

TECHNISCHE UNIVERSITÄT DRESDEN
Fakultät Informatik

DIPLOMARBEIT

Free Theorems for Sublanguages of Haskell

Sascha Böhme

29. Mai 2007

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Heiko Vogler
Lehrstuhl Grundlagen der Informatik
Institut für Theoretische Informatik
Betreuer: Dr. rer. nat. Janis Voigtländer

Abstract

Pure functional programs fulfill properties which can be derived solely from the types of their functions, especially from types of parametric polymorphic functions. These properties are called parametricity results, or more commonly, free theorems. First only considered for the polymorphic lambda calculus of Girard and Reynolds, research has studied how adding aspects of current functional programming languages like Haskell influence the expressiveness of such free theorems. These aspects cover undefined values, fixpoint combinators and selective strictness. The contribution of this thesis is to subsume these results in one common scheme and to enhance it with other aspects of Haskell, namely simple type classes and three kinds of user-defined data types. Additionally, several simplifications commonly used in deriving free theorems are identified. Based on these theoretic foundations, an implementation is described which allows to automatically generate free theorems.

Contents

1	Introduction	7
2	Haskell	10
2.1	User-defined data types	12
2.2	Type classes	15
3	Parametricity	17
3.1	Preliminaries	18
3.2	Type expressions	19
3.3	Type constructors and type classes	22
3.4	Semantics of closed type expressions	28
3.5	Relational actions	30
3.6	Logical relations	35
3.7	Parametricity theorems	39
4	Free theorems	41
4.1	The basic model	41
4.2	The <i>fix</i> models	43
4.3	The <i>seq</i> models	46
4.4	Type classes	50
4.5	Miscellaneous	51
5	Implementation	52
5.1	The library <i>free-theorems</i>	52
5.2	The user interface <i>ftshell</i>	55
5.3	Used applications and libraries	58
6	Conclusions	61
	Bibliography	64
	Index	67
	Symbols	70

List of Figures

2.1	Computing the factorial	10
2.2	Computing the factorial using <i>fix</i>	11
2.3	Computing the factorial using <i>seq</i>	12
2.4	Some Haskell functions	13
2.5	Some Haskell data types	14
2.6	Some Haskell type classes	16
5.1	The structure of the <i>free-theorems</i> library	53
5.2	An example demonstrating generics	55
5.3	The most important commands of <i>ftshell</i>	57
5.4	Package description of the <i>free-theorems</i> library	60

1 Introduction

According to [Str67], parametric polymorphic functions behave uniformly at every type. In functional programming languages based on the polymorphic lambda calculus of Girard and Reynolds [Gir72, Rey74], this concept is captured by *parametricity theorems* [Rey83, Wad89]. In [Wad89], it was then pointed out how these theorems may be used to derive properties of functions, especially of parametric polymorphic ones, solely from their types, that is, virtually for free. This is the reason why the results obtained from parametricity theorems are also called *free theorems*.

The key idea of [Rey83], which finally leads to free theorems, is to interpret types as relations instead of as sets. In this setting, type variables are interpreted as relations, basic types are usually interpreted as identity relations and, for every type constructor of arity n , there is a *relational action* which maps n given relations to a relational interpretation of the type constructor. In particular, there are relational actions for the type constructor of function types and for type abstractions, but in the same way, also relational actions for (selected) user-defined data type may be defined. Based on these relational actions, a *logical relation* [Plo80, Sta85] may be constructed, for which a parametricity theorem can then be proven. Note that the whole process is basically a rewriting of types into relational interpretations and finally into free theorems, which can also be automated.

Current functional programming languages like Haskell [Jon03] feature language constructs not available in the Girard-Reynolds calculus, notably undefined values, fixpoint combinators and selective strictness. Several works [JV06, LP96, Wad89] have studied the impact of these extensions on free theorems and found out suitable restrictions such that similar, but weaker free theorems can still be obtained.

This thesis subsumes these results and the aforementioned basic free theorems in one common scheme and applies them to the programming language Haskell 98 with the extension of higher-rank types. Three different sublanguages of Haskell are considered, one corresponding to the Girard-Reynolds calculus enriched with user-defined data types, a second sublanguage adding undefined values and fixpoint combinators to the first one, and finally a third sublanguage extending the second one with selective strictness. The notation used in this thesis is partly adopted from [Böh06, JV06].

As an example of free theorems, consider the Haskell function *length*, which has the following type.

$$\forall \alpha. [\alpha] \rightarrow Int$$

Without knowing the definition of *length* and just by considering the given type, the following can be deduced. For every Haskell type, this function maps every list of that type to an integer, which might be based on the structure of the argument list. For

1 Introduction

example, this function may consider only parts of the list's structure for constructing a result, or it may ignore the argument entirely and return a constant value for every list. Note, however, that the result is independent of the list elements, because the function *length*, due to its polymorphic type, cannot inspect the list elements, but must behave similarly for every type of list. That means, modifying every list element of a list before applying *length* must result in the same value as just applying *length* to the original list. Exactly this is expressed by the following free theorem, where *l* is an arbitrary list and *h* is a suitably typed, strict Haskell function.

$$\text{length } (\text{map } h \ l) = \text{length } l \quad (h \text{ strict})$$

Note that *map* is a higher order function which applies *h* to every element of a given list.

This equation may also be considered as a rewrite rule. Since the left-hand side is equivalent to the right-hand side, every occurrence of the term (*length (map h l)*) in a Haskell program may be replaced by (*length l*), which omits the additional modification of every list element. This suggests that free theorems may lead to program optimisations. In fact, one of their most prominent application is to prove correct one particular optimisation [GLPJ93]. Similarly, [Voi02] also applies free theorems in its proofs.

Introducing additional features like fixpoint combinators to the considered languages requires extra conditions for a parametricity theorem to hold. Some of these requirements partly vanish, when additionally considering only left-closed relations. Then, inequational results are obtained instead of equations as the one given earlier. For example, inequational free theorems for *length* are as follows.

$$\begin{aligned} \text{length } (\text{map } h \ l) &\sqsupseteq \text{length } l && (h \text{ strict}) \\ \text{length } (\text{map } h \ l) &\sqsubseteq \text{length } l \end{aligned}$$

The symbol \sqsubseteq denotes the semantic approximation partial order, which is introduced in Chapter 3. The intuition behind this partial order is to describe that one value is less defined than another one.

Note that only the first of the two inequational free theorems requires the function *h* to be strict, while, in contrast, the second theorem holds without restrictions on *h*. Similarly as before, these free theorems may be considered as rewrite rules. In particular, the second free theorem leads to improvements both in speed and in possibly avoiding errors, if every occurrence of (*length (map h l)*) in a Haskell program is replaced by (*length l*), because the latter is equally or more defined than the former and omits the application of (*map h*) to the list.

The idea of inequational free theorems is adopted from [JV06], where it is used to prove correct the optimisation strategy of [GLPJ93] in the presence of Haskell's selective strictness primitive *seq*.

Haskell also provides a restricted form of polymorphism by means of type classes [WB89]. In [Wad89], it is already sketched how they influence free theorems. In short, free theorems do not hold anymore for all types, but only for those which respect the additional operations described by type classes. Consider, for example, the following

type of the Haskell function *elem*.

$$\forall \alpha. Eq \alpha \Rightarrow \alpha \rightarrow [\alpha] \rightarrow Bool$$

This type describes that the function *elem* is only defined for types which respect the type class *Eq*, and this condition is directly reflected to the free theorem of *elem*. That means, if τ and τ' are types respecting *Eq*, then for every x of type τ , for every list xs of type $[\tau]$ and for every strict Haskell function $f :: \tau \rightarrow \tau'$ respecting *Eq*, the following theorem holds.

$$elem\ x\ xs = elem\ (f\ x)\ (map\ f\ xs) \quad (f\ \text{strict and respects } Eq)$$

Note that also inequational free theorems may be derived in the presence of type classes.

Respecting *Eq* means that f has to fulfill the parametricity properties of all operations of *Eq*, that is, the function f must maintain equality and inequality. More formally, for every x_1 and x_2 of type τ , the following condition must hold.

$$(x_1 == x_2 \Leftrightarrow f\ x_1 == f\ x_2) \wedge (x_1 /= x_2 \Leftrightarrow f\ x_1 /= f\ x_2)$$

The underlying idea of this example can be generalised to other type classes of Haskell as well, and this thesis uses the hints given in [Wad89] for incorporating simple type classes into the aforementioned scheme.

This thesis is structured as follows. Chapter 2 introduces language features of Haskell on which the other parts of this thesis are relying. Instead of giving syntax definitions, this chapter describes the language by means of examples. Thereafter, Chapter 3 describes the theoretic foundations of free theorems. It defines type expressions and restricts Haskell types and type classes to permissible ones. Based on a (naive) semantics of Haskell types, relational actions and logical relations are defined, which then lead to parametricity theorems. The application of these theorems to derive free theorems is described in Chapter 4, along with several simplifications. Chapter 5 covers an implementation of these algorithms, which allows to automatically generate free theorems from Haskell types. The implementation is compared with existing solutions, and additionally, the used tools are shortly described. Finally, Chapter 6 concludes this thesis by pointing out the contributions and the results as well as further studies and possible improvements.

2 Haskell

Guided by examples, this chapter is intended to give a concise introduction to various features of the functional programming language Haskell 98, including one language extension. For a complete description of this language, however, consult [HPF99] or the Haskell report [Jon03].

The basic components of Haskell are functions. Their notation is close to that of mathematical functions as can be seen from the example shown in Figure 2.1. Another similarity between mathematics and Haskell is that functions are pure, that means, functions always return the same result when applied to the same values.

Being based on the lambda calculus, Haskell's evaluation model is to simplify *terms*. Terms in Haskell are, for example, $(n - 1)$ and also the whole right-hand side of *fac* in Figure 2.1. If a term does not contain free variables, it is called *closed*. Examples for closed terms are $(1 + 2)$, which is trivially closed, and the lambda term $(\lambda x \rightarrow 2 * x)$. From the perspective of terms, functions can be considered as names for closed terms.

Every term, and thus every function, can be assigned a most general type in Haskell. Due to Haskell's roots in the Hindley-Milner type system [Hin69, Mil78], the most general type can nearly always be inferred automatically. As a consequence, Haskell terms and functions mostly do not need explicit type annotations.

Strongly included in Haskell's type system is the concept of parametric polymorphism [Str67], which allows to define functions behaving uniformly for every type. For example, the function *id* shown in Figure 2.4 on page 13 is defined for every type and returns its argument unmodified. Note that the universal quantification of type variables is given explicitly for all functions in this and subsequent chapters, although the syntax of Haskell 98 does not know of such a concept.

Haskell allows to define higher order functions, that is functions which have functions as arguments. An example is *map*, whose definition is given in Figure 2.4 on page 13. This function applies a function to every element of a list without changing the structure of that list.

$$\begin{aligned} \textit{fac} &:: \textit{Int} \rightarrow \textit{Int} \\ \textit{fac} \ n &= \mathbf{case} \ n \ \mathbf{of} \\ &\quad 0 \quad \quad \quad \rightarrow 1 \\ &\quad \textit{otherwise} \rightarrow n * \textit{fac} \ (n - 1) \end{aligned}$$

Figure 2.1: Definition of a Haskell function to compute the factorial.

$$\begin{aligned}
fac' &:: Int \rightarrow Int \\
fac' &= fix\ h \\
\text{where } h\ f\ n &= \text{case } n \text{ of} \\
&\quad 0 \quad \quad \rightarrow 1 \\
&\quad otherwise \rightarrow n * f\ (n - 1)
\end{aligned}$$

Figure 2.2: Definition of a function to compute the factorial using the fixpoint combinator *fix*.

