tau = \alpha$
$\Rightarrow \quad \{ \tau \preceq \tau' \text{ only by rule (S-VAR)} \}$
$\tau' = \alpha$
$\Rightarrow \quad \{ \tau' \preceq \tau'' \text{ only by rule (S-VAR)} \}$
$\tau'' = \alpha$
$\Rightarrow \quad \{ \text{(S-VAR)}, \tau = \alpha \text{ and } \tau'' = \alpha \}$
$\tau \preceq \tau''$

$\tau = \tau_1 \rightarrow^{\nu} \tau_2$

$\tau = \tau_1 \rightarrow^{\nu} \tau_2$
$\Rightarrow \quad \{ \tau \preceq \tau' \text{ only by rule (S-ARROW}_{\nu,\nu'}) \}$
$\tau' = \tau_1' \rightarrow^{\nu'} \tau_2'$ with $\nu' \leqslant \nu, \tau_1' \preceq \tau_1$ and $\tau_2 \preceq \tau_2'$
$\Rightarrow \quad \{ \tau' \preceq \tau'' \text{ only by rule (S-ARROW}_{\nu',\nu''}) \}$
$\tau'' = \tau_1'' \rightarrow^{\nu''} \tau_2''$ with $\nu'' \leqslant \nu', \tau_1'' \preceq \tau_1'$ and $\tau_2' \preceq \tau_2''$
$\Rightarrow \quad \{ \text{induction hypothesis and transitivity of } \leqslant \}$
$\nu'' \leqslant \nu \wedge \tau_1'' \preceq \tau_1 \wedge \tau_2 \preceq \tau_2''$
$\Rightarrow \quad \{ \text{(S-ARROW}_{\nu,\nu''}) \}$
$\tau \preceq \tau''$

$\tau = [\tau_1]$

Similar to $\tau = \tau_1 \rightarrow^{\nu} \tau_2$.

□

Proof of Lemma 25 on page 130

Proof. We use induction on the depth of the derivation tree. Consider $\Gamma \vdash t :: \tau$ derivable in $\lambda^{\alpha}_{\mathbf{seq}*}$. By Lemma 24 we assume that the root of the derivation tree is (SUB) followed by another rule from Figure 5.4. Hence, it suffices to replace every combination (SUB) plus another typing rule of $\lambda^{\alpha}_{\mathbf{seq}*}$ by a rule (sequence) from $\lambda^{\alpha}_{\mathbf{seq}+}$, potentially with calls to (SUB) at the leaves of the derivation fragment. We regard each rule (family) different from (SUB) separately.

(VAR)

We replace the sequence

$$\frac{\Gamma, x :: \tau \vdash x :: \tau \ \text{(VAR)} \qquad \tau \preceq \tau'}{\Gamma, x :: \tau \vdash x :: \tau'} \ \text{(SUB)}$$

by

$$\frac{\tau \preceq \tau'}{\Gamma, x :: \tau \vdash x :: \tau'} \ \text{(VAR}^+)$$

(NIL)

Similar to case (VAR), we replace (NIL) by (NIL$^+$).

(CONS)

We replace

$$\dfrac{\dfrac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]}\,(\textsc{Cons}) \qquad [\tau] \preceq \tau''}{\Gamma \vdash (t_1 : t_2) :: \tau''}\,(\textsc{Sub})$$

by

$$\dfrac{\dfrac{\Gamma \vdash t_1 :: \tau \qquad \tau \preceq \tau'}{\Gamma \vdash t_1 :: \tau'}\,(\textsc{Sub}) \qquad \dfrac{\Gamma \vdash t_2 :: [\tau] \qquad [\tau] \preceq [\tau']}{\Gamma \vdash t_2 :: [\tau']}\,(\textsc{Sub})}{\Gamma \vdash (t_1 : t_2) :: [\tau']}\,(\textsc{Cons})$$

Forcing $\tau'' = [\tau']$ is not a limitation, since subtyping is shape conformant.

(LCASE)

We replace

$$\dfrac{\dfrac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau}{\Gamma \vdash (\textbf{case } t \textbf{ of } \{[\,] \to t_1; x_1 : x_2 \to t_2\}) :: \tau}\,(\text{LC}\textsc{ase}) \qquad \tau \preceq \tau'}{\Gamma \vdash (\textbf{case } t \textbf{ of } \{[\,] \to t_1; x_1 : x_2 \to t_2\}) :: \tau'}\,(\textsc{Sub})$$

by

$$\dfrac{\Gamma \vdash t :: [\tau_1] \qquad \dfrac{\Gamma \vdash t_1 :: \tau \qquad \tau \preceq \tau'}{\Gamma \vdash t_1 :: \tau'}\,(\textsc{Sub}) \qquad \dfrac{\Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau \qquad \tau \preceq \tau'}{\Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau'}\,(\textsc{Sub})}{\Gamma \vdash (\textbf{case } t \textbf{ of } \{[\,] \to t_1; x_1 : x_2 \to t_2\}) :: \tau'}\,(\text{LC}\textsc{ase})$$

(ABS$_\nu$)$_{\nu \in \{\circ, \epsilon\}}$

We replace

$$\dfrac{\dfrac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1 \to^{\nu} \tau_2}\,(\textsc{Abs}_\nu) \qquad \tau_1 \to^{\nu} \tau_2 \preceq \tau_1' \to^{\nu'} \tau_2'}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1' \to^{\nu'} \tau_2'}\,(\textsc{Sub})$$

by

$$\dfrac{\dfrac{\Gamma, x :: \tau_1 \vdash t :: \tau_2 \qquad \tau_2 \preceq \tau_2'}{\Gamma, x :: \tau_1 \vdash t :: \tau_2'}\,(\textsc{Sub}) \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1' \to^{\nu'} \tau_2'}\,(\textsc{Abs}^+_{\nu'})$$

$(\textsc{App}_\nu)_{\nu\in\{\circ,\epsilon\}}$

We replace

$$\dfrac{\dfrac{\Gamma\vdash t_1 :: \tau_1 \to^\nu \tau_2 \qquad \Gamma\vdash t_2 :: \tau_1}{\Gamma\vdash (t_1\ t_2) :: \tau_2}(\textsc{App}_\nu) \qquad \tau_2 \preceq \tau_2'}{\Gamma\vdash (t_1\ t_2) :: \tau_2'}(\textsc{Sub})$$

by

$$\dfrac{\dfrac{\Gamma\vdash t_1 :: \tau_1 \to^\nu \tau_2 \qquad \tau_1 \to^\nu \tau_2 \preceq \tau_1 \to^\nu \tau_2'}{\Gamma\vdash t_1 :: \tau_1 \to^\nu \tau_2'}(\textsc{Sub}) \qquad \Gamma\vdash t_2 :: \tau_1}{\Gamma\vdash (t_1\ t_2) :: \tau_2'}(\textsc{App}_\nu)$$

$(\textsc{Fix}_\nu)_{\nu\in\{\circ,\epsilon\}}$

We replace

$$\dfrac{\dfrac{\Gamma\vdash t :: \tau \to^\nu \tau}{\Gamma\vdash (\mathbf{fix}\ t) :: \tau}(\textsc{Fix}_\nu) \qquad \tau \preceq \tau'}{\Gamma\vdash (\mathbf{fix}\ t) :: \tau'}(\textsc{Sub})$$

by

$$\dfrac{\Gamma\vdash t :: \tau \to^\nu \tau \qquad \tau \preceq \tau'}{\Gamma\vdash (\mathbf{fix}\ t) :: \tau'}(\textsc{Fix}^+_\nu)$$

$(\textsc{SLet}')$

We replace

$$\dfrac{\dfrac{\Gamma\vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma\vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma\vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}(\textsc{SLet}') \qquad \tau_2 \preceq \tau_2'}{\Gamma\vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2'}(\textsc{Sub})$$

by

$$\dfrac{\Gamma\vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma\vdash t_1 :: \tau_1 \qquad \dfrac{\Gamma, x :: \tau_1 \vdash t_2 :: \tau_2 \qquad \tau_2 \preceq \tau_2'}{\Gamma, x :: \tau_1 \vdash t_2 :: \tau_2'}(\textsc{Sub})}{\Gamma\vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2'}(\textsc{SLet}')$$

To translate a derivation tree in $\lambda^\alpha_{\mathbf{seq}+}$ to one in $\lambda^\alpha_{\mathbf{seq}*}$ that yields the same typing judgment, we transform each typing rule in $\lambda^\alpha_{\mathbf{seq}+}$ into a (sequence of) typing rules in $\lambda^\alpha_{\mathbf{seq}*}$. Of course, the typing rules that are present in both calculi can be taken over unchanged. For the other rules (or rule families) the concrete transformations are as follows.

(VAR$^+$)

We replace

$$\frac{\tau \preceq \tau'}{\Gamma, x :: \tau \vdash x :: \tau'}\ (\textsc{Var}^+)$$

by

$$\frac{\Gamma, x :: \tau \vdash x :: \tau\ (\textsc{Var}) \qquad \tau \preceq \tau'}{\Gamma, x :: \tau \vdash x :: \tau'}\ (\textsc{Sub})$$

(NIL$^+$)

Similar to case (VAR$^+$).

(ABS$^+_\nu$)$_{\nu \in \{\circ, \epsilon\}}$

We replace

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2 \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1' \to^\nu \tau_2}\ (\textsc{Abs}^+_\nu)$$

by

$$\frac{\dfrac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1 \to^\nu \tau_2}\ (\textsc{Abs}_\nu) \qquad \tau_1 \to^\nu \tau_2 \preceq \tau_1' \to^\nu \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1' \to^\nu \tau_2}\ (\textsc{Sub})$$

(FIX$^+_\nu$)$_{\nu \in \{\circ, \epsilon\}}$

We replace

$$\frac{\Gamma \vdash t :: \tau \to^\nu \tau \qquad \tau \preceq \tau'}{\Gamma \vdash (\mathbf{fix}\ t) :: \tau'}\ (\textsc{Fix}^+_\nu)$$

by

$$\frac{\dfrac{\Gamma \vdash t :: \tau \to^\nu \tau}{\Gamma \vdash (\mathbf{fix}\ t) :: \tau}\ (\textsc{Fix}_\nu) \qquad \tau \preceq \tau'}{\Gamma \vdash (\mathbf{fix}\ t) :: \tau'}\ (\textsc{Sub})$$

□

Proof of Theorem 10 on page 135

The proof of Theorem 10 proceeds basically as follows: We construct for every concrete valid typing judgment in $\lambda^\alpha_{\mathsf{seq}^+}$ a conditional one that can be instantiated as shown in Definition 41 and thus shows that the original type statement is also valid in $\lambda^\alpha_{\mathsf{seq}^C}$. The components of the conditional type statement are called parametrizations of its concrete counterparts:

DEFINITION 51 (parametrization)

Let κ a concrete term, type or typing context. We call $\dot\kappa$ a *parametrization* of κ if there exists an annotation substitution ϱ, such that $\dot\kappa\varrho = \kappa$.