An extension to Haskell's type system represents the concept of higher-rank types, which allows to quantify the type variables of higher order functions. In [GLPJ93] for example, the higher-rank type $(\forall\alpha. (\forall\beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha])$ is discussed. Other prominent occurrences of higher-rank types can be found in [LJ03].

One of the means to express algorithms in Haskell is by recursion. The factorial function shown in Figure 2.1 on the preceding page, for example, is defined recursively. In Haskell, it is possible to describe the concept of recursion itself in an elegant way, using higher order functions, which is demonstrated by the definition of the fixpoint combinator *fix* given in Figure 2.4 on page 13. With the help of *fix*, the factorial function can be rewritten into two parts, a non-recursive function describing one computation step and a function applying *fix* to that one-step computation (see Figure 2.2).

Recursively defined functions or functions making use of the fixpoint combinator *fix* do not always terminate. An example is the following non-terminating function *nonterm*.

$$\begin{aligned}
nonterm &:: \forall\alpha.\alpha \\
nonterm &= fix\ id
\end{aligned}$$

In denotational semantics [Sch86], such non-terminating computations are described by the symbol \perp . Additionally, this symbol represents any undefined value, which also covers, for example, errors caused by division by zero.

Haskell is a non-strict language, which means arguments are not evaluated when applied to a function, they are only evaluated when they are needed. This has an impact especially for undefined values or non-terminating terms passed as arguments. For example, consider the function *const* shown in Figure 2.4 on page 13. The term $(const\ 1\ nonterm)$ can be evaluated to 1, because *const* ignores its second argument.

In certain situations, however, in particular to increase performance, strictness is required. Therefore, Haskell offers a way to force evaluation by means of the strictness primitive *seq*. The Haskell report gives its definition in pseudo-code as follows.

$$\begin{aligned}
seq &:: \forall\alpha\beta.\alpha \rightarrow \beta \rightarrow \beta \\
seq\ \perp\ b &= \perp \\
seq\ a\ b &= b \quad (\text{if } a \neq \perp)
\end{aligned}$$

```

fac'' :: Int → Int
fac'' n = h n 1
  where h n a = case n of
            0      → a
            otherwise → let a' = n * a
                          in seq a' (h (n - 1) a')

```

Figure 2.3: Definition of a function to compute the factorial using the strictness primitive *seq*. The purpose of *seq* is to increase the performance of *fac''* by forced evaluation of products.

The primitive *seq* tries to evaluate its first argument to weak head normal form and returns its second argument afterwards. Note that *seq* can be used to distinguish \perp from $(\backslash x \rightarrow \perp)$, although both are equal in that they map every argument to an undefined value. An example applying *seq* is shown in Figure 2.3, which is yet another version of the factorial function.

2.1 User-defined data types

Beside the built-in primitive types *Char*, *Int*, *Integer*, *Float* and *Double*, Haskell's rich type system allows to combine existing types to function types, list types and tuple types. Furthermore, the type system can be enriched by three kinds of user-defined types, namely *algebraic data types*, *type synonyms* and *type renamings*, which are introduced in this section. A small selection of user-defined data types available by default in Haskell can be seen in Figure 2.5 on page 14.

Instead of repeating the definitions of the Haskell report, several examples are chosen to informally describe the syntax of algebraic data type declarations. The first example shows the declaration of an algebraic data type *Tree*, which can store values of an arbitrary, but fixed type in its nodes.

```

data Tree α = Leaf | Node α (Tree α) (Tree α)

```

The declared type name *Tree* is called a *type constructor*, while both *Leaf* and *Node* are referred to as *data constructors*.

Haskell's list type also fits in the scheme of algebraic data types. Since it uses a special syntax, only a pseudo-declaration, which is not valid Haskell code, can be given as follows.

```

data [α] = [] | α : [α]

```

Note that `[]` is both the list type constructor accepting one type argument and the data constructor for empty lists. Note further, that binary data constructors like the colon

```

id :: ∀α. α → α
id x = x

const :: ∀α β. α → β → α
const x y = x

(o) :: ∀α β γ. (β → γ) → (α → β) → α → γ
f o g = λx → f (g x)

fix :: ∀α. (α → α) → α
fix f = f (fix f)

map :: ∀α β. (α → β) → [α] → [β]
map f [] = []
map f (x : xs) = f x : map f xs

length :: ∀α. [α] → Int
length [] = 0
length (x : xs) = 1 + length xs

filter :: ∀α. (α → Bool) → [α] → [α]
filter p [] = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs

elem :: ∀α. Eq α ⇒ α → [α] → Bool
elem x [] = False
elem x (y : ys) = (x == y) || (x `elem` ys)

```

Figure 2.4: Some Haskell functions. Note that the function composition operator (*o*) is written as a dot in Haskell.

```

data Bool = False | True
data Maybe  $\alpha$  = Nothing | Just  $\alpha$ 
data Either  $\alpha$   $\beta$  = Left  $\alpha$  | Right  $\beta$ 
type String = [Char]

```

Figure 2.5: Some Haskell data types

which consist solely of non-alphanumeric symbols can be written in infix notation in Haskell.

Functions over algebraic data types are usually defined by distinction of the cases occurring in the declaration of those types. This is also called *pattern matching* and can be seen, for example, on the functions *map* or *length* in Figure 2.4 on the previous page. There, similarly as in the pseudo-declaration of the list type, empty lists and lists having at least one element are handled in different cases.

Recursively declared algebraic data types allow for infinite values. The following function *ones*, for example, generates an infinite list whose elements are all equal to the number 1.

```

ones :: [Int]
ones = 1 : ones

```

Note that infinite lists, and infinite values in general, are closely connected to recursion in Haskell, that is, recursion is the necessary means to construct or traverse infinite values.

Haskell allows algebraic data types to be declared using *strictness flags*, denoted by an exclamation mark. The purpose of this is to improve performance by forced evaluation of the flagged data constructor argument. For example, rational numbers are defined using strictness flags.

```

data Ratio  $\alpha$  = ! $\alpha$  :% ! $\alpha$ 

```

Note that `:%` is the single data constructor of *Ratio*. Note further that this declaration is a simplified version of the declaration given by the Haskell report, because class constraints (see Section 2.2) for α have been omitted here.

Strictness flags only play a role in constructing values of algebraic data types which have such flags. Especially, they do not affect pattern matching. As an example, consider the negation of rational numbers, which may be written in Haskell as follows.

```

negate (x :% y) = (-x) :% y

```

Since rational numbers are declared to have a strictness flag both at the numerator and at the denominator, the *negate* function is automatically expanded to the following

function.

$$\text{negate } (x \% y) = \text{let } x' = -x \text{ in seq } y \text{ (seq } x' \text{ (} x' \% y \text{))}$$

That means, the result of negating a rational number in Haskell is always evaluated immediately. Without strictness flags, the negation would be deferred until the result is needed.

Opposed to algebraic data types, type synonyms do not declare new types, but give new names to existing types. For example, the type *String* is another name for a list of characters (see Figure 2.5 on the facing page). The following declaration of a type synonym gives yet another example.

$$\text{type } \textit{ErrorOr } \alpha = \textit{Either } \textit{String } \alpha$$

The type synonym *ErrorOr* encapsulates a possible error message and introduces an intuitive name. It could be applied in the type signature of a parser, as for example in $(\textit{String} \rightarrow \textit{ErrorOr } \textit{AbstractSyntax})$, where *AbstractSyntax* would be the parsing result. Note that this type signature is the same as $(\textit{String} \rightarrow \textit{Either } \textit{String } \textit{AbstractSyntax})$, although the former is more succinct and descriptive in this context.

Finally, the last kind of user-defined data types are type renamings. They may be considered as simplified algebraic data types, in that they have only one data constructor with exactly one argument. There is, however, an important semantic difference between type renamings and algebraic data types. To illustrate that, consider the following declarations and functions taken from [Jon03].

$$\begin{array}{ll} \text{newtype } N = N \textit{ Int} & n (N \ i) = 0 \\ \text{data } D = D \textit{ Int} & d (D \ i) = 0 \end{array}$$

The term $(d \ \perp)$ is equivalent to \perp , while $(n \ \perp)$ is equivalent to 0. The reason for this is as follows. Performing pattern matching of \perp against $(D \ i)$ fails, because $(D \ i)$ is always distinct to \perp , even if i equals \perp . This is why algebraic data types are called *lifted*. Opposed to that, type renamings are *unlifted*, that is, $(N \ \perp)$ equals \perp . Put in an informal way, the type renaming *N* only tags the type *Int* with a data constructor, but behaves operationally as an integer. Therefore, pattern matching of \perp against $(N \ i)$ is always successful, and thus, the term $(n \ \perp)$ can be evaluated to 0.

2.2 Type classes

There exist problems, where parametric polymorphism is not applicable, but nevertheless, a restricted kind of polymorphism is desirable. Consider, for example, the multiplication, which may be defined for natural and rational numbers, but not for strings. Opposed to parametric polymorphism, however, multiplication cannot be defined uniformly for every type. There are differences between multiplying two natural numbers and multiplying two rational numbers, for example.

```

class Eq  $\alpha$  where
  (==) ::  $\alpha \rightarrow \alpha \rightarrow Bool$ 
  (/=) ::  $\alpha \rightarrow \alpha \rightarrow Bool$ 

class Eq  $\alpha \Rightarrow Ord$   $\alpha$  where
  compare ::  $\alpha \rightarrow \alpha \rightarrow Ordering$ 
  (<), (<=), (>=), (>) ::  $\alpha \rightarrow \alpha \rightarrow Bool$ 
  max, min ::  $\alpha \rightarrow \alpha \rightarrow \alpha$ 

```

Figure 2.6: Some Haskell type classes

The concept of such a restricted polymorphism is known as ad-hoc polymorphism [Str67]. In Haskell, it is provided by *type classes* and *instances* [WB89]. Type classes specify a set of signatures, that is, function names, called *class methods*, equipped with associated types. An instance of a type class then gives an implementation of these class methods at a fixed type. See Figure 2.6 for two type class declarations in Haskell. Instances of the type class *Eq*, for example, are the primitive type *Int* and the list type $[\alpha]$, if α represents an instance of *Eq* itself.

To allow for a fine-grained classification, type classes may be arranged in a type class hierarchy, that is, every type class may have one or more superclasses, and this type class then inherits the class methods of its superclasses. For example, the type class *Eq* in Figure 2.6 is the superclass of *Ord*. The latter extends the former by comparison operations, while the equality and inequality operations are inherited from the former. Thus, instances of *Ord* can also be compared for equality, but it is mandatory for every instance of *Ord* to be an instance of *Eq*, too, that is, instances of *Ord* also have to implement the class methods of *Eq*.

Polymorphic functions which are defined for instances of certain type classes only may use the corresponding class methods in their definition. The function *elem* shown in Figure 2.4 on page 13, for example, is defined for all types instantiating the type class *Eq*. Therefore, the equality operation can be applied in the definition of that function. The explicitly given restrictions for type variables, for example $(Eq\ \alpha)$ in the type of *elem*, are called *class constraints*.

Note that Haskell's system of type classes also offers type constructor classes [Jon93], although they are not covered in subsequent chapters. For example, the following type class *Functor* is a type constructor class.

```

class Functor  $\alpha$  where
  fmap ::  $\forall \beta \gamma. (\beta \rightarrow \gamma) \rightarrow \alpha\ \beta \rightarrow \alpha\ \gamma$ 

```

3 Parametricity

Starting with type expressions and declarations, continuing with relational actions and finishing with parametricity theorems, this chapter covers the theoretical foundations for deriving parametricity results.

It has been stated earlier already, that these parametricity results are derived for three different sublanguages of Haskell. The basic sublanguage corresponds to the polymorphic lambda calculus of Girard and Reynolds [Gir72, Rey74], enriched with data types and type classes. This sublanguage does not allow undefined values, which are sometimes referred to as errors, to occur in any term. Undefined values are caused, for example, by non-termination, failed pattern matching or division by zero. Fixed point combinators and the selective strictness primitive *seq* are not available. Note that arguments to functions are never undefined. Also note, that, due to the lack of *seq*, algebraic data types with strictness flags are not part of the basic sublanguage. The second sublanguage allows for fixpoint combinators and undefined values, but it does neither provide the selective strictness primitive *seq* nor algebraic data types with strictness flags. Finally, the third sublanguage enriches the former two by adding the selective strictness primitive *seq* and by permitting algebraic data types with strictness flags. Note that this sublanguage also allows for errors and undefined values as well as fixpoint combinators.

In addition to equational parametricity results, the second and the third sublanguage also allow to derive inequational results. To better distinguish these possibilities, the following set M of five *models* is used, whose elements will be described afterwards.

$$M = \{(basic, =), (fix, =), (fix, \sqsubseteq), (seq, =), (seq, \sqsubseteq)\}$$

The $(basic, =)$ model corresponds to the basic sublanguage. The two models $(fix, =)$ and (fix, \sqsubseteq) belong to the second sublanguage, where derived parametricity results are equational in the first model and inequational in the second model. Similarly, the two models $(seq, =)$ and (seq, \sqsubseteq) correspond to the third sublanguage, again the first for equational results and the second for inequational ones.

The contents of this chapter are oriented towards describing properties of Haskell. To maintain a close connection to this language, types of Haskell functions are given in the Haskell style, and function application of Haskell functions uses the curried form. In contrast to that, mathematical functions occurring in definitions follow the mathematical notation. Throughout this chapter, assume a Haskell program H which conforms to the Haskell 98 report, but which may additionally use higher-rank types.

This chapter is organised as follows. First, Section 3.1 introduces basic notations, some of which are usual concepts of denotational semantics [Sch86]. The chapter continues by defining the structure of type expressions and functions over them in Section 3.2. Since

3 Parametricity

certain restrictions are necessary for defining relational actions of type constructors in a general way, Section 3.3 specifies special restricted sets of type constructors. For similar reasons, restrictions on type classes are formulated there, too. Section 3.4 then shortly introduces semantics of type expressions, guided by several examples, before relational actions are defined in Section 3.5. Thereafter, logical relations are defined in Section 3.6, and finally, based on these logical relations, Section 3.7 formulates parametricity theorems, one for each model.

3.1 Preliminaries

The set of *natural numbers* without zero is \mathbb{N} , while the set of natural numbers including zero is denoted by \mathbb{N}_0 .

When writing down logical formulas, conjunctions are symbolised by \wedge , implications are denoted as \Rightarrow , and logical equivalence is represented by \Leftrightarrow . To reduce the need for parentheses, universal quantifications, using the symbol \forall , extend as far as possible to the right.

Given a set X , the set $\mathcal{P}(X)$ is the *power set* of X , that is, the set of all subsets of X .

A *ranked set* is a pair (A, rank) where A is a finite set and $\text{rank} : A \rightarrow \mathbb{N}_0$. For every $n \in \mathbb{N}_0$, the set $\{a \in A \mid \text{rank}(a) = n\}$ is denoted by $A^{(n)}$. To simplify notation, every ranked set (A, rank) is written as A from now on. Ranked sets are considered as sets where it is convenient.

Let X be a set and \sqsubseteq_X be a partial order over X . A function $\bar{x} : \mathbb{N}_0 \rightarrow X$ is called a *monotone sequence* over X , if $\bar{x}(i) \sqsubseteq_X \bar{x}(i+1)$ for every $i \in \mathbb{N}_0$. If \sqsubseteq_X has a least element, denoted by \perp_X , and every monotone sequence \bar{x} over X has a supremum, written as $\bigsqcup \bar{x}$, then \sqsubseteq_X is a *complete partial order* and X is called *ordered*. In what follows, the subscripts will often be omitted.

Let $R \in X \times Y$ be a binary relation over two ordered sets X and Y . Then, R is called *strict* if $(\perp, \perp) \in R$. Furthermore, R is *total* if, for every pair (x, y) contained in it, $y = \perp$ implies $x = \perp$. The relation R is called *bottom-reflecting* if, for every pair (x, y) contained in it, $y = \perp$ iff $x = \perp$. If, for every two monotone sequences \bar{x} and \bar{y} over X and Y , respectively, satisfying $(\bar{x}(i), \bar{y}(i)) \in R$ for every $i \in \mathbb{N}_0$, the pair of suprema $(\bigsqcup \bar{x}, \bigsqcup \bar{y})$ is in R , then R is *continuous*. A relation is *admissible* if it is strict and continuous.

Given three sets X , Y and Z , and two binary relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$, the composition of R and S , denoted by $R;S$, is defined as

$$R;S = \{(x, z) \mid \exists y \in Y. (x, y) \in R \wedge (y, z) \in S\} \subseteq X \times Z$$

If X and Y are sets and X is additionally ordered, then a relation $R \subseteq X \times Y$ is *left-closed* if $\sqsubseteq_X;R = R$.

The inverse of a relation $R \subseteq X \times Y$ is defined as $R^{-1} = \{(y, x) \mid (x, y) \in R\} \subseteq Y \times X$. The relation \sqsubseteq^{-1} is also denoted by \sqsupseteq .

Functions are special relations, because a function $f : X \rightarrow Y$ may be interpreted as the set $\{(x, y) \mid f(x) = y\} \subseteq X \times Y$, called the graph of f . Let X and Y be ordered sets.

The function f is said to be *monotonic* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for every $x, y \in X$.

Let V be a finite set and let E be a binary relation over V . The tuple $G = (V, E)$ is called a *directed graph*. The elements of V are referred to as vertices. The transitive closure of E is denoted by E^+ . Let $V_c = \{v \in V \mid (v, v) \in E^+\}$ be the set of all vertices in V which are part of a cycle. For any two subsets $V_1, V_2 \subseteq V$, the set $R(V_1, V_2) = \{v_2 \in V_2 \mid \exists v_1 \in V_1. (v_1, v_2) \in E^+\}$ specifies all vertices in V_2 which are reachable from a vertex of V_1 . If V' is a subset of V , then the set $V' \setminus (R(V \setminus V', V') \cup R(V_c, V'))$ is called the *restricted, acyclic subset* of V' .

3.2 Type expressions

Here and subsequently, the following three sets are fixed. Let V be a countable infinite set whose elements are called *type variables*. They are usually denoted by α, β or γ , and sometimes additionally written with an index. Furthermore, let C be the ranked set of *type constructors* occurring in the assumed Haskell program H . The elements of C are mostly denoted by σ . Finally, let D denote the set of all *type classes* occurring in H . The elements of D are denoted by δ , and subsets of D are sometimes written as Δ . Note that the three sets V, C and D are pairwise disjoint. See Section 3.3 for a more thorough discussion of type constructors and type classes.

Definition 1. The set of *type expressions* $T(\hat{C}, \hat{D})$ over an arbitrary ranked set $\hat{C} \subseteq C$ and over an arbitrary set $\hat{D} \subseteq D$ is the smallest set X which fulfills the following requirements.

- $V \subseteq X$
- For every $n \in \mathbb{N}_0$, for every $\sigma \in \hat{C}^{(n)}$ and for every $\tau_1, \dots, \tau_n \in X$, it holds that $(\sigma \tau_1 \dots \tau_n) \in X$.
- For every $\tau_1, \tau_2 \in X$, it holds that $(\tau_1 \rightarrow \tau_2) \in X$.
- For every $\Delta \subseteq \hat{D}$, for every $\alpha \in V$ and for every $\tau \in X$, it holds that $(\forall_{\Delta} \alpha. \tau) \in X$.

For every $\tau_1, \tau_2 \in T(\hat{C}, \hat{D})$, the type expression $(\tau_1 \rightarrow \tau_2)$ is called a *function type*. The symbol \rightarrow is referred to as *function type constructor*, and, despite its name, it is required to not be a member of C . For every $\Delta \subseteq \hat{D}$, for every $\alpha \in V$ and for every $\tau \in T(\hat{C}, \hat{D})$, the type expression $(\forall_{\Delta} \alpha. \tau)$ is called *type abstraction*. \square

Example 1. Let $\alpha \in V$ be a type variable. Then, the type expression $(\alpha \rightarrow \alpha)$ is in the set $T(\emptyset, \emptyset)$. If the nullary type constructor Int is in $C^{(0)}$ and the unary type constructor $[]$ is in $C^{(1)}$, then the type expression $(\forall_{\emptyset} \alpha. (([] \alpha) \rightarrow (Int)))$ is in $T(C, \emptyset)$. If the type class Eq is in D and the nullary type constructor $Bool$ is in $C^{(0)}$, then the type expression $(\forall_{\{Eq\}} \alpha. (\alpha \rightarrow (\alpha \rightarrow (Bool))))$ is an element of $T(C, D)$. \square

As an extension to Definition 1, the special notation of Haskell's list type constructor $[]$ is adopted. More precisely, if $([] \tau) \in T(C, D)$, then $([\tau])$ is also in $T(C, D)$. The

3 Parametricity

parentheses around the latter type expression are usually dropped. Additionally, the inner type expression τ may omit outermost parentheses. For example, if $\alpha \in V$ is a type variable, then $([] (\alpha \rightarrow \alpha))$ can also be written as $[\alpha \rightarrow \alpha]$. Similar to the list type constructor, Haskell’s special notation for the pair type constructor $(,)$ is also allowed, and the same rules concerning parentheses apply. As an example, if $Int, Float \in C^{(0)}$ are nullary type constructors, the type expression $((,) (Int) (Float)) \in T(C, D)$ can also be represented by $(Int, Float)$. It is straight-forward to generalise this notation for tuple type constructors of arbitrary rank.

More rules to simplify the notation of type expressions are as follows. The function type constructor is right-associative. Thus, for example, the type expression $\alpha \rightarrow \alpha \rightarrow \alpha$ equals $(\alpha \rightarrow (\alpha \rightarrow \alpha))$ for a type variable $\alpha \in V$. Empty sets of type classes can be dropped. That means, the type expression $(\forall \alpha. \alpha)$ equals $(\forall_{\emptyset} \alpha. \alpha)$ for every type variable $\alpha \in V$. Quantifications extend to the right as far as possible. As an example, the type expression $(\forall \alpha. \alpha \rightarrow \alpha)$ is the same as $(\forall \alpha. (\alpha \rightarrow \alpha))$. Finally, the outermost parentheses of type expressions occurring as arguments to the function type constructor can be dropped, except if these type expressions are function types or type abstractions.

Example 2. Let $\alpha, \beta, \gamma \in V$ be type variables. Assume that $Int \in C^{(0)}$, $Maybe \in C^{(1)}$ and $Either \in C^{(2)}$. Then consider the following type expression, which has all parentheses as required by Definition 1.

$$((\forall \alpha. (Maybe \alpha)) \rightarrow ((Either ((\alpha \rightarrow \beta)) (Maybe \gamma)) \rightarrow (((Int), \beta))))$$

The result of omitting all optional parentheses is then as follows.

$$(\forall \alpha. Maybe \alpha) \rightarrow Either [\alpha \rightarrow \beta] (Maybe \gamma) \rightarrow (Int, \beta) \quad \square$$

According to Definition 1, type expressions are written in a slightly different way than in Haskell (compare with Section 2.2). For example, the Haskell type expression $(\forall \alpha \beta. (Eq \alpha, Ord \beta) \Rightarrow \alpha \rightarrow \beta \rightarrow Bool)$ is written as $(\forall_{\{Eq\}} \alpha. \forall_{\{Ord\}} \beta. \alpha \rightarrow \beta \rightarrow Bool)$. Note that the first type expression can also be given as $(\forall \alpha. Eq \alpha \Rightarrow (\forall \beta. Ord \beta \Rightarrow \alpha \rightarrow \beta \rightarrow Bool))$ in Haskell, which resembles the second type expression. In fact, when ignoring type constructor classes [Jon93], it is always possible to reorder the class constraints in a Haskell type expression, such that the constraints for a type variable stand next to that type variable’s quantification. Therefore, there is a bijection between Haskell type expressions and type expressions conforming to Definition 1.