Conversely, we show that every instance of a conditional typing judgment in $\lambda^{\alpha}_{\mathbf{seq}^C}$ that satisfies the typing constraint, is also a valid typing judgment in $\lambda^{\alpha}_{\mathbf{seq}+}$. To get the proof through, a few mostly technical restrictions of parametrizations and annotation substitutions are necessary. They are given by the next definitions.

DEFINITION 52 Let $\dot{\kappa}$, $\dot{\kappa}'$ (parametrized) terms, types, typing contexts or typing constraints and ϱ, ϱ' annotation substitutions. We define:

- $\mathrm{AV}(\dot{\kappa})$ denotes the *set of annotation variables in* $\dot{\kappa}$.
- $\dot{\kappa}$ is *general* if no annotation variable occurs more than once in $\dot{\kappa}$.
- $\dot{\kappa}$ is *closed under* ϱ if $\mathrm{AV}(\dot{\kappa}) \subseteq dom(\varrho)$.
- ϱ is *tight* w.r.t. $\dot{\kappa}$ if $\mathrm{AV}(\dot{\kappa}) = dom(\varrho)$.
- Annotation substitutions are *disjoint* if their domains are.
- ϱ and ϱ' are *compatible* if $\nu \in dom(\varrho) \cap dom(\varrho')$ implies $\varrho(\nu) = \varrho'(\nu)$.
- $\dot{\kappa}$ and $\dot{\kappa}'$ are *disjoint* if $\mathrm{AV}(\dot{\kappa}) \cap \mathrm{AV}(\dot{\kappa}') = \emptyset$.
- If ϱ and ϱ' are compatible, we define their union, $\varrho \cup \varrho'$, as the union of the graphs.

DEFINITION 53 The notion *tight* extends canonically to parametrized typing judgments, i.e., an annotation substitution is tight w.r.t. a typing judgment if its domain contains exactly the annotation variables that occur in the typing judgment.

Now we are prepared to present a proof of Theorem 10.

First, we ensure that it suffices to consider general, pairwise disjoint parametrizations.

LEMMA 39 For all concrete terms, types or typing contexts $\kappa_1, \ldots, \kappa_n$, $n \in \mathbb{N}$, there exist general, pairwise disjoint parametrizations $\dot{\kappa}_1, \ldots, \dot{\kappa}_n$ and an annotation substitution ϱ, such that $\dot{\kappa}_1\varrho = \kappa_1, \ldots, \dot{\kappa}_n\varrho = \kappa_n$.

Proof. We sketch how to construct $\dot{\kappa}_1, \ldots, \dot{\kappa}_n$ and ϱ. Since all terms, types and typing contexts are finite, each of them has only finitely many annotations. Thus, we can replace each concrete type annotation ($\circ$ or ϵ) in $\kappa_1, \ldots, \kappa_n$ by a fresh annotation variable ν. The resulting parametrized entities are general and pairwise disjoint by construction. A suitable annotation substitution ϱ is defined by mapping each newly introduced annotation variable back to the concrete annotation that was replaced by the variable. □

The next lemmas are grouped by the subsystems of conditional and concrete typing rules that they relate. The proofs for the main rule system will rely on these lemmas. We start with the rules for the Seqable-check, i.e., relate the systems shown in Figure 5.5 and Figure 5.12.

Lemma 40 If $\Gamma \vdash \tau \in \mathsf{Seqable}$ valid, there exist pairwise disjoint, general $\dot{\Gamma}$ and $\dot{\tau}$, a typing constraint C and an annotation substitution ϱ, such that $\dot{\Gamma}\varrho = \Gamma$, $\dot{\tau}\varrho = \tau$, $[\![C\varrho]\!] = \mathit{True}$ and $\langle\dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow (C)$ valid.

Proof. The proof is by induction over the depth of the derivation tree of $\Gamma \vdash \tau \in \mathsf{Seqable}$. It mainly employs Lemma 39. We regard each rule from Figure 5.5 and translate it to a conditional class membership rule in $\lambda^{\alpha}_{\mathbf{seq}^C}$, i.e., a rule shown in Figure 5.12.

For (CS-List), Lemma 39 suffices to get a parametrization that fulfills (CS-List^C) and also a suitable ϱ.

For (CS-Arrow) we employ again Lemma 39 to get general, pairwise disjoint parametrizations for $\dot{\Gamma}$, $\dot{\tau}_1$ and $\dot{\tau}_2$, as well as the corresponding annotation substitution ϱ. Finally, we extend ϱ by a new entry, mapping the variable ν (w.l.o.g. not yet in the $dom(\varrho)$) to ϵ.

For (CS-Var) we take a general parametrization $\dot{\Gamma}$ of Γ and a corresponding annotation substitution ϱ. By (CS-Var^C) we have $\langle\dot{\Gamma} \vdash \alpha \in \mathsf{Seqable}\rangle \Rrightarrow (\nu = \epsilon)$ valid in $\lambda^{\alpha}_{\mathbf{seq}^C}$. By the premise of (CS-Var), we know that ϱ maps the annotation variable at α to ϵ. Hence, we have also $(\nu = \epsilon)\varrho = \mathit{True}$. □

Lemma 41 For every valid $\langle\dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow (C)$ and every ϱ with $dom(\varrho) \supseteq \mathrm{AV}(\dot{\Gamma}) \cup \mathrm{AV}(\dot{\tau})$ and $[\![C\varrho]\!] = \mathit{True}$, we have $\dot{\Gamma}\varrho \vdash \dot{\tau}\varrho \in \mathsf{Seqable}$ valid.

Proof. The proof is by induction on the depth of the derivation tree of $\langle\dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow (C)$. We translate each rule from Figure 5.12 to a rule from Figure 5.5 that is applicable for each concrete instantiation of $\dot{\Gamma}$ and $\dot{\tau}$ that is obtained via an annotation substitution ϱ with $[\![C\varrho]\!] = \mathit{True}$.

The rule (CS-List^C) is translated to (CS-List). Since in (CS-List) no type annotations are checked, the rule is applicable independently of the concrete ϱ.

The rule (CS-Var^C) is translated to (CS-Var). Since every ϱ under which the typing constraint has truth value True has to map ν to $\circ$, (CS-Var) is applicable for the instantiations obtained by each such ϱ.

The rule (CS-ARROW^C) is translated to (CS-ARROW). For the same reasons as for (CS-VAR^C), (CS-ARROW) is applicable for each annotation substitution under which the typing constraint in (CS-ARROW^C) has truth value True. □

Next we relate the systems for subtyping, shown in Figures 5.6 and 5.13.

LEMMA 42 If $\tau \preceq \tau'$ valid then there exist general, disjoint $\dot\tau$ and $\dot\tau'$, a typing constraint C and an annotation substitution ϱ, such that $\dot\tau\varrho = \tau$, $\dot\tau'\varrho = \tau'$, $[\![C\varrho]\!] = \mathit{True}$, and $\langle \dot\tau \preceq \cdot\rangle \Rrightarrow (C, \dot\tau')$, as well as $\langle \cdot \preceq \dot\tau'\rangle \Rrightarrow (C, \dot\tau)$, valid.

Proof. The proof is by induction on the depth of the derivation tree of $\tau \preceq \tau'$.

(S-VAR)

Immediately by Lemma 39 and (S-VAR$^C_1$) and (S-VAR$^C_2$) because the typing constraint has truth value True independently of ϱ.

(S-LIST)

Immediately by the induction hypothesis since in the conclusion, compared to the premise, no new annotations occur.

$(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\}, \nu' \leqslant \nu}$

By the induction hypothesis for the first premise for each rule of the family $(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\}, \nu' \leqslant \nu}$ we have general, disjoint $\dot\tau_1$ and $\dot\tau_1'$, a typing constraint C_1 and an annotation substitution ϱ_1, such that $\dot\tau_1\varrho_1 = \tau_1$, $\dot\tau_1'\varrho_1 = \tau_1'$, $[\![C_1\varrho_1]\!] = \mathit{True}$, and $\langle \dot\tau_1' \preceq \cdot\rangle \Rrightarrow (C_1, \dot\tau_1)$, as well as $\langle \cdot \preceq \dot\tau_1\rangle \Rrightarrow (C_1, \dot\tau_1')$, valid. By the induction hypothesis for the second premise for each rule of the family $(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\}, \nu' \leqslant \nu}$ we have general, disjoint $\dot\tau_2$ and $\dot\tau_2'$, a typing constraint C_2 and an annotation substitution ϱ_2, such that $\dot\tau_2\varrho_2 = \tau_2$, $\dot\tau_2'\varrho_2 = \tau_2'$, $[\![C_2\varrho_2]\!] = \mathit{True}$, and $\langle \dot\tau_2 \preceq \cdot\rangle \Rrightarrow (C_2, \dot\tau_2')$, as well as $\langle \cdot \preceq \dot\tau_2'\rangle \Rrightarrow (C_2, \dot\tau_2)$, valid. Employing these hypotheses, rules (S-ARROW$^C_1$) and (S-ARROW$^C_2$) are applicable and yield the statements $\langle \dot\tau_1 \to^{\nu} \dot\tau_2 \preceq \cdot\rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu' \leqslant \nu), \dot\tau_1' \to^{\nu'} \dot\tau_2')$ and $\langle \cdot \preceq \dot\tau_1' \to^{\nu'} \dot\tau_2'\rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu' \leqslant \nu), \dot\tau_1 \to^{\nu} \dot\tau_2)$. For both, relying on the induction hypotheses and w.l.o.g. assuming that ν and ν' fresh, it holds that $\dot\tau_1 \to^{\nu} \dot\tau_2$ and $\dot\tau_1' \to^{\nu'} \dot\tau_2'$ are general, disjoint parametrizations of the types in the conclusion of each rule of the family $(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\}, \nu' \leqslant \nu}$ and that for ϱ, defined as $\varrho_1 \cup \varrho_2$ extended by entries for ν and ν' according to the concrete rule of the family $(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\epsilon\}, \nu' \leqslant \nu}$, as well as for $C = C_1 \wedge C_2 \wedge \nu' \leqslant \nu$ the statements of Lemma 42 hold.

□

Lemma 43 If $\langle \dot\tau \preceq \cdot \rangle \Rrightarrow (C, \dot\tau')$ or $\langle \cdot \preceq \dot\tau' \rangle \Rrightarrow (C, \dot\tau)$ valid and ϱ such that $dom(\varrho) \supseteq \mathrm{AV}(\dot\tau) \cup \mathrm{AV}(\dot\tau')$ and $[\![C\varrho]\!] = \mathit{True}$, then $\dot\tau\varrho \preceq \dot\tau'\varrho$ valid.