It is worth noting that type expressions introduced so far resemble in their structure untyped lambda terms [Bar84]. More precisely, type abstraction is comparable to lambda abstraction, while the function type constructor and the other type constructors are similar to constants in the lambda calculus. This analogy allows to adopt various concepts from the lambda calculus, notably free and bound type variables, renaming of bound type variables and replacement of type variables with type expressions, which will all be introduced in detail in the remainder of this section.

Definition 2. The function $f_v : T(C, D) \rightarrow \mathcal{P}(V)$ maps a type expression to the set of *free type variables* of that type expression and is defined by structural induction as follows.

- For every $\alpha \in V$, $f_v(\alpha) = \{\alpha\}$.
- For every $n \in \mathbb{N}_0$ and for every $\sigma \in C^{(n)}$, if $\tau_1, \dots, \tau_n \in T(C, D)$, then

$$f_v(\sigma \tau_1 \dots \tau_n) = \bigcup \{f_v(\tau_i) \mid i \in \mathbb{N}, 1 \leq i \leq n\}$$

- If $\tau_1, \tau_2 \in T(C, D)$, then $f_v(\tau_1 \rightarrow \tau_2) = f_v(\tau_1) \cup f_v(\tau_2)$.
- For every $\Delta \subseteq D$ and for every $\alpha \in V$, if $\tau \in T(C, D)$, then

$$f_v(\forall_{\Delta} \alpha. \tau) = f_v(\tau) \setminus \{\alpha\} \quad \square$$

To illustrate this definition, let $\alpha, \beta, \gamma \in V$ be type variables and consider the following type expression.

$$\alpha \rightarrow (\forall_{\{Eq\}} \beta. \forall \gamma. \beta \rightarrow \text{Maybe } \gamma) \rightarrow (\alpha, \beta)$$

The free type variables of this type expression are α and β . Note that, although β is quantified in a type abstraction, it occurs free in the pair on the right-hand side.

A type variable is called *bound* in a type expression, if it is quantified in a type abstraction. For instance, in the previous example, the type variables β and γ are bound while α is not. The set of bound type variables of a type expression can be defined in much the same way as the set of free type variables. Thus, a definition is omitted here.

Closely connected to free and bound type variables is the notion of *closed type expressions*. A type expression is closed, if it does not contain free variables. For example, (*Either Int Bool*) and $(\forall \alpha. \alpha \rightarrow \alpha)$ are closed where $\alpha \in V$ is a type variable. The set of all closed type expressions over the ranked set $\hat{C} \subseteq C$ of type constructors and over the set $\hat{D} \subseteq D$ of type classes is denoted as $T_c(\hat{C}, \hat{D})$.

In lambda calculus, α -conversion specifies the renaming of bound variables. Due to the close relationship mentioned earlier, this concept carries over to the renaming of bound type variables in type expressions. Instead of giving a formal definition here, an example shall highlight the intuitive idea. Consider the type variables $\alpha, \beta, \gamma \in V$ and the type expression $(\alpha \rightarrow (\forall \alpha. \text{Maybe } \alpha \rightarrow (\forall \beta. \alpha)))$. Renaming every bound occurrence of α to γ then yields the type expression $(\alpha \rightarrow (\forall \gamma. \text{Maybe } \gamma \rightarrow (\forall \beta. \gamma)))$. Note that the free occurrence of α is not renamed. Note furthermore that the renaming of α to β is not defined because β occurs already bound in the type expression.

The next definition also originates from the lambda calculus. In short, it defines how free type variables can be replaced by type expressions. This is formally defined as follows.

Definition 3. Let $n \in \mathbb{N}_0$ and let $\tau \in T(C, D)$ be a type expression. Let $\alpha_1, \dots, \alpha_n \in V$ be n pairwise distinct type variables. Furthermore, let $\tau_1, \dots, \tau_n \in T(C, D)$ be type expressions such that, for every $i \in \mathbb{N}$ with $1 \leq i \leq n$, the set of bound type variables of τ and the set of free type variables of τ_i are disjoint. The result of replacing every free occurrence of α_i in τ with τ_i for every $i \in \{1, \dots, n\}$ is then denoted by $\tau[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$. To shorten notation in this definition, it is abbreviated to $[\tau, \eta]$, where $\eta : V \rightarrow T(C, D)$ is a

3 Parametricity

mapping of type variables to type expressions such that $\eta(\alpha_i) = \tau_i$ for every $i \in \{1, \dots, n\}$ and $\eta(\alpha) = \alpha$ for every other type variable $\alpha \in V \setminus \{\alpha_1, \dots, \alpha_n\}$. The type expression $[\tau, \eta]$ is defined as follows by induction over the structure of τ .

- For every $\alpha \in V$:

$$[\alpha, \eta] = \eta(\alpha)$$

- For every $k \in \mathbb{N}_0$, for every $\sigma \in C^{(k)}$, for every $\tau_1, \dots, \tau_k \in T(C, D)$:

$$[\sigma \tau_1 \dots \tau_k, \eta] = \sigma [\tau_1, \eta] \dots [\tau_k, \eta]$$

- For every $\tau_1, \tau_2 \in T(C, D)$:

$$[\tau_1 \rightarrow \tau_2, \eta] = [\tau_1, \eta] \rightarrow [\tau_2, \eta]$$

- For $\Delta \subseteq D$, for every $\alpha \in V$ and for every $\tau \in T(C, D)$:

$$[\forall_{\Delta} \alpha. \tau, \eta] = \forall_{\Delta} \alpha. [\tau, \eta'] \quad \text{where} \quad \forall \beta \in V. \eta'(\beta) = \begin{cases} \alpha & \text{if } \beta = \alpha \\ \eta(\beta) & \text{otherwise} \end{cases} \quad \square$$

To exemplify this definition, consider the type expressions $\tau_1 = (\alpha \rightarrow (\forall \alpha. \alpha))$ and $\tau_2 = (\text{Maybe } \beta)$ where $\alpha, \beta \in V$ are type variables. Note that the set of bound type variables of τ_1 and the set of free type variables of τ_2 are disjoint. Then, $\tau_1[\tau_2/\alpha]$ yields the type expression $(\text{Maybe } \beta \rightarrow (\forall \alpha. \alpha))$.

Another example is the following one. Let $\tau = (\alpha \rightarrow \beta)$, let $\tau_1 = (\text{Maybe } \beta)$ and let $\tau_2 = \text{Int}$ where $\alpha, \beta \in V$ are type variables. Then, $\tau[\tau_1/\alpha, \tau_2/\beta] = (\text{Maybe } \beta \rightarrow \text{Int})$. Note that the type expression yielded by replacing α and β in τ still contains β which, however, has its origin in τ_1 and not in τ .

The restrictions about bound and free type variables on the occurring type expressions in Definition 3 are essential to avoid that free variables are getting bound. As an example, let $\alpha, \alpha_1 \in V$ be type variables and let $\tau = (\forall \alpha. \alpha \rightarrow \alpha_1)$ and $\tau_1 = (\text{Maybe } \alpha)$ be type expressions. The type expression $\tau[\tau_1/\alpha_1]$ is not defined because α occurs both free in τ_1 and bound in τ . If these restrictions were dropped, then $\tau[\tau_1/\alpha_1] = (\forall \alpha. \alpha \rightarrow \text{Maybe } \alpha)$. That means, the free type variable α of τ_1 would get bound in $\tau[\tau_1/\alpha_1]$. However, by renaming α in τ to a new type variable $\beta \in V$, the restrictions can still be fulfilled, and α_1 can be replaced with τ_1 without free type variables getting bound.

3.3 Type constructors and type classes

The purpose of this section is to give necessary restrictions which specify a subset C_* of the ranked set of type constructors C and a subset D_* of the set of type classes D , such that type expressions over C_* and D_* are eligible for deriving parametricity results.

In Haskell, the five types *Int*, *Integer*, *Float*, *Double* and *Char* are provided as primitive data types for efficiency reasons. It is convenient to treat these five types as nullary type constructors and call them *primitive type constructors*. With respect to the Haskell program H assumed in the beginning of this chapter, the set C_p of primitive type constructors occurring in H is defined as $C \cap \{Int, Integer, Float, Double, Char\}$.

The ranked set C of type constructors can be partitioned into four ranked sets with regard to H . These four ranked sets are the set of type constructors of algebraic data types C_a , the set of type constructors of type renamings C_r , the set of type constructors of type synonyms C_s and finally the set C_p as introduced earlier. Note that Haskell's list type constructor $[]$ as well as the tuple type constructors as for instance $(,)$ are considered to be elements of the ranked set C_a . The elements of the two ranked sets C_a and C_r are referred to as *user-defined type constructors* to distinguish them from other type constructors. Every user-defined type constructor and every type constructor in C_s has exactly one associated declaration in H , but not all declarations are valid under the light of further constructions in subsequent sections. Therefore, Definition 4, Definition 7 and Definition 8 restrict declarations to valid ones, and finally, Definition 10 subsumes the requirements necessary for a type constructor to be permissible.

Definition 4. Let $n \in \mathbb{N}_0$ and let $\sigma \in C_s^{(n)}$ be a type constructor of a type synonym. Let $\alpha_1, \dots, \alpha_n$ be n pairwise distinct type variables and let $\tau \in T(C \setminus \{\sigma\}, D)$ be a type expression having free type variables only in $\{\alpha_1, \dots, \alpha_n\}$. Then, a *valid declaration of σ* is as follows.

$$\mathbf{type} \ \sigma \ \alpha_1 \dots \alpha_n = \tau$$

The type expression τ is also referred to as *right-hand side of σ* and, for every $i \in \{1, \dots, n\}$, the type variable α_i is called the *i -th type variable of σ* . The set of all type constructors in C_s having a valid declaration in H is denoted by \tilde{C}_s . \square

Examples of Haskell type synonyms with valid declarations can be found in Figure 2.5 on page 14 and in Section 2.1 on page 15. Note that there are Haskell type synonym declarations which are not valid according to Definition 4. As an example, consider the following declaration.

$$\mathbf{type} \ Apply \ \alpha \ \beta = \alpha \ \beta$$

This declaration is not valid, because $(\alpha \ \beta)$ is not a type expression in $T(C \setminus \{Apply\}, D)$ (compare to Definition 1 on page 19).

Valid type synonym declarations may have type constructors of type synonyms occurring in their right-hand side which do not have a valid declaration. The following definition restricts the set \tilde{C}_s appropriately.

Definition 5. Let $\sigma \in \tilde{C}_s$ be a type constructor of a type synonym with a valid declaration, and let $\tau_\sigma \in T(C, D)$ be its right-hand side. If $X \subseteq C$ is the smallest set such that $\tau_\sigma \in T(X, D)$, then $C_\sigma = X \cap C_s$ denotes the set of type constructors of type synonyms occurring in the declaration of σ . Let $d = \{(\sigma_1, \sigma_2) \mid \sigma_2 \in \tilde{C}_s \wedge \sigma_1 \in C_{\sigma_2}\} \subseteq C_s \times C_s$ be a relation. Then, the tuple (C_s, d) is a directed graph, and the restricted, acyclic subset

3 Parametricity

of \tilde{C}_s is denoted by \hat{C}_s . The elements of \hat{C}_s are called *valid type constructors of type synonyms*. \square

Note that the Haskell report requires that type synonym declarations are neither recursive nor mutually recursive. Although the former is explicitly ensured by Definition 4 on the previous page and the latter is implicitly included in Definition 5 on the preceding page, both requirements are guaranteed already a priori, because every type synonym in C_s is taken from the syntactically correct Haskell program H assumed in the beginning of this chapter. Nevertheless, this formalisation is given here for clarification.