Proof. Induction over the depth of the derivation tree for $\langle \dot\tau \preceq \cdot \rangle \Rrightarrow (C, \dot\tau')$ or for $\langle \cdot \preceq \dot\tau' \rangle \Rrightarrow (C, \dot\tau)$. □

For the conditional equality checks (the rule system in Figure 5.14) there is no concrete counterpart in $\lambda^{\alpha}_{\mathbf{seq}+}$. The typing rules of $\lambda^{\alpha}_{\mathbf{seq}+}$ force syntactic equality of types via identical naming, e.g. in (LCase) in Figure 5.9 τ is used in the second and third premise to denote equality of the types in the different premises. Hence, conditional equality in $\lambda^{\alpha}_{\mathbf{seq}*}$ should coincide with syntactic equality in $\lambda^{\alpha}_{\mathbf{seq}+}$.

Lemma 44 For every type τ there exist disjoint, general parametrizations $\dot\tau$ and $\dot\tau'$, a typing constraint C and an annotation substitution ϱ, such that $\dot\tau\varrho = \dot\tau'\varrho = \tau$, $[\![C\varrho]\!] = \mathit{True}$ and $\langle \dot\tau = \dot\tau' \rangle \Rrightarrow (C)$ valid.

Proof. The proof is by induction on the structure of τ, which is also the structure of the parametrizations $\dot\tau$ and $\dot\tau'$.

τ a type variable

The case is immediately by employing rule (E-Var^C) and setting $\varrho = \emptyset$.

$\tau = \tau_1 \to^{\circ} \tau_2$

By the induction hypotheses there exist $\dot\tau_1$, $\dot\tau_1'$, $\dot\tau_2$, $\dot\tau_2'$, and ϱ_1, ϱ_2, w.l.o.g. disjoint, such that $\langle \dot\tau_1 = \dot\tau_1' \rangle \Rrightarrow (C_1)$ and $\langle \dot\tau_2 = \dot\tau_2' \rangle \Rrightarrow (C_2)$ hold, as well as $\dot\tau_1\varrho_1 = \tau_1$, $\dot\tau_1'\varrho_1 = \tau_1'$, $\dot\tau_2\varrho_2 = \tau_2$, $\dot\tau_2'\varrho_2 = \tau_2'$, $[\![C_1\varrho_1]\!] = \mathit{True}$, and $[\![C_2\varrho_2]\!] = \mathit{True}$. Hence, we can apply (E-Arrow^C) to obtain $\langle \dot\tau_1 \to^{\nu} \dot\tau_2 = \dot\tau_1' \to^{\nu'} \dot\tau_2' \rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu = \nu'))$ valid. For this statement the annotation substitution $(\varrho_1 \cup \varrho_2)[\nu \mapsto \circ, \nu' \mapsto \circ]$ fulfills the conditions from Lemma 44.

All other cases are similar to case $\tau = \tau_1 \to^{\circ} \tau_2$. □

Lemma 45 If $\langle \dot\tau = \dot\tau' \rangle \Rrightarrow (C)$ valid, $dom(\varrho) \supseteq \mathrm{AV}(\dot\tau) \cup \mathrm{AV}(\dot\tau')$ and $[\![C\varrho]\!] = \mathit{True}$ then $\dot\tau\varrho = \dot\tau'\varrho$.

Proof. Induction on the depth of the derivation tree of $\langle \dot\tau = \dot\tau' \rangle \Rrightarrow (C)$. □

Having lemmas for all subsystems, we focus on the main system.

If $\Gamma \vdash t :: \tau$ valid in $\lambda^{\alpha}_{\mathbf{seq}+}$, then there exist pairwise disjoint, general parametrizations $\dot{\Gamma}$, $\dot{t}$, $\dot{\tau}$, a typing constraint C, and an annotation substitution ϱ, such that $\dot{t}\varrho = t$, $\dot{\tau}\varrho = \tau$, $[\![C\varrho]\!] = \mathit{True}$, and $\langle\dot{\Gamma} \vdash \dot{t}\rangle \Rrightarrow (C, \dot{\tau})$ valid.

Lemma 46 (stronger version of Lemma 26)

Proof. The proof is by induction on the depth of the derivation tree of $\Gamma \vdash t :: \tau$ in $\lambda^{\alpha}_{\mathbf{seq}*}$.

(Var$^+$)

By Lemma 42 we have $\dot{\tau}$, $\dot{\tau}'$, C and ϱ^p, such that $\langle\dot{\tau} \preceq \cdot\rangle \Rrightarrow (C, \dot{\tau}')$, $C\varrho^p = \mathit{True}$, $\dot{\tau}\varrho^p = \tau$, and $\dot{\tau}'\varrho^p = \tau'$. By Lemma 39 we can find a general parametrization $\dot{\Gamma}$ of Γ such that $\mathrm{AV}(\dot{\Gamma})$ and $\mathit{dom}(\varrho^p)$ are disjoint. Furthermore, we find an annotation substitution ϱ' that is tight w.r.t. $\dot{\Gamma}$ and satisfies $\dot{\Gamma}\varrho' = \Gamma$. Now with $\varrho = \varrho^p \cup \varrho'$ it holds that $[\![C\varrho]\!] = \mathit{True}$, $\dot{\Gamma}\varrho = \Gamma$, $\dot{\tau}\varrho = \tau$. By (Var^C) we additionally get that $\langle\dot{\Gamma}, x :: \dot{\tau} \vdash x\rangle \Rrightarrow (C, \dot{\tau}')$ is valid.

(Nil$^+$)

Similar to case (Var$^+$).

(Cons)

We can rewrite rule (Cons) to

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau'] \qquad \tau = \tau'}{\Gamma \vdash (t_1 : t_2) :: [\tau]}$$

By the induction hypotheses and Lemma 44 we have general parametrizations $\dot{\Gamma}_1$ and $\dot{\Gamma}_2$ of Γ, $\dot{\tau}_1$ and $\dot{\tau}_2$ of τ, as well as $\dot{\tau}'_1$ and $\dot{\tau}'_2$ of τ'. Furthermore, we have ϱ^p_1, ϱ^p_2 and ϱ^p_3 such that

- $\langle\dot{\Gamma} \vdash \dot{t}_1\rangle \Rrightarrow (C_1, \dot{\tau})$ holds and $[\![C_1\varrho^p_1]\!] = \mathit{True}$, $\dot{\Gamma}\varrho^p_1 = \Gamma$, $\dot{t}_1\varrho^p_1 = t_1$, $\dot{\tau}\varrho^p_1 = \tau$;
- $\langle\dot{\Gamma} \vdash \dot{t}_2\rangle \Rrightarrow (C_2, [\dot{\tau}'])$ holds and $[\![C_2\varrho^p_2]\!] = \mathit{True}$, $\dot{\Gamma}\varrho^p_2 = \Gamma$, $\dot{t}_2\varrho^p_2 = t_2$, $[\dot{\tau}']\varrho^p_2 = [\tau']$ and thus $\dot{\tau}'\varrho^p_2 = \tau'$;
- $\langle\dot{\tau} = \dot{\tau}'\rangle \Rrightarrow (C_3)$ holds and $C_3\varrho^p_3 = \mathit{True}$, $\dot{\tau}\varrho^p_3 = \tau$, $\dot{\tau}'\varrho^p_3 = \tau'$.

We can assume $\mathit{dom}(\varrho^p_1) \cap \mathit{dom}(\varrho^p_2) = \mathrm{AV}(\dot{\Gamma})$, $\mathit{dom}(\varrho^p_1) \cap \mathit{dom}(\varrho^p_3) = \mathrm{AV}(\dot{\tau})$ and $\mathit{dom}(\varrho^p_2) \cap \mathit{dom}(\varrho^p_3) = \mathrm{AV}(\dot{\tau}')$. Therefore, the three annotation substitutions are pairwise compatible and we take $\varrho = \varrho^p_1 \cup \varrho^p_2 \cup \varrho^p_3$. Applying (Cons^C) with the three just stated premises completes the proof case.

For the remaining rules we just point out differences to the already proved cases. Regarding ($\textsc{Abs}^+_\epsilon$) we extend ϱ by the entry $\nu \mapsto \epsilon$. Similarly, for ($\textsc{Abs}^+_\circ$) we add $\nu \mapsto \circ$. □

Lemma 47 (repetition of Lemma 27) If $\langle\dot\Gamma \vdash \dot t\rangle \Rrightarrow (C, \dot\tau)$ valid then for every ϱ with $dom(\varrho) \supseteq \mathrm{AV}(\dot\Gamma) \cup \mathrm{AV}(\dot t) \cup \mathrm{AV}(\dot\tau)$ and $[\![C\varrho]\!] = \mathit{True}$ we have $\dot\Gamma\varrho \vdash \dot t\varrho :: \dot\tau\varrho$ valid in $\lambda^\alpha_{\mathbf{seq+}}$.

Proof. The proof is by induction on the depth of the derivation tree of $\langle\dot\Gamma \vdash \dot t\rangle \Rrightarrow (C, \dot\tau)$. For each rule we consider, the induction hypothesis is that every premise either fulfills Lemma 27 or satisfies one of the Lemmas 41, 43, 45.

We give only three proof cases. The remaining ones cause no additional problems.

($\textsc{Var}^C$)

By Lemma 43 we have $\dot\tau\,\varrho \preceq \dot{\tau'}\,\varrho$ valid for all ϱ with $\mathrm{AV}(\dot\tau) \cup \mathrm{AV}(\dot{\tau'}) \subseteq dom(\varrho)$ and $[\![C\varrho]\!] = \mathit{True}$. Hence, this holds also for all ϱ with $\mathrm{AV}(\dot\tau) \cup \mathrm{AV}(\dot{\tau'}) \cup \mathrm{AV}(\dot\Gamma) \subseteq dom(\varrho)$ and $[\![C\varrho]\!] = \mathit{True}$. Thus, whenever the conditions of Lemma 27 are fulfilled, we can apply ($\textsc{Var}^+$) and obtain the proof obligation.