The Haskell report states that type synonyms introduce new types which are equivalent to existing types. To explain that, let $n \in \mathbb{N}_0$ and let $\sigma \in \hat{C}_s^{(n)}$ be a valid type constructor of a type synonym. For every $i \in \{1, \dots, n\}$, let $\alpha_i \in V$ be the i -th type variable of σ and let $\tau \in T(C \setminus \{\sigma\}, D)$ be the right-hand side of σ . Then, for all type expressions $\tau_1, \dots, \tau_n \in T(C, D)$, the type expression $(\sigma \tau_1 \dots \tau_n)$ is equivalent to $\tau'[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$ where τ' is obtained from τ by renaming every bound type variable such that, for every $i \in \{1, \dots, n\}$, the set of bound type variables of τ' is disjoint from the set of free type variables of τ_i . For abbreviation, let $(\sigma, [\tau_1, \dots, \tau_n])$ stand for $\tau'[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$.

With this notion of equivalence, given an arbitrary type expression, there exists an equivalent type expression without valid type constructors of type synonyms.

Definition 6. The function $f_s : T(C, D) \rightarrow T(C \setminus \hat{C}_s, D)$ maps a type expression to an equivalent one without valid type constructors of type synonyms and is defined by structural induction as follows.

- For every $\alpha \in V$: $f_s(\alpha) = \alpha$
- For every $n \in \mathbb{N}_0$, for every $\sigma \in C^{(n)}$ and for every $\tau_1, \dots, \tau_n \in T(C, D)$:

$$f_s(\sigma \tau_1 \dots \tau_n) = \begin{cases} f_s((\sigma, [\tau_1, \dots, \tau_n])) & \text{if } \sigma \in \hat{C}_s \\ (\sigma f_s(\tau_1) \dots f_s(\tau_n)) & \text{otherwise} \end{cases}$$

- For every $\tau_1, \tau_2 \in T(C, D)$: $f_s(\tau_1 \rightarrow \tau_2) = (f_s(\tau_1) \rightarrow f_s(\tau_2))$
- For every $\Delta \subseteq D$, for every $\alpha \in V$ and for every $\tau \in T(C, D)$:

$$f_s(\forall_{\Delta} \alpha. \tau) = (\forall_{\Delta} \alpha. f_s(\tau)) \quad \square$$

Note that this definition is well-defined because declarations of type synonyms are neither recursive nor mutually recursive, and because the ranked set of valid type constructors of type synonyms is finite. Note further, that, in the second item of the definition, only valid type constructors of type synonyms are replaced, while any other type constructor is kept unmodified.

Example 3. Let $\alpha, \beta \in V$ be type variables. Consider further the two following declarations for valid type constructors of type synonyms.

$$\begin{aligned} \mathbf{type} \ T1 \ \alpha &= (\alpha, \forall_{\{Eq\}} \beta. \beta \rightarrow \alpha) \\ \mathbf{type} \ T2 \ \alpha &= T1 \ [\alpha] \end{aligned}$$

Application of f_s to the type expression $(T2 \ \beta)$ then yields $([\beta], \forall_{\{Eq\}} \gamma. \gamma \rightarrow [\beta])$ where $\gamma \in V$ is a new type variable. Note that, as an intermediate step, the renaming of the bound type variable β to γ in the declaration of $T1$ is a necessary prerequisite for replacing α with the type expression $[\beta]$. \square

Since valid type synonyms are interchangeable with their right-hand side as defined by Definition 6 on the preceding page, they can be omitted from now on. Thus, by excluding also all other type synonyms, the reduced ranked set of type constructors is defined as $C_+ = C_a \cup C_r \cup C_p$.

The next definitions make use of simplified type expressions without occurring function type constructors or type abstractions. Such simplified type expressions can be defined similarly to Definition 1 on page 19, by omitting the last two items of that definition. Thus, for every ranked set of type constructors $\hat{C} \subseteq C$, let $\hat{T}(\hat{C})$ denote the set of type expressions without occurring function type constructors and type abstractions. Note that, since type classes occur only in type abstractions, the set $\hat{T}(\hat{C})$ is defined without referring to D .

Definition 7. Let $n \in \mathbb{N}_0$ and let $\sigma \in C_a^{(n)}$ be a type constructor of an algebraic data type. Let $\alpha_1, \dots, \alpha_n \in V$ be n pairwise distinct type variables. Furthermore, let $k \in \mathbb{N}$ and let ξ_1, \dots, ξ_k be k pairwise distinct symbols referred to as *data constructors*. For every $i \in \{1, \dots, k\}$, let $l_i \in \mathbb{N}_0$ and, for every $j \in \{1, \dots, l_i\}$, let $\tau_{i,j} \in \hat{T}(C_+)$ be a type expression without any occurring function type constructor or type abstraction constructor and additionally having free type variables only in $\{\alpha_1, \dots, \alpha_n\}$. For every $i \in \{1, \dots, k\}$ and for every $j \in \{1, \dots, l_i\}$, let $\tau'_{i,j}$ be either $!\tau_{i,j}$ or $\tau_{i,j}$ where $!\tau_{i,j}$ is called a *type expression with a strictness flag*. Then, a *valid declaration of σ* is as follows.

$$\begin{aligned} \mathbf{data} \ \sigma \ \alpha_1 \dots \alpha_n &= \begin{array}{l} \xi_1 \ \tau'_{1,1} \dots \tau'_{1,l_1} \\ | \ \xi_2 \ \tau'_{2,1} \dots \tau'_{2,l_2} \\ \vdots \\ | \ \xi_k \ \tau'_{k,1} \dots \tau'_{k,l_k} \end{array} \end{aligned}$$

For every $i \in \{1, \dots, k\}$, the type variable α_i is referred to as the *i -th type variable of σ* . Also the order of the data constructors and their arguments is fixed by this declaration. Thus, for every $i \in \{1, \dots, k\}$ and for every $j \in \{1, \dots, l_i\}$, the pair (i, j) is called a *position of σ* , while the set of all positions of σ is denoted by $pos(\sigma)$. Furthermore, for every position $(i, j) \in pos(\sigma)$, the notation (σ, i, j) specifies the type expression $\tau_{i,j}$. The set of all type constructors of algebraic data types with valid declarations in H is denoted by \tilde{C}_a . \square

3 Parametricity

Examples of algebraic data types with valid declarations are given in Section 2.1, especially in Figure 2.5 on page 14.

Definition 8. Let $n \in \mathbb{N}_0$ and let $\sigma \in C_r^{(n)}$ be a type constructor of a type renaming. Let $\alpha_1, \dots, \alpha_n \in V$ be n pairwise distinct type variables and let ξ be a *data constructor*. Furthermore, let $\tau \in \hat{T}(C_+)$ be a type expression without any function type constructor or type abstraction and additionally having free type variables only in $\{\alpha_1, \dots, \alpha_n\}$. Then, a *valid declaration of σ* is as follows.

$$\mathbf{newtype} \sigma \alpha_1 \dots \alpha_n = \xi \tau$$

Similar to algebraic data types, the following notation is fixed. For every $i \in \{1, \dots, n\}$, the type variable α_i is called the *i -th type variable of σ* . The set $\{(1, 1)\}$ containing the only *position of σ* is denoted by $pos(\sigma)$, and the type expression at that position is $(\sigma, 1, 1) = \tau$. The set of all type constructors of type renamings with valid declarations in H is denoted by \tilde{C}_r . \square

An example of a type renaming with a valid declaration is shown in Section 2.1 on page 15.

The following definition introduces the notion of *nested type constructors* [BM98]. This notion is needed afterwards to restrict user-defined type constructors to permissible ones.

Definition 9. Let $n \in \mathbb{N}$ and let $\sigma \in (\tilde{C}_a \cup \tilde{C}_r)^{(n)}$ be a user-defined type constructor with a valid declaration. For every $i \in \{1, \dots, n\}$, let $\alpha_i \in V$ be the i -th type variable of σ . Let $\beta \in (V \setminus \{\alpha_1, \dots, \alpha_n\})$ be a new type variable. If there is a position $(i, j) \in pos(\sigma)$, if there is a type expression $\tau \in \hat{T}(C_+)$ such that β is free in τ , and if, for some $\tau_1, \dots, \tau_n \in \hat{T}(C_+)$, the type expression $\tau[(\sigma \tau_1 \dots \tau_n)/\beta]$ equals (σ, i, j) where $\tau_i \neq \alpha_i$ for some $i \in \{1, \dots, n\}$, then σ is called a *nested type constructor*. \square

Example 4. In [HP06], an efficient implementation for sequences is described based on the following declaration of a nested type constructor.

$$\begin{aligned} \mathbf{data} \textit{FingerTree} \alpha &= \textit{Empty} \\ &| \textit{Single} \alpha \\ &| \textit{Deep} [\alpha] (\textit{FingerTree} (\textit{Node} \alpha)) [\alpha] \end{aligned}$$

Here, *Node* is a unary type constructor and $\alpha \in V$ is a type variable. The type expression at position (3, 2), namely $(\textit{FingerTree} (\textit{Node} \alpha))$, makes the type constructor *FingerTree* nested, because $(\textit{Node} \alpha)$ is not equal to α . \square

Opposed to the Haskell report, declarations of user-defined type constructors are additionally required to not be mutually recursive. An example for mutually recursive declarations is as follows.

$$\begin{aligned} \mathbf{data} \textit{Tree} &= \textit{Leaf} | \textit{Node} \textit{TreePair} \\ \mathbf{newtype} \textit{TreePair} &= \textit{TreePair} (\textit{Tree}, \textit{Tree}) \end{aligned}$$

Additionally to the restrictions that declarations are not mutually recursive and that type constructors are not nested, user-defined type constructors without valid declarations must not occur in declarations of permissible user-defined type constructors. The following definition specifies a subset of $(C_a \cup C_r)$ with respect to these three restrictions. To simplify notation, let $C_u = C_a \cup C_r$ and let $\tilde{C}_u = \tilde{C}_a \cup \tilde{C}_r$.

Definition 10. Let $n \in \mathbb{N}_0$ and let $\sigma \in \tilde{C}_u^{(n)}$ be a user-defined type constructor with a valid declaration. The set of all user-defined type constructors occurring in the declaration of σ , denoted by C_σ , is the smallest set $X \subseteq C_u$ such that, for every position (i, j) of σ , the type expression (σ, i, j) is in $\hat{T}(X \cup C_p)$. Let $d = \{(\sigma_1, \sigma_2) \mid (\sigma_2 \in \tilde{C}_u) \wedge (\sigma_1 \in C_{\sigma_2})\}$ be a binary relation over C_u . Then, the tuple (C_u, d) is a directed graph. Let $\tilde{C}'_u \subseteq \tilde{C}_u$ be the set of all user-defined type constructors with valid declarations which are not nested. Then, the restricted, acyclic subset of \tilde{C}'_u is denoted by \hat{C}_u . The ranked set of *permissible type constructors* C_* is defined as $C_p \cup \hat{C}_u$. \square

For example, the type constructors `[]`, `Bool`, `Maybe` and `Either`, given in Section 2.1 on page 12 and in Figure 2.5 on page 14, are permissible.

The remainder of this section is concerned with type classes. Similar to user-defined type constructors, the syntax of type class declarations is discussed before necessary restrictions for later constructions are defined.