($\textsc{Cons}^C$)

From the premises we get

$$\begin{gathered}\forall\varrho \in \mathrm{P}_1.\ \langle\dot\Gamma \vdash \dot{t_1}\rangle \Rrightarrow (C_1, \dot\tau) \Rightarrow \dot\Gamma\varrho \vdash \dot{t_1}\varrho :: \dot\tau\varrho,\\ \forall\varrho \in \mathrm{P}_2.\ \langle\dot\Gamma \vdash \dot{t_2}\rangle \Rrightarrow (C_2, [\dot{\tau'}]) \Rightarrow \dot\Gamma\varrho \vdash \dot{t_2}\varrho :: [\dot{\tau'}]\varrho,\\ \forall\varrho \in \mathrm{P}_3.\ \langle\dot\tau = \dot{\tau'}\rangle \Rrightarrow (C_3) \Rightarrow \dot\tau\varrho = \dot{\tau'}\varrho\end{gathered}$$

with

$$\begin{gathered}\mathrm{P}_1 = \{\varrho \mid \mathrm{AV}(\dot\Gamma) \cup \mathrm{AV}(\dot{t_1}) \cup \mathrm{AV}(\dot\tau) \subseteq dom(\varrho) \wedge ([\![C_1\varrho]\!] = \mathit{True})\},\\ \mathrm{P}_2 = \{\varrho \mid \mathrm{AV}(\dot\Gamma) \cup \mathrm{AV}(\dot{t_2}) \cup \mathrm{AV}(\dot{\tau'}) \subseteq dom(\varrho) \wedge ([\![C_2\varrho]\!] = \mathit{True})\},\\ \mathrm{P}_3 = \{\varrho \mid \mathrm{AV}(\dot\tau) \cup \mathrm{AV}(\dot{\tau'}) \subseteq dom(\varrho) \wedge ([\![C_3\varrho]\!] = \mathit{True})\}.\end{gathered}$$

Hence, rewriting ($\textsc{Cons}$) as in the proof of Lemma 26, making syntactic equality of types explicit by the additional premise $\tau = \tau'$, the induction hypotheses guarantee that for each $\varrho \in \mathrm{P}_1 \cap \mathrm{P}_2 \cap \mathrm{P}_3$ and $\dot\Gamma\varrho$, $\dot{t_1}\varrho$, $\dot{t_2}\varrho$, $\dot\tau\varrho$, $\dot{\tau'}\varrho$ the premises of the rewritten ($\textsc{Cons}$) are valid. Consequently, in such cases we can apply the rule to obtain $\dot\Gamma\varrho \vdash \dot{t_1}\varrho : \dot{t_2}\varrho :: [\dot\tau\varrho]$ valid.
Restricting the domain of ϱ to the annotation variables that are really looked up, and distributivity of ϱ, we obtain the proof obligation, i.e., $\dot\Gamma \vdash (\dot{t_1} : \dot{t_2})\varrho :: [\dot\tau]\varrho$ for each $\varrho \in \{\varrho \mid \mathrm{AV}(\dot\Gamma) \cup \mathrm{AV}(\dot{t_1}) \cup \mathrm{AV}(\dot{t_2}) \cup \mathrm{AV}(\dot\tau) \subseteq dom(\varrho) \wedge [\![(C_1 \wedge C_2 \wedge C_3)\varrho]\!] = \mathit{True}\}$.

(ABS^C)

From the premises we get

$$\forall \varrho \in \mathrm{P}_1.\ \langle \dot\Gamma, x :: \dot\tau_1 \vdash \dot t\rangle \Rrightarrow (C_1, \dot\tau_2) \Rightarrow \dot\Gamma\varrho, x :: \dot\tau_1\varrho \vdash \dot t\varrho :: \dot\tau_2\varrho,$$
$$\forall \varrho \in \mathrm{P}_2.\ \langle \cdot \preceq \dot\tau_1\rangle \Rrightarrow (C_1, \dot\tau_1') \Rightarrow \dot\tau_1'\varrho \preceq \dot\tau_1\varrho$$

with

$$\mathrm{P}_1 = \{\varrho \mid \mathrm{AV}(\dot\Gamma) \cup \mathrm{AV}(\dot\tau_1) \cup \mathrm{AV}(\dot\tau_2) \subseteq dom(\varrho) \wedge (\llbracket C_1\varrho\rrbracket = True)\},$$
$$\mathrm{P}_2 = \{\varrho \mid \mathrm{AV}(\dot\tau_1) \cup \mathrm{AV}(\dot\tau_1') \subseteq dom(\varrho) \wedge (\llbracket C_2\varrho\rrbracket = True)\}.$$

Hence, by the induction hypotheses we can apply (ABS$_\circ$) or (ABS$_\epsilon$) in $\lambda^{\alpha}_{\mathbf{seq}*}$. For (ABS$_\circ$) we get (with distributivity of ϱ) that $\dot\Gamma\varrho \vdash (\lambda x :: \dot\tau_1.\dot t)\varrho :: (\dot\tau_1' \to^\nu \dot\tau_2)\varrho$ is valid for all $\varrho \in \mathrm{P}_1 \cap \mathrm{P}_2 \cap \{\varrho \mid \varrho(\nu) = \circ\}$. For (ABS$_\epsilon$)we get (again with distributivity of ϱ) that $\dot\Gamma\varrho \vdash (\lambda x :: \dot\tau_1.\dot t)\ \varrho :: (\dot\tau_1' \to^\nu \dot\tau_2)\ \varrho$ is valid for all $\varrho \in \mathrm{P}_1 \cap \mathrm{P}_2 \cap \{\varrho \mid \varrho(\nu) = \epsilon\}$.
Therefore, we have $\varrho \in \mathrm{P}_1 \cap \mathrm{P}_2 \cap \{\varrho \mid \varrho(\nu) = \epsilon\}$ valid for all $\varrho \in \{\varrho \mid \mathrm{AV}(\dot\Gamma) \cup \mathrm{AV}(\dot\tau_1) \cup \mathrm{AV}(\dot t) \cup \mathrm{AV}(\dot\tau_1' \to^\nu \dot\tau_2) \subseteq dom(\varrho) \wedge (\llbracket C_1 \wedge C_2\rrbracket = True)\}$, which is the proof obligation.

□

Proof (Theorem 10). Equivalence, in the sense of typeability, is directly by Definition 41, Lemma 26 and Lemma 27. □

A.3 Proofs from Chapter 6

Proof of Lemma 30 on page 155

Proof.

$$(\mathbf{f}, \mathbf{g}) \in \mathcal{C}(\Delta'_{\tau_1\to\tau_2,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$cost(\mathbf{f}) = cost(\mathbf{g}) \wedge (val(\mathbf{f}), val(\mathbf{g})) \in \Delta'_{\tau_1\to\tau_2,\rho}$$

$\Leftrightarrow$ $\left\{ \text{definition of } \Delta'_{\tau_1\to\tau_2,\cdot} \right\}$

$$cost(\mathbf{f}) = cost(\mathbf{g}) \wedge \forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1,\rho}.\ (val(\mathbf{f})\ \mathbf{v}, val(\mathbf{g})\ \mathbf{v}') \in \mathcal{C}(\Delta'_{\tau_2,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting and $val(\cdot)$ }

$$cost(\mathbf{f}) = cost(\mathbf{g}) \wedge \forall (\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}).$$
$$(val(\mathbf{f})\ val(\mathbf{x}), val(\mathbf{g})\ val(\mathbf{y})) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$\begin{aligned}
& \mathit{cost}(\mathbf{f}) = \mathit{cost}(\mathbf{g}) \wedge \forall(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}).\\
& \quad (\mathit{cost}(\mathbf{x}) \triangleright (\mathit{val}(\mathbf{f})\ \mathit{val}(\mathbf{x})), \mathit{cost}(\mathbf{y}) \triangleright (\mathit{val}(\mathbf{g})\ \mathit{val}(\mathbf{y}))) \in \mathcal{C}(\Delta'_{\tau_2,\rho})\\
& \Leftrightarrow \quad \{ \text{ definition of cost-lifting } \}\\
& \mathit{cost}(\mathbf{f}) = \mathit{cost}(\mathbf{g}) \wedge \forall(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}).\\
& \quad ((\mathit{cost}(\mathbf{f}) + \mathit{cost}(\mathbf{x})) \triangleright (\mathit{val}(\mathbf{f})\ \mathit{val}(\mathbf{x})),\\
& \quad\quad (\mathit{cost}(\mathbf{g}) + \mathit{cost}(\mathbf{y})) \triangleright (\mathit{val}(\mathbf{g})\ \mathit{val}(\mathbf{y}))) \in \mathcal{C}(\Delta'_{\tau_2,\rho})\\
& \Leftrightarrow \quad \{ \text{ definition of } ¢ \ \}\\
& \mathit{cost}(\mathbf{f}) = \mathit{cost}(\mathbf{g}) \wedge \forall(\mathbf{x},\mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}).\ (\mathbf{f}\ ¢\ \mathbf{x}, \mathbf{g}\ ¢\ \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})
\end{aligned}$$

□

Proof of Theorem 11 on page 154

Proof. The proof is by induction over the type derivation, i.e., we have to consider the derivation rules in Figures 2.2 and 6.3. In the proof we use the same names for terms and types as in the figures with the typing rules; we name environments as in Theorem 11 (i.e., ρ, σ_1, σ_2) and we assume the conditions on them that are given in Theorem 11 to be satisfied.

(VAR)

$$\begin{aligned}
& ([\![x]\!]^{¢}_{\sigma_1}, [\![x]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})\\
& \quad \Leftrightarrow \quad \{ \text{ term semantics } \}\\
& ((\sigma_1(x), 0), (\sigma_2(x), 0)) \in \mathcal{C}(\Delta'_{\tau,\rho})\\
& \quad \Leftrightarrow \quad \{ \text{ definition of the cost-lifting } \}\\
& (\sigma_1(x), \sigma_2(x)) \in \Delta'_{\tau,\rho}
\end{aligned}$$

The last statement is true by the second condition of Theorem 11.

(NAT)

$$\begin{aligned}
& ([\![n]\!]^{¢}_{\sigma_1}, [\![n]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{Nat,\rho})\\
& \quad \Leftrightarrow \quad \{ \text{ term semantics } \}\\
& ((\mathbf{n}, 0), (\mathbf{n}, 0)) \in \mathcal{C}(\Delta'_{Nat,\rho})\\
& \quad \Leftrightarrow \quad \{ \text{ definition of the cost-lifting } \}\\
& (\mathbf{n}, \mathbf{n}) \in \Delta'_{Nat,\rho}
\end{aligned}$$

The last statement is true by the definition of $\Delta'_{Nat,\rho}$.

(NIL)

Similar to case (NAT).