Definition 11. Let $n, k \in \mathbb{N}_0$. Let $\alpha \in V$ be a type variable and let $\delta, \delta_1, \dots, \delta_n \in D$ be $n + 1$ pairwise distinct type classes. Furthermore, let f_1, \dots, f_k be k pairwise distinct symbols referred to as *class methods of δ* . Let $\tau_1, \dots, \tau_k \in T(C_*, D \setminus \{\delta\})$ be type expressions such that, for every $i \in \{1, \dots, k\}$, the set of free variables of τ_i equals $\{\alpha\}$. Then, a valid declaration of the type class δ is as follows.

$$\mathbf{class} (\delta_1 \alpha, \dots, \delta_n \alpha) \Rightarrow \delta \alpha \mathbf{where} f_1 :: \tau_1 \dots f_k :: \tau_k$$

The set $\{\delta_1, \dots, \delta_n\}$ of type classes is referred to as the set of *superclasses of δ* . The type variable α is called the *type variable of δ* . The order of the class methods is fixed by this declaration. For every $i \in \{1, \dots, k\}$, the type expression of the class method f_i is τ_i . The set of all type classes in D which have a valid declaration in H is denoted by \tilde{D} . \square

Note that the keyword **where** is dropped if there is no class method. Similarly, parentheses around superclasses are omitted if there is only one superclass. If there is no superclass at all, then the parentheses and the arrow are dropped.

Type classes with valid declarations are, for example, the standard Haskell classes `Eq` and `Ord` shown in Figure 2.6 on page 16. Note that it is possible to declare Haskell 98 type classes whose declarations are invalid with respect to the preceding definition. One such type class is shown in the following example.

Example 5. Consider the following type class.

$$\mathbf{class} C \alpha \mathbf{where}$$

$$f :: \forall \beta. \mathit{FingerTree} \alpha \rightarrow \beta \alpha$$

This declaration is invalid with respect to Definition 11 on the preceding page for two reasons. First, $(\beta \alpha)$ is not a type expression according to Definition 1 on page 19, and second, *FingerTree* is not a permissible type constructor (see Example 4 on page 26 for its declaration). \square

The Haskell report requires that the superclass relation must be acyclic. Additionally to this and to ensure well-definedness of later constructions, type classes must not be mutually depending.

Example 6. The following two type classes A and B are mutually depending.

$$\begin{aligned} \text{class } A \alpha \text{ where } a &:: \forall_{\{B\}} \beta. \alpha \rightarrow \beta \\ \text{class } B \beta \text{ where } b &:: \forall_{\{A\}} \alpha. \alpha \rightarrow \beta \end{aligned}$$

Note that B occurs in the type expression of the class method a of A , while the type class A occurs in the type expression of b in the declaration of B . \square

Similarly as for user-defined type constructors, the following definition specifies the subset of \tilde{D} , which incorporates the previously mentioned restrictions. Additionally, it ensures that type classes without valid declarations do not occur in declarations of permissible type classes declarations.

Definition 12. Let $\delta \in \tilde{D}$ be a type class with a valid declaration, let $\Delta \subseteq D$ be the set of superclasses of δ and let $k \in \mathbb{N}_0$ be the number of class methods of δ . Furthermore, let $X \subseteq D$ be the smallest set such that, for every $i \in \{1, \dots, k\}$, the i -th type expression of δ is an element of $T(C_*, X)$. Then, the set of all type classes on which δ depends is denoted by $D_\delta = \Delta \cup X$. Let $d = \{(\delta_1, \delta_2) \mid (\delta_2 \in \tilde{D}) \wedge (\sigma_1 \in D_{\sigma_2})\}$ be a binary relation over D . Then, the tuple (D, d) is a directed graph, and the restricted, acyclic subset of \tilde{D} is denoted by D_* . The elements of D_* are called *permissible type classes*. \square

For example, the type classes *Eq* and *Ord* given in Figure 2.6 on page 16 are permissible.

Definition 13. The set $T(C_*, D_*)$ of type expressions over permissible type constructors and permissible type classes is written as T , while T_c abbreviates the set $T_c(C_*, D_*)$. \square

3.4 Semantics of closed type expressions

Although there is not yet a formal semantics of Haskell, it is practice in several publications to assume a naive denotational semantics (see for example [DJ04, JV06, Voi02]). In that setting, every closed type expression $\tau \in T_c$ can be interpreted as a set of similar values in two ways, either as a plain set $\langle \tau \rangle$ or as a set $\langle \tau \rangle^\perp$ equipped with a complete partial order, the *semantic approximation partial order*. The intuition of this order is to describe that one value is less defined than another one. Without going into details, this section is only motivating these interpretations by means of examples.

Corresponding to the type Int is the plain set $\langle Int \rangle = \{-2^{31}, \dots, -1, 0, 1, \dots, 2^{31} - 1\}$ and the ordered set $\langle Int \rangle^\perp = \{\perp\} \cup \langle Int \rangle$. Under the semantic approximation partial order, \perp is less defined than every element of $\langle Int \rangle$, and any two different elements of $\langle Int \rangle$ are incomparable.

Consider the two user-defined type constructors D and N given in Section 2.1 on page 15. Their set interpretations are as follows.

$$\begin{aligned} \langle D \rangle &= \{(D\ i) \mid i \in \langle Int \rangle\} & \langle D \rangle^\perp &= \{\perp\} \cup \{(D\ i) \mid i \in \langle Int \rangle^\perp\} \\ \langle N \rangle &= \{(N\ i) \mid i \in \langle Int \rangle\} & \langle N \rangle^\perp &= \{(N\ i) \mid i \in \langle Int \rangle^\perp\} \end{aligned}$$

Note that the ordered sets $\langle D \rangle^\perp$ and $\langle N \rangle^\perp$ differ in that the former has an additional value \perp . The semantic approximation order for $\langle D \rangle^\perp$ is as follows. Its least element is \perp , and $(D\ \perp)$ is less defined than $(D\ i)$ for any $i \in \langle Int \rangle^\perp$. Opposed to that, the least element of $\langle N \rangle^\perp$ with regard to the semantic approximation order is $(N\ \perp)$. Note that, for any $i, j \in \langle Int \rangle^\perp \setminus \{\perp\}$, where $i \neq j$, the values $(D\ i)$ and $(D\ j)$ as well as $(N\ i)$ and $(N\ j)$ are incomparable.

Another example is presented by means of the list type constructor. The interpretation of $[Int]$ as a plain set $\langle [Int] \rangle$ of values is the set of all finite lists whose elements are in $\langle Int \rangle$. For example, the empty list $[]$ and the list $[0, 1]$ are both elements of $\langle [Int] \rangle$. The ordered set $\langle [Int] \rangle^\perp$ is the set which contains all finite, partial and infinite lists over $\langle Int \rangle^\perp$. In addition to the elements of the plain set of integer lists, it includes, for example, the finite list $[0, \perp, 1]$ and the partial list $(2 : \perp)$ as well as the infinite list constructed by the function *ones* (see Section 2.1 on page 14). The semantic approximation order for lists of integers is as follows. Besides \perp being the least element and $[] \sqsubseteq []$, the list $(a : as)$ is less defined than $(b : bs)$ iff $a \sqsubseteq b$ and $as \sqsubseteq bs$.

Type expressions describing the type of functions can be interpreted as sets, too. Thus, there is also a plain set and an ordered set for the type expression $(Int \rightarrow Int)$. Note that Haskell functions are considered to be monotonic and continuous, and therefore, both the plain and the ordered set comprise monotonic and continuous functions only. The plain set $\langle Int \rightarrow Int \rangle$ contains all monotonic and continuous mathematical functions f which map values of $\langle Int \rangle$ to values of $\langle Int \rangle$. This includes, for example, the function which negates integers. In contrast, the ordered set $\langle Int \rightarrow Int \rangle^\perp$ contains functions mapping values of $\langle Int \rangle^\perp$ to values of $\langle Int \rangle^\perp$. An example is the function $(\backslash x \rightarrow \perp)$ mapping every value of $\langle Int \rangle^\perp$ to \perp . Also, a partial function which maps 0 to 0 and is undefined for any other argument, that means, it maps $i \in (\langle Int \rangle^\perp \setminus \{0\})$ to \perp , is an element of the ordered set. A function $f \in \langle Int \rightarrow Int \rangle^\perp$ is less defined than a function $g \in \langle Int \rightarrow Int \rangle^\perp$, if, for all $x \in \langle Int \rangle^\perp$, it holds that $f\ x \sqsubseteq g\ x$ and $f \neq \perp$ implies $g \neq \perp$.

Characterising the set of values corresponding to type abstractions is more complicated. Consider first the following examples. The function $(const\ 0)$ may be considered as an element of $\langle \forall \alpha. \alpha \rightarrow Int \rangle$ (see Figure 2.4 on page 13). Instantiating the quantified type variable α to Int yields a function which maps every integer to the constant value 0. In the same way, the type $Bool$ may be taken for α , which results in a function contained in $\langle Bool \rightarrow Int \rangle$ and which also returns 0 for every argument. Note that this behaviour of $(const\ 0)$, that is, mapping every argument to 0, holds for every type. A similar

observation can be made in the case of ordered sets. For instance, consider the function *length* (see Figure 2.4) as an element of $\langle \forall \alpha. [\alpha] \rightarrow \text{Int} \rangle^\perp$. Choosing α to be *Int* gives a function which maps finite, partial and infinite lists of integers to an element of $\langle \text{Int} \rangle^\perp$; more specific, this function computes the length of any finite list of integers and returns \perp for every partial or infinite list. Similarly, taking *Bool* for α results in a function in $\langle [\text{Bool}] \rightarrow \text{Int} \rangle^\perp$, which computes the length of lists with boolean elements. Observe that the function *length* behaves in both cases in the same way, independent of the chosen types. This independence of polymorphic values, that is, the same behaviour at different instantiating types, led to the discovery of the parametricity theorem in [Rey83]. Additionally, this is the characteristic property underlying plain and ordered sets corresponding to type abstractions.

The instantiation of a polymorphic value is called *type instantiation*. It can be described more formally as follows. Let $\tau_1 \in T$ be a type expression with at most one free type variable, say $\alpha \in V$, and let $\tau_2 \in T_c$ be a closed type expression. For every element $x \in \langle \forall \alpha. \tau_1 \rangle$, the instance at type τ_2 is denoted by x_{τ_2} and is an element of $\langle \tau_1[\tau_2/\alpha] \rangle$. Similarly, for every element $x \in \langle \forall \alpha. \tau_1 \rangle^\perp$, the instance at type τ_2 is denoted by x_{τ_2} and is an element of $\langle \tau_1[\tau_2/\alpha] \rangle^\perp$.

The notion of an instance has also another meaning in Haskell. A closed type expression can be an *instance of a type class*, if there are implementations of all class methods for that type expression. For example, the primitive type *Int* is an instance of the type class *Eq* (see Figure 2.6 on page 16), because the two class methods of *Eq*, equality test and inequality test, are implemented for *Int*. Given a set of type classes $\Delta \subseteq D_*$, the set T_Δ denotes the set of all closed type expressions which are instances of every type class in Δ .

Now, type instantiations can be generalised as follows. Let $\Delta \subseteq D_*$, let $\tau_1 \in T$ be a type expression with at most one free type variable, say $\alpha \in V$, and let $\tau_2 \in T_\Delta$. For every element $x \in \langle \forall \alpha. \tau_1 \rangle$, the instance x_{τ_2} is an element of $\langle \tau_1[\tau_2/\alpha] \rangle$, and, similarly, for every element $x \in \langle \forall \alpha. \tau_1 \rangle^\perp$, the instance x_{τ_2} is an element of $\langle \tau_1[\tau_2/\alpha] \rangle^\perp$.

3.5 Relational actions

In the preceding section, a naive denotational semantics of closed type expressions was introduced, which now forms the basis for defining binary relations.

Definition 14. Let $\tau_1, \tau_2 \in T_c$ be two closed type expressions. The set of all binary relations in $\mathcal{P}(\langle \tau_1 \rangle^\perp \times \langle \tau_2 \rangle^\perp)$ is denoted by $Rel(\tau_1, \tau_2)$, and the set of all relations in $\mathcal{P}(\langle \tau_1 \rangle \times \langle \tau_2 \rangle)$ is written as $Rel^{(basic,=)}(\tau_1, \tau_2)$. Special subsets of $Rel(\tau_1, \tau_2)$ are given in the following list.

$$Rel^{(fix,=)}(\tau_1, \tau_2) = \{R \in Rel(\tau_1, \tau_2) \mid R \text{ is admissible}\}$$

$$Rel^{(fix,\sqsubseteq)}(\tau_1, \tau_2) = \{R \in Rel(\tau_1, \tau_2) \mid R \text{ is admissible and left-closed}\}$$

$$Rel^{(seq,=)}(\tau_1, \tau_2) = \{R \in Rel(\tau_1, \tau_2) \mid R \text{ is admissible and bottom-reflecting}\}$$

$$Rel^{(seq,\sqsubseteq)}(\tau_1, \tau_2) = \{R \in Rel(\tau_1, \tau_2) \mid R \text{ is admissible, total and left-closed}\}$$

The union of all sets $Rel(\tau_1, \tau_2)$ for arbitrary closed type expressions $\tau_1, \tau_2 \in T_c$ is denoted by Rel . In the same way, the sets $Rel^{(basic,=)}$, $Rel^{(fix,=)}$, $Rel^{(fix,\sqsubseteq)}$, $Rel^{(seq,=)}$ and $Rel^{(seq,\sqsubseteq)}$ can be defined. \square

A partial mapping from the set V of type variables into the set Rel is called an *environment* and is denoted by η . The empty environment is written as \emptyset , while the update or modification of an environment η by the mapping of a type variable $\alpha \in V$ to a relation $R \in Rel$ is denoted by $\eta[R/\alpha]$. If $m \in M$ is a model and η maps type variables only to relations in Rel^m , then η is called *restricted to m* . If η contains mappings for every free type variable of a type expression $\tau \in T$, then η is said to *close τ* . If η is additionally restricted to a model $m \in M$, then η is said to *close τ under m* . Let $\tau \in T$ be a type expression such that η closes τ . For every free type variable $\alpha \in V$ of τ , let $\tau_{\alpha,1}, \tau_{\alpha,2} \in T_c$ be two closed type expressions such that $\eta(\alpha) \in Rel(\tau_{\alpha,1}, \tau_{\alpha,2})$. Then, the pair of closed type expressions $(\tau_1, \tau_2) \in T_c \times T_c$, where τ_1 is obtained from τ by replacing every free type variable α with $\tau_{\alpha,1}$ and where τ_2 is obtained from τ by replacing every free variable α with $\tau_{\alpha,2}$, is referred to as the *closure of τ under η* .

Example 7. Let $\alpha \in V$ be a type variable, let $\tau = ([\alpha] \rightarrow Id)$ and let $m = (fix, \sqsubseteq)$. Choosing the closed type expressions $\tau_{\alpha,1} = Char$ and $\tau_{\alpha,2} = Bool$ and taking an arbitrary relation $R \in Rel^{(fix,\sqsubseteq)}(Char, Bool)$, any environment η restricted to (fix, \sqsubseteq) and having $\eta(\alpha) = R$ is closing τ under (fix, \sqsubseteq) . The type expression pair $([Char] \rightarrow Int, [Bool] \rightarrow Int)$ is then the closure of τ under η . \square

For every closed type expression, two *identity relations* can be defined as follows.

Definition 15. Let $\tau \in T_c$ be a closed type expression. The identity relation $id_\tau \in Rel^{(basic,=)}(\tau, \tau)$ is defined as $\{(x, x) \mid x \in \langle \tau \rangle\}$, while the identity relation $id_\tau^\perp \in Rel(\tau, \tau)$ is defined as $\{(x, x) \mid x \in \langle \tau \rangle^\perp\}$. \square

Note that, for every $\sigma \in C_*^{(0)}$, the relation id_σ^\perp is strict, continuous and bottom-reflecting. Examples of identity relations are the ones for the Haskell type $Bool$, namely $id_{Bool} = \{(False, False), (True, True)\}$ and $id_{Bool}^\perp = \{(\perp, \perp)\} \cup id_{Bool}$.

Before defining the relational actions of type constructors having a rank greater than zero, the following examples motivates a construction needed in that definition.

Example 8. Consider the type constructor $Tree$ as shown in Section 2.1 on page 12, which has $\alpha \in V$ as its first type variable. With the help of a new type variable $\beta \in V \setminus \{\alpha\}$, a new type constructor $Tree_\beta$ can be declared. Its declaration resembles that of $Tree$, in that it also has α as its first type variable, it has the same data constructors in the same order, and it has the same set of positions. The difference between $Tree$ and $Tree_\beta$, however, is that β is the second type variable of $Tree_\beta$, and every occurrence of $(Tree \alpha)$ in the right-hand side of $Tree_\beta$ is replaced with β . Thus, the declaration of $Tree_\beta$ is as follows.

```
data Treeβ α β = Leaf | Node α β β
```

Note that $(Tree_\beta \alpha (Tree \alpha))$ equals $(Tree \alpha)$ and that $Tree_\beta$ is not recursive. Also note that $(Tree_\beta, 2, 2) = \beta$ and $(Tree_\beta, 2, 3) = \beta$. \square

3 Parametricity

The construction of this example can be applied to every possibly recursive declaration of a permissible type constructor. Let $n \in \mathbb{N}$ and let $\sigma \in C_*^{(n)}$ be a permissible type constructor of an algebraic data type or a type renaming. For every $i \in \{1, \dots, n\}$, the i -th type variable of σ shall be $\alpha_i \in V$. With $\beta \in V \setminus \{\alpha_1, \dots, \alpha_n\}$, a new type variable is denoted which is distinct from all type variables of σ . Since σ is not nested, there exists, for every position $(i, j) \in \text{pos}(\sigma)$, a type expression $(\sigma_\beta, i, j) \in T$ such that $(\sigma_\beta, i, j)[(\sigma \alpha_1 \dots \alpha_n)/\beta] = (\sigma, i, j)$ and σ does not occur in (σ_β, i, j) .

Based on this construction, relational actions for permissible type constructors of rank greater than zero are defined in a similar way as in [Joh02]. Assume there is, for every environment η restricted to the $(\text{basic}, =)$ model, a mapping $\llbracket \cdot \rrbracket_\eta$ which maps every type expression $\tau \in T$ closed under η to a relation $\llbracket \tau \rrbracket_\eta \in \text{Rel}^{(\text{basic}, =)}(\tau_1, \tau_2)$, where $(\tau_1, \tau_2) \in T_c \times T_c$ is the closure of τ under η . Further assume there is, for every environment η a mapping $\llbracket \cdot \rrbracket_\eta^\perp$ which maps every type expression $\tau \in T$ closed under η to a relation $\llbracket \tau \rrbracket_\eta^\perp \in \text{Rel}(\tau_1, \tau_2)$, where $(\tau_1, \tau_2) \in T_c \times T_c$ is the closure of τ under η .

Definition 16. Let $n \in \mathbb{N}$ and let $\sigma \in C_*^{(n)}$ be a permissible type constructor having the type variables $\alpha_1, \dots, \alpha_n \in V$ in this order in its declaration. Let $k \in \mathbb{N}$ be the number of data constructors of σ . For every $i \in \{1, \dots, k\}$, the i -th data constructor is denoted by ξ_i , and $l_i \in \mathbb{N}_0$ stands for the number of arguments of ξ_i . Let $\beta \in V \setminus \{\alpha_1, \dots, \alpha_n\}$ be a type variable. Furthermore, let $\tau_1, \dots, \tau_n \in T$ and let η be an environment closing $(\sigma \tau_1 \dots \tau_n)$. Let $(\tau_{\sigma,1}, \tau_{\sigma,2}) \in T_c \times T_c$ be the closure of $(\sigma \tau_1 \dots \tau_n)$ under η . If σ is a type constructor of either an algebraic data type or a type renaming and η is restricted to the $(\text{basic}, =)$ model, then the relation $\text{lift}_\sigma(\llbracket \tau_1 \rrbracket_\eta, \dots, \llbracket \tau_n \rrbracket_\eta) \in \text{Rel}^{(\text{basic}, =)}(\tau_{\sigma,1}, \tau_{\sigma,2})$ is the smallest relation L such that

$$L = \bigcup_{i=1}^k \{(\xi_i x_1 \dots x_{l_i}, \xi_i y_1 \dots y_{l_i}) \mid \forall j \in \{1, \dots, l_i\}. (x_j, y_j) \in \llbracket (\sigma_\beta, i, j) \rrbracket_{\bar{\eta}[L/\beta]}\}$$

where $\bar{\eta}$ is an environment such that $\bar{\eta}(\alpha_i) = \llbracket \tau_i \rrbracket_\eta$ for every $i \in \{1, \dots, n\}$.

Now, let $\bar{\eta}(\alpha_i) = \llbracket \tau_i \rrbracket_\eta^\perp$ for every $i \in \{1, \dots, n\}$. If σ is a type constructor of an algebraic data type, then the relation $\text{lift}_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta, \dots, \llbracket \tau_n \rrbracket_\eta) \in \text{Rel}(\tau_{\sigma,1}, \tau_{\sigma,2})$ is the largest relation L such that

$$L = (\perp, \perp) \cup \bigcup_{i=1}^k \{(\xi_i x_1 \dots x_{l_i}, \xi_i y_1 \dots y_{l_i}) \mid \forall j \in \{1, \dots, l_i\}. (x_j, y_j) \in \llbracket (\sigma_\beta, i, j) \rrbracket_{\bar{\eta}[L/\beta]}^\perp\}$$

If σ is a type constructor of a type renaming, then the relation $\text{lift}_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta, \dots, \llbracket \tau_n \rrbracket_\eta) \in \text{Rel}(\tau_{\sigma,1}, \tau_{\sigma,2})$ is the largest relation L such that

$$L = \left\{ (\xi_1 x, \xi_1 y) \mid (x, y) \in \llbracket (\sigma_\beta, 1, 1) \rrbracket_{\bar{\eta}[L/\beta]}^\perp \right\} \quad \square$$

As opposed to the smallest relation in the bottom-free case, the relations obtained from the relational action lift^\perp are defined as the largest relations to allow for infinite structures. A detailed example using lists is given in Section 3.6.

Note that in the definition of the relational action $lift^\perp$ for type renamings, the pair (\perp, \perp) was omitted because type renamings are unlifted in Haskell, that is $(\xi \perp)$ equals \perp , where ξ is the data constructor of a type renaming.

A similar situation arises for type constructors of algebraic data types with strictness flags. Let $n \in \mathbb{N}$ and let $\sigma \in C_*^{(n)}$ be a type constructor of an algebraic data type with at least one strictness flag in its declaration. For every $l \in \{1, \dots, n\}$, let $\alpha_l \in V$ be the l -th type variable of σ and let $\tau_l \in T_c$ be a type expression. Let (i, j) be a position of σ at which a strictness flag occurs. Additionally, let ξ_i be the i -th data constructor of σ and let $k \in \mathbb{N}$ be the number of arguments of ξ_i . For every $l \in \{1, \dots, j-1, j+1, \dots, k\}$, let $x_l \in \langle (\sigma, i, l)[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n] \rangle^\perp$. Then $(\xi_i x_1 \dots x_{j-1} \perp x_{j+1} \dots x_k)$ equals \perp (see also Section 2.1 on page 14). Therefore, if the relation corresponding to the type expression at position (i, j) of σ is strict, the pair (\perp, \perp) is not required to be added explicitly to $lift_\sigma^\perp(R_1, \dots, R_n)$ for appropriate relations R_1, \dots, R_n , but this distinction is not made here.