(SUM)

$([\![t_1 + t_2]\!]^{¢}_{\sigma_1}, [\![t_1 + t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{Nat,\rho})$

$\Leftrightarrow$ { term semantics, definition of $(+^{¢})$ }

$((\mathbf{n_1} + \mathbf{n_2}, c_1 + c_2), (\mathbf{n'_1} + \mathbf{n'_2}, c'_1 + c'_2)) \in \mathcal{C}(\Delta'_{Nat,\rho})$

where $(\mathbf{n_1}, c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1}, c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2}, c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2}, c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftrightarrow$ { definition of cost-lifting and $\Delta'_{Nat,\rho}$ }

$\mathbf{n_1} + \mathbf{n_2} = \mathbf{n'_1} + \mathbf{n'_2} \wedge c_1 + c_2 = c'_1 + c'_2$

where $(\mathbf{n_1}, c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1}, c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2}, c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2}, c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftarrow$ { sum of equal addends is equal }

$\mathbf{n_1} = \mathbf{n'_1} \wedge \mathbf{n_2} = \mathbf{n'_2} \wedge c_1 = c'_1 \wedge c_2 = c'_2$

where $(\mathbf{n_1}, c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1}, c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2}, c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2}, c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftrightarrow$ { definition of $\Delta'_{Nat,\rho}$ and cost-lifting }

$((\mathbf{n_1}, c_1), (\mathbf{n'_1}, c'_1)) \in \mathcal{C}(\Delta'_{Nat,\rho}) \wedge ((\mathbf{n_2}, c_2), (\mathbf{n'_2}, c'_2)) \in \mathcal{C}(\Delta'_{Nat,\rho})$

where $(\mathbf{n_1}, c_1) = [\![t_1]\!]^{¢}_{\sigma_1}, (\mathbf{n'_1}, c'_1) = [\![t_1]\!]^{¢}_{\sigma_2}, (\mathbf{n_2}, c_2) = [\![t_2]\!]^{¢}_{\sigma_1}, (\mathbf{n'_2}, c'_2) = [\![t_2]\!]^{¢}_{\sigma_2}$

$\Leftrightarrow$ { term semantics }

$([\![t_1]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{Nat,\rho}) \wedge ([\![t_2]\!]^{¢}_{\sigma_1}, [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{Nat,\rho})$

The last statement is true by the induction hypotheses.

(CONS)

Similar to case (SUM).

(NCASE)

The proof is by case distinction on the value of $[\![t]\!]^{¢}_{\sigma_1}$. First, we assume $[\![t]\!]^{¢}_{\sigma_1} = (\mathbf{0}, c)$ for an arbitrary $c \in \mathbb{Z}$. Taking the induction hypothesis from the first premise and the definition of the logical relation for type Nat into account, we know $[\![t]\!]^{¢}_{\sigma_2} = (\mathbf{0}, c)$, too. Hence, evaluating the semantics of the case expression reduces to evaluating the first branch of the semantics of the case expression:

$([\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_1}, [\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$

$\Leftrightarrow$ { term semantics with choice of the first branch }

$(c \triangleright [\![t_1]\!]^{¢}_{\sigma_1}, c \triangleright [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$

$\Leftrightarrow$ { definition of cost-lifting }

$([\![t_1]\!]^{¢}_{\sigma_1}, [\![t_1]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$

The last statement is by the induction hypothesis from the second premise. Second, we take $[\![t]\!]^{¢}_{\sigma_1} = (\mathbf{n}, c)$ for $\mathbf{n} > \mathbf{0}$ and $c \in \mathbb{Z}$ arbitrary. Again, using the induction hypothesis from the first premise we have $[\![t]\!]^{¢}_{\sigma_2} = (\mathbf{n}, c)$, too. Hence, we can reason as follows:

$([\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_1}, [\![\mathbf{case}\ t\ \mathbf{of}\ \{0 \to t_1; \_ \to t_2\}]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$

$\Leftrightarrow$ { term semantics with choice of the second branch }

$(c \triangleright [\![t_2]\!]^{¢}_{\sigma_1}, c \triangleright [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$

$\Leftrightarrow$ { definition of cost-lifting }

$([\![t_2]\!]^{¢}_{\sigma_1}, [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$

The last statement is true by the induction hypothesis from the third premise.

(LCASE)

We need a case distinction via the semantics of t similar to the induction step for (NCASE). The case $[\![t]\!]^{¢}_{\sigma_1} = ([\,], c)$ with $c \in \mathbb{Z}$ arbitrary is similar to the first case in the proof for (NCASE).
The case $[\![t]\!]^{¢}_{\sigma_1} = (\mathbf{v} : \mathbf{vs}, c)$ requires some extra care w.r.t. the term environments. The induction hypothesis from the first premise (using the definition of the logical relation for list types) ensures that $[\![t]\!]^{¢}_{\sigma_2} = (\mathbf{v}' : \mathbf{vs}', c)$ with $(\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau,\rho}$ and $(\mathbf{vs}, \mathbf{vs}') \in \Delta'_{[\tau],\rho}$. Thus, under both environments σ_1 and σ_2, semantic evaluation of the case expression reduces to the evaluation of the second branch of its semantics:

$$([\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1; x_1 : x_2 \to t_2\}]\!]^{¢}_{\sigma_1},$$
$$[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1; x_1 : x_2 \to t_2\}]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

$\Leftrightarrow$ { term semantics with choice of the second branch }

$$(c \triangleright [\![t_2]\!]^{¢}_{\sigma_1[x_1 \mapsto \mathbf{v}, x_2 \mapsto \mathbf{vs}]}, c \triangleright [\![t_2]\!]^{¢}_{\sigma_2[x_1 \mapsto \mathbf{v}', x_2 \mapsto \mathbf{vs}']}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$([\![t_2]\!]^{¢}_{\sigma_1[x_1 \mapsto \mathbf{v}, x_2 \mapsto \mathbf{vs}]}, [\![t_2]\!]^{¢}_{\sigma_2[x_1 \mapsto \mathbf{v}', x_2 \mapsto \mathbf{vs}']}) \in \mathcal{C}(\Delta'_{\tau,\rho})$$

Because, as already stated, $(\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau,\rho}$ and $(\mathbf{vs}, \mathbf{vs}') \in \Delta'_{[\tau],\rho}$ are ensured, we end up with (a statement covered by) the induction hypothesis from the third premise.

(PAIR)

Similar to case (SUM).

(PCASE)

Similar to the second case of case (LCASE).

(ABS)

$$(\llbracket \lambda x :: \tau_1 . t \rrbracket^{¢}_{\sigma_1}, \llbracket \lambda x :: \tau_1 . t \rrbracket^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_1 \to \tau_2, \rho})$$

$\Leftrightarrow$ { term semantics }

$$((\lambda \mathbf{v}.1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, 0), (\lambda \mathbf{v}'.1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}, 0)) \in \mathcal{C}(\Delta'_{\tau_1 \to \tau_2, \rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$(\lambda \mathbf{v}.1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, \lambda \mathbf{v}.1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_2[x \mapsto \mathbf{v}]}) \in \Delta'_{\tau_1 \to \tau_2, \rho}$$

$\Leftrightarrow$ { definition of $\Delta'_{\tau_1 \to \tau_2, \cdot}$ }

$$\forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1, \rho} . ((\lambda \mathbf{v}.1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_1[x \mapsto \mathbf{v}]})\ \mathbf{v}, (\lambda \mathbf{v}.1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_2[x \mapsto \mathbf{v}]})\ \mathbf{v}') \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftrightarrow$ { function application }

$$\forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1, \rho} . (1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, 1 \rhd \llbracket t \rrbracket^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$\forall (\mathbf{v}, \mathbf{v}') \in \Delta'_{\tau_1, \rho} . (\llbracket t \rrbracket^{¢}_{\sigma_1[x \mapsto \mathbf{v}]}, \llbracket t \rrbracket^{¢}_{\sigma_2[x \mapsto \mathbf{v}']}) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

The last statement is true by the induction hypothesis.

(APP)

$$(\llbracket t_1\ t_2 \rrbracket^{¢}_{\sigma_1}, \llbracket t_1\ t_2 \rrbracket^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftrightarrow$ { term semantics }

$$(\llbracket t_1 \rrbracket^{¢}_{\sigma_1} \ ¢\ \llbracket t_2 \rrbracket^{¢}_{\sigma_1}, \llbracket t_1 \rrbracket^{¢}_{\sigma_2} \ ¢\ \llbracket t_2 \rrbracket^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftarrow$ { induction hypothesis from the second premise }

$$\forall (\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1, \rho}) . (\llbracket t_1 \rrbracket^{¢}_{\sigma_1} \ ¢\ \mathbf{x}, \llbracket t_1 \rrbracket^{¢}_{\sigma_2} \ ¢\ \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftarrow$ { Lemma 30 }

$$(\llbracket t_1 \rrbracket^{¢}_{\sigma_1}, \llbracket t_1 \rrbracket^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_1 \to \tau_2, \rho})$$

The last statement is true by the induction hypothesis from the first premise.

(LFOLD)

We take $[\![t_1]\!]^{¢}_{\sigma_1} = (\mathbf{f}, c_1)$ and thus have $[\![t_1]\!]^{¢}_{\sigma_2} = (\mathbf{f}', c_1)$ with $(\mathbf{f}, \mathbf{f}') \in \Delta'_{\tau_1 \to \tau_2 \to \tau_2, \rho}$ by the induction hypothesis from the first premise. By the induction hypothesis from the third premise, taking $[\![t_3]\!]^{¢}_{\sigma_1} = ([\mathbf{v_1}, \ldots, \mathbf{v_n}], c_3)$, we have $[\![t_3]\!]^{¢}_{\sigma_2} = ([\mathbf{v'_1}, \ldots, \mathbf{v'_n}], c_3)$ for the same n and c_3 and with $\{(\mathbf{v_1}, \mathbf{v'_1}), \ldots, (\mathbf{v_n}, \mathbf{v'_n})\} \subseteq \Delta'_{\tau_1, \rho}$. We reason as follows:

$$([\![\mathbf{lfold}(t_1, t_2, t_3)]\!]^{¢}_{\sigma_1}, [\![\mathbf{lfold}(t_1, t_2, t_3)]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftrightarrow$ { term semantics }

$$((c_1 + c_3) \triangleright ((\mathbf{f}\ \mathbf{v_1}) ¢ (\ldots ((\mathbf{f}\ \mathbf{v_n}) ¢ [\![t_2]\!]^{¢}_{\sigma_1}))),$$
$$(c_1 + c_3) \triangleright ((\mathbf{f}'\ \mathbf{v'_1}) ¢ (\ldots ((\mathbf{f}'\ \mathbf{v'_n}) ¢ [\![t_2]\!]^{¢}_{\sigma_2})))) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$(((\mathbf{f}\ \mathbf{v_1}) ¢ (\ldots ((\mathbf{f}\ \mathbf{v_n}) ¢ [\![t_2]\!]^{¢}_{\sigma_1}))),$$
$$((\mathbf{f}'\ \mathbf{v'_1}) ¢ (\ldots ((\mathbf{f}'\ \mathbf{v'_n}) ¢ [\![t_2]\!]^{¢}_{\sigma_2})))) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

$\Leftarrow$ $\{$ side conditions on $\mathbf{f}$, $\mathbf{f}'$, $\mathrm{v_i}$, $\mathrm{v'_i}$, cost-lifting, definition of $\Delta'_{\tau_1 \to \tau_2, \rho}$ $\}$

$$([\![t_2]\!]^{¢}_{\sigma_1}, [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau_2, \rho})$$

The last statement is true by the induction hypothesis from the second premise.

(NFOLD)

Taking $[\![t_1]\!]^{¢}_{\sigma_1} = (\mathbf{f}, c_1)$ we have $[\![t_1]\!]^{¢}_{\sigma_2} = (\mathbf{f}', c_1)$ with $(\mathbf{f}, \mathbf{f}') \in \Delta'_{\tau \to \tau, \rho}$ by the induction hypothesis from the first premise. By the induction hypothesis from the third premise, taking $[\![t_3]\!]^{¢}_{\sigma_1} = (\mathbf{n}, c_3)$, we have $[\![t_3]\!]^{¢}_{\sigma_2} = (\mathbf{n}, c_3)$, too. We can reason as follows:

$$([\![\mathbf{ifold}(t_1, t_2, t_3)]\!]^{¢}_{\sigma_1}, [\![\mathbf{ifold}(t_1, t_2, t_3)]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau, \rho})$$

$\Leftrightarrow$ { term semantics }

$$((c_1 + c_3) \triangleright (\underbrace{(\mathbf{f}, 0) ¢ (\ldots ((\mathbf{f}, 0) ¢}_{\mathbf{n}\text{ times}} [\![t_2]\!]^{¢}_{\sigma_1}))),$$
$$(c_1 + c_3) \triangleright (\underbrace{(\mathbf{f}', 0) ¢ (\ldots ((\mathbf{f}', 0) ¢}_{\mathbf{n}\text{ times}} [\![t_2]\!]^{¢}_{\sigma_2})))) \in \mathcal{C}(\Delta'_{\tau, \rho})$$

$\Leftrightarrow$ { definition of cost-lifting }

$$((\underbrace{(\mathbf{f}, 0) ¢ (\ldots ((\mathbf{f}, 0) ¢}_{\mathbf{n}\text{ times}} [\![t_2]\!]^{¢}_{\sigma_1}))), (\underbrace{(\mathbf{f}', 0) ¢ (\ldots ((\mathbf{f}', 0) ¢}_{\mathbf{n}\text{ times}} [\![t_2]\!]^{¢}_{\sigma_2})))) \in \mathcal{C}(\Delta'_{\tau, \rho})$$

$\Leftarrow$ $\{$ side conditions on $\mathbf{f}$, $\mathbf{f}'$, cost-lifting, definition of $\Delta'_{\tau \to \tau, \rho}$ $\}$

$$([\![t_2]\!]^{¢}_{\sigma_1}, [\![t_2]\!]^{¢}_{\sigma_2}) \in \mathcal{C}(\Delta'_{\tau, \rho})$$

The last statement is true by the induction hypothesis from the second premise.

□

Proof of Lemma 31 on page 158

Proof. Let $\mathbf{x} = ((\mathbf{x_1}, \mathbf{x_2}), c_x)$ and $\mathbf{y} = ((\mathbf{y_1}, \mathbf{y_2}), c_y)$. By $(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{(\tau_1,\tau_2),\rho})$ we know $c_x = c_y$, $(\mathbf{x_1}, \mathbf{y_1}) \in \Delta'_{\tau_1,\rho}$ and $(\mathbf{x_2}, \mathbf{y_2}) \in \Delta'_{\tau_2,\rho}$. We define $\mathbf{p} = (\mathbf{x_1}, c_1)$, $\mathbf{p}' = (\mathbf{x_2}, c_2)$, $\mathbf{q} = (\mathbf{y_1}, c_1)$, $\mathbf{q}' = (\mathbf{y_2}, c_2)$ with $c_1 + c_2 = c_x = c_y$ to get the forward direction of Lemma 31.

To show the backward direction of Lemma 31, we take $\mathbf{p} = (\mathbf{x_1}, c_1)$, $\mathbf{p}' = (\mathbf{x_2}, c_2)$, $\mathbf{q} = (\mathbf{y_1}, c_1)$, $\mathbf{q}' = (\mathbf{y_2}, c_2)$. By the definition of $(\cdot,\cdot)^{¢}$ we get $\mathbf{x} = ((\mathbf{x_1}, \mathbf{x_2}), c_1 + c_2)$ and $\mathbf{y} = ((\mathbf{y_1}, \mathbf{y_2}), c_1 + c_2)$. Since $(\mathbf{p}, \mathbf{q}) \in \mathcal{C}(\Delta'_{\tau_1,\rho})$ and $(\mathbf{p}', \mathbf{q}') \in \mathcal{C}(\Delta'_{\tau_2,\rho})$, we have $(\mathbf{x_1}, \mathbf{y_1}) \in \Delta'_{\tau_1,\rho}$ and $(\mathbf{x_2}, \mathbf{y_2}) \in \Delta'_{\tau_2,\rho}$, which, already knowing that $\mathbf{x}$ and $\mathbf{y}$ have the same costs, is sufficient for $(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{(\tau_1,\tau_2),\rho})$. □

Proof of Lemma 32 on page 158

Proof. First we prove the forward direction. Since $(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{[\tau],\rho})$ we know (by the definition of cost-lifting and the definition of $\Delta'_{[\tau],\rho}$) that there exist $c \in \mathbb{Z}$, $n \in \mathbb{N}$ and $\{(\mathbf{v_1}, \mathbf{v'_1}), \ldots (\mathbf{v_n}, \mathbf{v'_n})\} \subseteq \Delta'_{\tau,\rho}$, such that we have $(\mathbf{x}, \mathbf{y}) = (([\mathbf{v_1}, \ldots, \mathbf{v_n}], c), ([\mathbf{v'_1}, \ldots, \mathbf{v'_n}], c))$. If we choose $(\mathbf{x_1}, \mathbf{y_1}) = ((\mathbf{v_1}, 0), (\mathbf{v'_1}, 0)), \ldots$, $(\mathbf{x_n}, \mathbf{y_n}) = ((\mathbf{v_n}, 0), (\mathbf{v'_n}, 0))$, then by the definition of $[\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}$ and the term semantics for lists we get the right-hand side of Lemma 32: $(\mathbf{x}, \mathbf{y}) = (c \triangleright [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}, c \triangleright [\mathbf{y_1}, \ldots, \mathbf{y_n}]^{¢})$.

Now consider the backward direction. For every $n \in \mathbb{N}$ and $i \in \{1, \ldots, n\}$ we have $(\mathbf{x_i}, \mathbf{y_i}) \in \mathcal{C}(\Delta'_{\tau,\rho})$ and hence, by the definition of the cost-lifting, there exist $(\mathbf{v_i}, \mathbf{v'_i}) \in \Delta'_{\tau,\rho}$ and $c_i \in \mathbb{Z}$ such that $(\mathbf{x_i}, \mathbf{y_i}) = ((\mathbf{v_i}, c_i), (\mathbf{v'_i}, c_i))$. Consequently, exploring the definition of $[\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}$ and the term semantics for lists, we get for all $c \in \mathbb{Z}$ that $(c \triangleright [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}, c \triangleright [\mathbf{x_1}, \ldots, \mathbf{x_n}]^{¢}) = (([\mathbf{v_1}, \ldots, \mathbf{v_n}], c + c_1 + \ldots + c_n), ([\mathbf{v'_1}, \ldots, \mathbf{v'_n}], c + c_1 + \ldots + c_n))$, which is in $\mathcal{C}(\Delta'_{[\tau],\rho})$ by the definition of cost-lifting and of $\Delta'_{[\tau],\rho}$. □

Proof of Lemma 34 on page 161

Proof. Let $\mathbf{x_1}, \mathbf{x_2} \in \mathcal{C}(S_1)$ such that $val(\mathbf{x_1}) = val(\mathbf{x_2}) = \mathbf{v}$. Furthermore, let $\mathbf{f} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2))$. By the definition of $appCost$ and $(¢)$, we get

$$\begin{aligned} & appCost(\mathbf{f}, \mathbf{x_1}) \\ = {} & cost(\mathbf{f} \mathbin{¢} \mathbf{x_1}) - cost(\mathbf{x_1}) \end{aligned}$$

$$\begin{aligned}
&= cost((cost(\mathbf{f}) + cost(\mathbf{x_1})) \rhd val(\mathbf{f})\ val(\mathbf{v})) - cost(\mathbf{x_1}) \\
&= cost(\mathbf{f}) + cost(\mathbf{x_1}) + cost(val(\mathbf{f})\ val(\mathbf{v})) - cost(\mathbf{x_1}) \\
&= cost(\mathbf{f}) + cost(val(\mathbf{f})\ val(\mathbf{v}))
\end{aligned}$$

and by similar reasoning we get

$$appCost(\mathbf{f}, \mathbf{x_2}) = cost(\mathbf{f}) + cost(val(\mathbf{f})\ val(\mathbf{v}))$$

□

Proof of Lemma 35 on page 161

Proof. Let S_1, S_2 sets, $\mathbf{g} \in \mathcal{C}(S_1 \to \mathcal{C}(S_2))$ and $\mathbf{x_1}, \dots, \mathbf{x_n} \in \mathcal{C}(S_1)$.

$$\begin{aligned}
&(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\dots,\mathbf{x_n}}) \\
&\Leftrightarrow \quad \{ \text{ definition of cost-lifting } \} \\
&cost(\mathbf{x}) = cost(\mathbf{y}) \wedge (val(\mathbf{x}), val(\mathbf{y})) \in \mathcal{R}^{\mathbf{g}}_{\mathbf{x_1},\dots,\mathbf{x_n}} \\
&\Leftrightarrow \quad \{ \text{ Definition 43 } \} \\
&\exists i \in \{1, \dots, n\}.\ cost(\mathbf{x}) = cost(\mathbf{y}) \\
&\qquad \wedge\ val(\mathbf{x}) = val(\mathbf{x_i}) \wedge val(\mathbf{y}) = val(\mathbf{g} \cent \mathbf{x_i}) \\
&\Leftrightarrow \quad \{ \text{ definition of } \rhd \text{ and } val(\cdot) \} \\
&\exists i \in \{1, \dots, n\}.\ cost(\mathbf{x}) = cost(\mathbf{y}) \\
&\qquad \wedge val(\mathbf{x}) = val(appCost(\mathbf{g}, \mathbf{x}) \rhd \mathbf{x_i}) \wedge val(\mathbf{y}) = val(\mathbf{g} \cent \mathbf{x_i}) \\
&\Leftrightarrow \quad \{ \text{ Definition 44 } \} \\
&\exists i \in \{1, \dots, n\}, c \in \mathbb{Z}.\ cost(\mathbf{x}) = c + cost(appCost(\mathbf{g}, \mathbf{x}) \rhd \mathbf{x_i}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad = c + cost(\mathbf{g} \cent \mathbf{x_i}) = cost(\mathbf{y}) \\
&\qquad \wedge val(\mathbf{x}) = val(appCost(\mathbf{g}, \mathbf{x}) \rhd \mathbf{x_i}) \wedge val(\mathbf{y}) = val(\mathbf{g} \cent \mathbf{x_i}) \\
&\Leftrightarrow \quad \{ \text{ definition of } val(\cdot) \text{ and } cost(\cdot) \} \\
&\exists i \in \{1, \dots, n\}, c \in \mathbb{Z}.\ \mathbf{x} = c \rhd appCost(\mathbf{g}, \mathbf{x_i}) \rhd \mathbf{x_i} \wedge \mathbf{y} = c \rhd (\mathbf{g} \cent \mathbf{x_i})
\end{aligned}$$

□

Proof of Corollary 5 on page 161

Proof. By Lemma 35 we know that there exist c and $\mathbf{x_i}$ such that we have $\mathbf{x} = c \rhd appCost(\mathbf{g}, \mathbf{x_i}) \rhd \mathbf{x_i}$ and $\mathbf{y} = c \rhd (\mathbf{g} \cent \mathbf{x_i})$. With these characterizations, we reason as follows:

$$\begin{aligned}
&\mathbf{g} \cent \mathbf{x} \\
&= \mathbf{g} \cent (c \rhd appCost(\mathbf{g}, \mathbf{x_i}) \rhd \mathbf{x_i}) \\
&= c \rhd appCost(\mathbf{g}, \mathbf{x_i}) \rhd (\mathbf{g} \cent \mathbf{x_i}) \\
&= appCost(\mathbf{g}, \mathbf{x_i}) \rhd c \rhd (\mathbf{g} \cent \mathbf{x_i}) \\
&= appCost(\mathbf{g}, \mathbf{x_i}) \rhd \mathbf{y}
\end{aligned}$$

□

Proof of Lemma 38 on page 169

Proof. First, we prove statement *(a)*. For the proof we simply calculate the value parts of the semantics of the two applications. Therefore, we need the semantics of (:). We have

$$[\![(:)]\!]^{¢}_{\emptyset} = [\![\lambda x :: \alpha.\lambda xs :: [\alpha].x : xs]\!]^{¢}_{\emptyset} = (\lambda\mathbf{x}.(\lambda\mathbf{xs}.(\mathbf{x} : \mathbf{xs}, 1), 1), 0)$$

Employing the just calculated semantics and the semantics of $(:^{¢\mathbf{k}})$, stated in Definition 46, we get that for all $\mathbf{x} \in \mathcal{C}([\![\tau]\!]^{¢}_{\emptyset})$ and $\mathbf{xs} \in \mathcal{C}([\![[\tau]]\!]^{¢}_{\emptyset})$ both value parts in statement *(a)* evaluate to $val(\mathbf{x}) : val(\mathbf{xs})$.

Second, we prove statement *(b)*. Above Definition 46 we already unfolded the relation $\Delta^{¢}_{\tau\to\alpha\to\alpha,[\alpha\mapsto\mathcal{R}^{\mathbf{fkz}}]}$. Regarding the unfolded version, we have to show

(i) $cost((:^{¢\mathbf{k}})) = cost(\mathbf{k})$

(ii) $\forall \mathbf{u} \in \mathcal{C}([\![\tau]\!]^{¢}_{\emptyset}).\ cost((:^{¢\mathbf{k}}) \mathbin{¢} \mathbf{u}) = cost(\mathbf{k} \mathbin{¢} \mathbf{u})$

(iii) $\forall \mathbf{u} \in \mathcal{C}([\![\tau]\!]^{¢}_{\emptyset}), \mathbf{v} \in \mathcal{C}([\![[\tau]]\!]^{¢}_{\emptyset}).$
$$\begin{aligned} &appCost(\mathbf{fkz},\mathbf{v}) \rhd (\mathbf{fkz} \mathbin{¢} ((:^{¢\mathbf{k}}) \mathbin{¢} \mathbf{u} \mathbin{¢} \mathbf{v})) \\ &\quad = appCost(\mathbf{fkz}, (:^{¢\mathbf{k}}) \mathbin{¢} \mathbf{u} \mathbin{¢} \mathbf{v}) \rhd (\mathbf{k} \mathbin{¢} \mathbf{u} \mathbin{¢} (\mathbf{fkz} \mathbin{¢} \mathbf{v}))\end{aligned}$$

Statements *(i)* and *(ii)* are directly by the definition of $(:^{¢\mathbf{k}})$. For statement *(iii)* we calculate

$$\begin{aligned}
&appCost(\mathbf{fkz},\mathbf{v}) \rhd (\mathbf{fkz} \mathbin{¢} ((:^{¢\mathbf{k}}) \mathbin{¢} \mathbf{u} \mathbin{¢} \mathbf{v})) \\
= {}& appCost(\mathbf{fkz},\mathbf{v}) \rhd (\mathbf{fkz} \mathbin{¢} (val(\mathbf{u}) : val(\mathbf{v}), \\
&\qquad c_k + cost(\mathbf{u}) + cost(\mathbf{v_k}\ val(\mathbf{u})) + cost(\mathbf{v}) + \\
&\qquad\quad cost(val(\mathbf{v_k}\ val(\mathbf{u}))\ val(val(\mathbf{fkz})\ val(\mathbf{v}))))) \\
= {}& appCost(\mathbf{fkz},\mathbf{v}) \rhd (c_k + cost(\mathbf{v_k}\ val(\mathbf{u})) + \\
&\qquad\quad cost(val(\mathbf{v_k}\ val(\mathbf{u}))\ val(val(\mathbf{fkz})\ val(\mathbf{v})))) \rhd (\mathbf{fkz} \mathbin{¢} (\mathbf{u} :^{¢} \mathbf{v})) \\
= {}& \quad \{\ \mathbf{fkz} \mathbin{¢} (\mathbf{u} :^{¢} \mathbf{v}) = -c_k \rhd (\mathbf{k} \mathbin{¢} \mathbf{u}) \mathbin{¢} (\mathbf{fkz} \mathbin{¢} \mathbf{v})\ \} \\
& appCost(\mathbf{fkz},\mathbf{v}) \rhd (cost(\mathbf{v_k}\ val(\mathbf{u})) + \\
&\qquad\quad cost(val(\mathbf{v_k}\ val(\mathbf{u}))\ val(val(\mathbf{fkz})\ val(\mathbf{v})))) \rhd (\mathbf{k} \mathbin{¢} \mathbf{u} \mathbin{¢} (\mathbf{fkz} \mathbin{¢} \mathbf{v})) \\
= {}& \quad \left\{ \begin{array}{l} appCost(\mathbf{fkz}, (:^{¢\mathbf{k}}) \mathbin{¢} \mathbf{u} \mathbin{¢} \mathbf{v}) - appCost(\mathbf{fkz},\mathbf{v}) = \\ \qquad cost(\mathbf{v_k}\ val(\mathbf{u})) + cost(val(\mathbf{v_k}\ val(\mathbf{u}))\ val(val(\mathbf{fkz})\ val(\mathbf{v}))) \end{array} \right\} \\
& appCost(\mathbf{fkz}, (:^{¢\mathbf{k}}) \mathbin{¢} \mathbf{u} \mathbin{¢} \mathbf{v}) \rhd (\mathbf{k} \mathbin{¢} \mathbf{u} \mathbin{¢} (\mathbf{fkz} \mathbin{¢} \mathbf{v}))
\end{aligned}$$

□

Bibliography

Samson Abramsky and Achim Jung. *Domain Theory*, In *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994. ISBN 0-19-853762-X.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09), Proceedings*, pages 340–353, 2009. ACM Press. DOI 10.1145/1480881.1480925.

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11), Proceedings*, pages 201–214, 2011. ACM Press. DOI 10.1145/1926385.1926409.

Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23:657–683, September 2001. DOI 10.1145/504709.504712.

Lennart Augustsson. Putting Curry-Howard to work (Invited talk). At *Approaches and Applications of Inductive Programming*, 2009.

Edwin S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990. DOI 10.1016/0304-3975(90)90151-7. Special Issue Fourth Workshop on Mathematical Foundations of Programming Semantics, Boulder, CO, May 1988.

Henk Barendregt. *Lambda Calculi with Types*, In *Handbook of Logic in Computer Science (vol. 2)*, volume 2. Oxford University Press, 1992. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.4391.

Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In *7th International Conference on Typed Lambda Calculi and Applications (TLCA '05), Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, 2005. DOI 10.1007/11417170_8.

Jean-Philippe Bernardy. *A Theory of Parametric Polymorphism and an Application*. PhD thesis, Chalmers University of Technology and Göteborg University, 2011.

Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *19th European Symposium on Programming as part of European Joint Conferences on Theory and Practice of Software (ESOP/ETAPS '10), Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144. Springer-Verlag, 2010. DOI 10.1007/978-3-642-11957-6_8.

Sascha Böhme. Free theorems for sublanguages of Haskell. Diplomarbeit, Technische Universität Dresden, 2007.

Nina Bohr and Lars Birkedal. Relational reasoning for recursive types and references. In *4th Asian Symposium on Programming Languages and Systems (APLAS '06), Proceedings*, volume 4279 of *Lecture Notes in Computer Science*, pages 79–96. Springer-Verlag, 2006. DOI 10.1007/11924661_5.

Alan Bundy and Julian Richardson. Proofs about lists using ellipsis. In *6th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR' 99), Proceedings*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 1–12. Springer-Verlag, 1999. DOI 10.1007/3-540-48242-3_1.

Jan Christiansen and Daniel Seidel. Minimally strict polymorphic functions. In *13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '11), Proceedings*, pages 53–64, 2011. ACM Press. DOI 10.1145/2003476.2003487.

Alonso Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. DOI 10.2307/2266170.

Alonso Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941. ISBN 978-0-691-08394-0.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *15th ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Proceedings*, pages 268–279, 2000. ACM Press. DOI 10.1145/351240.351266.

Pierre Corbineau. First-order reasoning in the calculus of inductive constructions. In *Types for Proofs and Programs: 3rd Annual Workshop of the Types Working Group (TYPES '03), Revised and selected papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag, 2004. DOI 10.1007/978-3-540-24849-1_11.

Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

Peter J. de Bruin. Naturalness of polymorphism. Technical Report CS8916, Department of Mathematics and Computing Science, University of Groningen, 1989.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *24th Annual IEEE Symposium on Logic in Computer Science (LICS '09), Proceedings*, pages 71–80, August 2009. DOI 10.1109/LICS.2009.34.

Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10), Proceedings*, pages 143–156, 2010a. ACM Press. DOI 10.1145/1863543.1863566.

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful ADTs. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10), Proceedings*, pages 185–198, 2010b. ACM Press. DOI 10.1145/1706299.1706323.

Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22:477–528, 2012. DOI 10.1017/S095679681200024X.

R. Kent Dybvig. *The Scheme Programming Language, 3rd Edition*. MIT Press, 2003. ISBN 0-262-54148-3.

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992. DOI 10.2307/2275431.

João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *ACM SIGPLAN Workshop on Haskell (Haskell '07), Proceedings*, pages 95–106, 2007. ACM Press. DOI 10.1145/1291201.1291216.

Peter J. Freyd, Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Semantic parametricity in polymorphic lambda calculus. In *3rd Annual IEEE Symposium on Logic in Computer Science (LICS '88), Proceedings*, pages 274–279, jul 1988. DOI 10.1109/LICS.1988.5126.

The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.3*, April 2013. URL http://www.haskell.org/ghc/docs/7.6.3/users_guide.pdf.

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *International Conference on Functional Programming Languages and Computer Architecture (FPCA '93), Proceedings*, pages 223–232, 1993. ACM Press. DOI 10.1145/165180.165214.

Jean-Yves Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1997. ISBN 3-8272-9543-2.

Haskell, 2010. Haskell 2010 language report, 2010. URL http://haskell.org/definition/haskell2010.pdf.

Stefan Holdermans and Jurriaan Hage. Making "stricterness" more relevant. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '10), Proceedings*, pages 121–130, 2010. ACM Press. DOI 10.1145/1706356.1706379.

Furio Honsell and Donald Sannella. Prelogical relations. *Information and Computation*, 178(1):23–43, 2002. DOI 10.1006/inco.2002.3115.

William A. Howard. *The Formulae-as-Types Notion of Construction*, In *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 476–490. Academic Press, 1980. ISBN 0123490502.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III '07), Proceedings*, pages 12–1–12–55, 2007. ACM Press. DOI 10.1145/1238844.1238856.

Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(04):355–372, 1999. DOI 10.1017/S0956796899003500.

Patricia Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002. DOI 10.1023/A:1022982420888.

Patricia Johann. On proving the correctness of program transformations based on free theorems for higher-order polymorphic calculi. *Mathematical Structures in Computer Science*, 15(02):201–229, 2005. DOI 10.1017/S0960129504004578.

Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 04), Proceedings*, pages 99–110, 2004. ACM Press. DOI 10.1145/964001.964010.

Patricia Johann and Janis Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.

Patricia Johann and Janis Voigtländer. A family of syntactic logical relations for the semantics of Haskell-like languages. *Information and Computation*, 207 (2):341–368, 2009. DOI 10.1016/j.ic.2007.11.009. Special issue on Structural Operational Semantics (SOS).

Neil D. Jones and Flemming Nielson. *Handbook of Logic in Computer Science (vol. 4)*, chapter Abstract Interpretation: A Semantics-Based Tool for Program Analysis, pages 527–636. Oxford University Press, 1995. ISBN 0-19-853780-8.

Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973. ISBN 0-201-89685-0.

John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *6th European Symposium on Programming (ESOP '96), Proceedings*, volume 1058 of *Lecture Notes in Computer Science*, pages 204–218. Springer-Verlag, 1996. DOI 10.1007/3-540-61055-3_38.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system - release 3.12*. Institut National de Recherche en Informatique et en Automatique, 2010. URL http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf. Documentation and user's manual.

Yanhong A. Liu and Gustavo Gómez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12):1295–1309, 2001. DOI 10.1109/TC.2001.970569.

Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *17th European Symposium on Programming (ESOP '08), Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 16–31. Springer-Verlag, 2008. DOI 10.1007/978-3-540-78739-6_2.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960. DOI 10.1145/367177.367199.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997. ISBN 0262631814.

John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996. ISBN 0262133210.

Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In *14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09), Proceedings*, pages 135–148, 2009. ACM Press. DOI 10.1145/1631687.1596572.

Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer-Verlag, 1999. DOI 10.1007/3-540-48092-7_6.

Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: Disciplined advice with explicit effects. In *9th International Conference on Aspect-Oriented Software Development (AOSD '10), Proceedings*, pages 109–120, 2010. ACM Press. DOI 10.1145/1739230.1739244.

John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Simon L. Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell, version 1.3, May 1996. URL http://www.haskell.org/definition/haskell-report-1.3.ps.gz.

Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. DOI 10.2277/0521826144.

Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI '99), Proceedings, pages 25–36, 1999. ACM Press. DOI 10.1145/301618.301637.

Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *ACM SIGPLAN Workshop on Haskell (Haskell '01), Preliminary Proceedings*, Technical Report UU-CS-2001-23, Utrecht University, pages 203–233, 2001. URL http://haskell.org/haskell-workshop/2001/proceedings.pdf.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000. DOI 10.1017/S0960129500003066.

Andrew M. Pitts and Ian D.B. Stark. *Operational Reasoning for Functions with Local State*, In *Higher Order Operational Techniques in Semantics*, pages 227–274. Cambridge University Press, 1998. DOI 10.2277/0521631688.

Rinus Plasmeijer and Marko van Eekelen. *Clean Language Report Version 2.1*, 2002. URL http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf.

Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In *1st International Conference on Typed Lambda Calculi and Applications (TLCA '93), Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 1993. DOI 10.1007/BFb0037118.

Gordon D. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, Department of AI, University of Edinburgh, 1973.

Gordon D. Plotkin, John Power, Donald Sannella, and Robert Tennent. Lax logical relations. In *27th International Colloquium on Automata, Languages and Programming (ICALP '00), Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 85–102. Springer-Verlag, 2000. DOI 10.1007/3-540-45022-X_9.

Rawle Prince, Neil Ghani, and Conor McBride. Proving properties about lists using containers. In *9th International Symposium on Functional and Logic Programming (FLOPS '08), Proceedings*, volume 4989 of *Lecture Notes in Computer Science*, pages 97–112. Springer-Verlag, 2008. DOI 10.1007/978-3-540-78969-7_9.

Uday Reddy and Hongseok Yang. Correctness of data representations involving heap data structures. In *12th European Symposium on Programming (ESOP '03), Proceedings*, volume 2618 of *Lecture Notes in Computer Science*, pages 223–237. Springer-Verlag, 2003. DOI 10.1007/3-540-36575-3_16.

John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Proceedings*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, 9th IFIP World Computer Congress, Proceedings*, pages 513–523. Elsevier, 1983.

John C. Reynolds. Polymorphism is not set-theoretic. In *International Symposium of Semantics of Data Types, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer-Verlag, 1984. DOI 10.1007/3-540-13346-1_7.

Mads Rosendahl. Automatic complexity analysis. In *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89), Proceedings*, pages 144–156, 1989. ACM Press. DOI 10.1145/99370.99381.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Haskell Symposium (Haskell '08), Proceedings*, pages 37–48. ACM Press, 2008. DOI 10.1145/1411286.1411292.

David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995. DOI 10.1093/logcom/5.4.495.

Manfred Schmidt-Schauß, David Sabel, and Marko Schütz. Safety of Nöcker's strictness analysis. *Journal of Functional Programming*, 18(04):503–551, 2008. DOI 10.1017/S0956796807006624.

Daniel Seidel and Janis Voigtländer. Taming selective strictness. In *Arbeitstagung Programmiersprachen at 39th Jahrestagung der Gesellschaft für Informatik e.V., Proceedings*, volume 154 of *Lecture Notes in Informatics*, pages 2916–2930. GI, 2009a.

Daniel Seidel and Janis Voigtländer. Automatically generating counterexamples to naive free theorems. Technical Report TUD-FI09-05, Technische Universität Dresden, 2009b.

Daniel Seidel and Janis Voigtländer. Taming selective strictness. Technical Report TUD-FI09-06, Technische Universität Dresden, 2009c.

Daniel Seidel and Janis Voigtländer. Automatically generating counterexamples to naive free theorems. In *10th International Symposium on Functional and Logic Programming (FLOPS '10), Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 2010. DOI 10.1007/978-3-642-12251-4_14.

Daniel Seidel and Janis Voigtländer. Refined typing to localize the impact of forced strictness on free theorems. *Acta Informatica*, 48(3):191–211, 2011a. DOI 10.1007/s00236-011-0136-9.

Daniel Seidel and Janis Voigtländer. Improvements for free. In *9th Workshop on Quantitative Aspects of Programming Languages (QAPL '11), Proceedings*, volume 57 of *Electronic Proceedings in Theoretical Computer Science*, 2011b. DOI 10.4204/EPTCS.57.

Daniel Seidel and Janis Voigtländer. Proving properties about functions on lists involving element tests. In *20th International Workshop on Algebraic Development Techniques (WADT '10), Revised Selected Papers*, volume 7137 of *Lecture Notes in Computer Science*, pages 270–286. Springer-Verlag, 2012. DOI 10.1007/978-3-642-28412-0_17.

Florian Stenger and Janis Voigtländer. Parametricity for Haskell with imprecise error semantics. In *9th International Conference on Typed Lambda Calculi and Applications (TLCA '09), Proceedings*, volume 5608 of *Lecture Notes in Computer Science*, pages 294–308. Springer-Verlag, 2009. DOI 10.1007/978-3-642-02273-9_22.

Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000. DOI 10.1023/A:1010000313106. The article is a reprint of lecuture notes from the International Summer School in Computer Programming, Copenhagen, August 1967.

Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Proceedings*, pages 124–132. ACM Press, 2002. DOI 10.1145/583852.581491.

Kathryn van Stone. *A Denotational Approach to Measuring Complexity in Functional Programs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 2003.

Janis Voigtländer. Concatenate, reverse and map vanish for free. In *7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Proceedings*, pages 14–25, 2002. ACM Press. DOI 10.1145/581478.581481.

Janis Voigtländer. Proving correctness via free theorems: The case of the destroy/build-rule. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '08), Proceedings*, pages 13–20, 2008a. ACM Press. DOI 10.1145/1328408.1328412.

Janis Voigtländer. Much ado about two (pearl): a pearl on parallel prefix computation. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08), Proceedings*, pages 29–35, 2008b. ACM Press. DOI 10.1145/1328438.1328445.

Janis Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *9th International Symposium on Functional and Logic Programming (FLOPS '08), Proceedings*, volume 4989 of *Lecture Notes in Computer Science*, pages 163–179. Springer-Verlag, 2008c. DOI 10.1007/978-3-540-78969-7_13.

Janis Voigtländer. Asymptotic improvement of computations over free monads. In *9th International Conference on Mathematics of Program Construction (MPC '08), Proceedings*, volume 5133 of *Lecture Notes in Computer Science*, pages 388–403. Springer-Verlag, 2008d. DOI 10.1007/978-3-540-70594-9_20.

Janis Voigtländer. Bidirectionalization for free! In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09), Proceedings*, pages 165–176. ACM Press, 2009a. DOI 10.1145/1480881.1480904.

Janis Voigtländer. Free theorems involving type constructor classes: Functional pearl. In *14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09), Proceedings*, pages 173–184, 2009b. ACM Press. DOI 10.1145/1596550.1596577.

Janis Voigtländer and Patricia Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, 388(1–3):290–318, 2007. DOI 10.1016/j.tcs.2007.09.014.

Philip Wadler. Strictness analysis aids time analysis. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88), Proceedings*, pages 119–132, 1988. ACM Press. DOI 10.1145/73560.73571.

Philip Wadler. Theorems for free! In *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89), Proceedings*, pages 347–359, 1989. DOI 10.1145/99370.99404.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89), Proceedings*, pages 60–76, 1989. ACM Press. DOI 10.1145/75277.75283.

Index

Typeset October 17, 2013