The properties of $lift^\perp$ are as follows. Suppose that σ is a type constructor of an algebraic data type with rank $n \in \mathbb{N}$, suppose that $\tau_1, \dots, \tau_n \in T$ are relations and suppose that η is as in Definition 16. Then, the relation $lift_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta^\perp, \dots, \llbracket \tau_n \rrbracket_\eta^\perp)$ is strict by construction. It is additionally bottom-reflecting and continuous, if, for every environment η' , the mapping $\llbracket \cdot \rrbracket_{\eta'}^\perp$ maps type expressions only to bottom-reflecting and continuous relations, respectively. The relation $\sqsubseteq; lift_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta^\perp, \dots, \llbracket \tau_n \rrbracket_\eta^\perp)$ is left-closed and strict by construction. It is also total and continuous if, for every environment η' , the mapping $\llbracket \cdot \rrbracket_{\eta'}^\perp$ maps type expressions only to total and to continuous, left-closed relations, respectively. Proofs for selected type constructors can be found in [JV06].

The argumentation for type constructors of type renamings is similar. Suppose that σ is a type constructor of a type renaming and τ_1, \dots, τ_n as well as η are as before. Since type renamings are unlifted in Haskell, $lift_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta^\perp, \dots, \llbracket \tau_n \rrbracket_\eta^\perp)$ is strict and bottom-reflecting if, for every environment η' , the mapping $\llbracket \cdot \rrbracket_{\eta'}^\perp$ maps type expressions only to strict and bottom-reflecting relations, respectively. The relation $\sqsubseteq; lift_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta, \dots, \llbracket \tau_n \rrbracket_\eta)$ is left-closed by construction. It is strict and total if, for every environment η' , the mapping $\llbracket \cdot \rrbracket_{\eta'}^\perp$ maps only to strict and total relations, respectively. Similar to the case of algebraic data types, continuity of $lift_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta, \dots, \llbracket \tau_n \rrbracket_\eta)$ and $\sqsubseteq; lift_\sigma^\perp(\llbracket \tau_1 \rrbracket_\eta, \dots, \llbracket \tau_n \rrbracket_\eta)$ holds given that, for every environment η' , the mapping $\llbracket \cdot \rrbracket_{\eta'}^\perp$ maps type expressions only to continuous and continuous, left-closed relations, respectively, and using the fact that type renamings are unlifted in Haskell.

The relational action of the function type constructor is defined in four ways, depending on the underlying model. Nevertheless, all definitions have in common that they describe relations containing pairs of functions which map related arguments to related results.

Definition 17. For all closed type expressions $\tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$ and for all relations $R_1 \in Rel^{(basic, =)}(\tau_1, \tau'_1)$ and $R_2 \in Rel^{(basic, =)}(\tau_2, \tau'_2)$, the relational action of the function type constructor in the $(basic, =)$ model maps R_1 and R_2 to a relation in $Rel^{(basic, =)}(\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2)$ and is defined as follows.

$$R_1 \rightarrow^{(basic, =)} R_2 = \{(f, g) \mid \forall (x, y) \in R_1. (f x, g y) \in R_2\} \quad \square$$

Definition 18. Given four closed type expressions $\tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$, let $R_1 \in \text{Rel}(\tau_1, \tau'_1)$ and $R_2 \in \text{Rel}(\tau_2, \tau'_2)$ be two relations. Then, the relational action of the function type constructor creates a relation in $\text{Rel}(\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2)$ and is defined as follows.

$$R_1 \rightarrow R_2 = \{(f, g) \mid \forall(x, y) \in R_1. (f\ x, g\ y) \in R_2\} \quad \square$$

Note that, if R_2 is admissible, then the relation $R_1 \rightarrow R_2$ is also admissible because function application in Haskell is monotonic and continuous. If R_2 is additionally left-closed, then $R_1 \rightarrow R_2$ is also left-closed.

Definition 19. For all closed type expressions $\tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$ and for all admissible, bottom-reflecting relations $R_1 \in \text{Rel}^{(seq,=)}(\tau_1, \tau'_1)$ and $R_2 \in \text{Rel}^{(seq,=)}(\tau_2, \tau'_2)$, the relational action of the function type constructor in the $(seq, =)$ model maps R_1 and R_2 to an element of $\text{Rel}^{(seq,=)}(\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2)$ and has the following definition.

$$R_1 \xrightarrow{(seq,=)} R_2 = \{(f, g) \mid (f \neq \perp \Leftrightarrow g \neq \perp) \wedge (\forall(x, y) \in R_1. (f\ x, g\ y) \in R_2)\} \quad \square$$

Note that due to the explicitly added restriction the relation $R_1 \xrightarrow{(seq,=)} R_2$ is bottom-reflecting. It is admissible because R_2 is admissible and function application in Haskell is monotonic and continuous.

Definition 20. Let $\tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$ be closed type expressions. Additionally, let $R_1 \in \text{Rel}^{(seq, \sqsubseteq)}(\tau_1, \tau'_1)$ and $R_2 \in \text{Rel}^{(seq, \sqsubseteq)}(\tau_2, \tau'_2)$ be admissible, total and left-closed relations. The relational action of the function type constructor in the (seq, \sqsubseteq) model maps R_1 and R_2 to a relation in $\text{Rel}^{(seq, \sqsubseteq)}(\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2)$ and is defined as follows.

$$R_1 \xrightarrow{(seq, \sqsubseteq)} R_2 = \{(f, g) \mid (f \neq \perp \Rightarrow g \neq \perp) \wedge (\forall(x, y) \in R_1. (f\ x, g\ y) \in R_2)\} \quad \square$$

The explicit restriction in this definition forces the relation $R_1 \xrightarrow{(seq, \sqsubseteq)} R_2$ to be total. Admissibility and left-closedness follow from monotonicity and continuity of function application in Haskell and from R_2 being admissible and left-closed.

The definition of the relational action of type abstractions relies on the following mapping. Let $m \in M$ be a model, let $\Delta \subseteq D_*$, let $\tau_1, \tau_2 \in T_\Delta$ be type expressions, let $R \in \text{Rel}^m(\tau_1, \tau_2)$ and let $\alpha \in V$. Then assume there is a mapping $\llbracket \cdot \rrbracket_{[R/\alpha]}^m$ such that, for every type expression $\tau \in T$ having the single free variable α , the relation $\llbracket \tau \rrbracket_{[R/\alpha]}^m$ is in $\text{Rel}^m(\tau[\tau_1/\alpha], \tau[\tau_2/\alpha])$.

The next two definitions specify predicates which are then used in the definition of the relational action of type abstractions at the end of this section.

Definition 21. Let $m \in M$ be a model, let $\delta \in D_*$ be a permissible type class, let $\tau_1, \tau_2 \in T_{\{\delta\}}$ be two type expressions and let $R \in \text{Rel}^m(\tau_1, \tau_2)$ be a relation. Additionally, let $\alpha \in V$ be the type variable of δ . For every class method f of δ , let $f' :: \forall_{\{\delta\}} \alpha. \tau$ be the constrained version of f , where $\tau \in T$ is the type expression of f . Then R respects δ under m , denoted by $P(m, R, \delta)$, if, for all class methods f of δ , the tuple $(f'_{\tau_1}, f'_{\tau_2})$ is contained in $\llbracket \tau \rrbracket_{[R/\alpha]}^m$, where $\tau \in T$ is the type expression of f , and if $P(m, R, \delta')$ holds for all superclasses δ' of δ . □

Note that this is well-defined for permissible type classes, because their superclass hierarchy is acyclic and they are not mutually depending.

Definition 22. Let $m \in M$ be a model, let $\Delta \subseteq D_*$ be a set of permissible classes and let $R \in Rel^m$ be a relation. If $P(m, R, \delta)$ is true for all type classes $\delta \in \Delta$, then R respects Δ under m , denoted by $P(m, R, \Delta)$. \square

The relational action of type abstractions resembles that of the function type constructor in that it is defined point-wise. However, instead of relating functions, the relation obtained from the relational action of a type abstraction relates polymorphic values if all their instances are contained in a properly constructed relation.

Definition 23. Let $m \in M$ be a model and let $\Delta \subseteq D_*$. Let $\tau_1, \tau_2 \in T$ be two type expressions with at most one free type variable, say α . Furthermore, let F be a function which maps, for any two closed type expressions $\tau'_1, \tau'_2 \in T_\Delta$, every relation $R \in Rel^m(\tau'_1, \tau'_2)$ to a relation $F(R) \in Rel^m(\tau_1[\tau'_1/\alpha], \tau_2[\tau'_2/\alpha])$. The relational action of the type abstraction then gives a relation in $Rel^m(\forall_\Delta \alpha. \tau_1, \forall_\Delta \alpha. \tau_2)$ and is defined as follows.

$$\begin{aligned} \forall_\Delta R &\in Rel^m.F(R) \\ &= \{(u, v) \mid \forall \tau'_1, \tau'_2 \in T_\Delta, R \in Rel^m(\tau'_1, \tau'_2) . P(m, R, \Delta) \Rightarrow (u_{\tau'_1}, v_{\tau'_2}) \in F(R)\} \quad \square \end{aligned}$$

Admissibility of $\forall_\Delta R \in Rel^m.F(R)$, where $m \in M \setminus \{(basic, =)\}$, follows from admissibility of all relations $F(R)$ and monotonicity and continuity of type instantiation in Haskell. Left-closedness of $\forall_\Delta R \in Rel^m.F(R)$ with $m \in \{(fix, \sqsubseteq), (seq, \sqsubseteq)\}$ follows from left-closedness of all $F(R)$ and monotonicity of type instantiation in Haskell. See [JV06] for more detailed justifications in the case that Δ equals \emptyset . Bottom-reflectiveness and totality hold only with extra conditions; this will be explained in the following section.

3.6 Logical relations

Based on the relational actions of the preceding section, the following definition constructs logical relations. At the same time, it assigns the mappings assumed for Definition 16 on page 32 and Definition 23 to appropriate logical relations.

Definition 24. Given a type expression $\tau \in T$ and a model $m \in M$. Let η be an environment closing τ under m and let the pair $(\tau_1, \tau_2) \in T_c \times T_c$ be the closure of τ under η . Then, the logical relation $\llbracket \tau \rrbracket_\eta^m \in Rel^m(\tau_1, \tau_2)$ is defined by structural induction over τ as follows.

- For every $\alpha \in V$:

$$\llbracket \alpha \rrbracket_\eta^m = \eta(\alpha)$$

3 Parametricity

- For every permissible $\sigma \in C_*^{(0)}$:

$$\frac{\llbracket \sigma \rrbracket_\eta^m}{\begin{array}{l} \text{basic} \\ \text{fix} \\ \text{seq} \end{array}} \Bigg| \begin{array}{l} = \\ id_\sigma \\ id_\sigma^\perp \\ id_\sigma^\perp \end{array} \frac{}{\sqsubseteq}$$

- For every $n \in \mathbb{N}$, for every permissible $\sigma \in C_*^{(n)}$, for every $\tau_1, \dots, \tau_n \in T$ which are closed by η , for every $i \in \{1, \dots, n\}$, let $R_i = \llbracket \tau_i \rrbracket_\eta^m$. Then:

$$\frac{\llbracket \sigma \tau_1 \dots \tau_n \rrbracket_\eta^m}{\begin{array}{l} \text{basic} \\ \text{fix} \\ \text{seq} \end{array}} \Bigg| \begin{array}{l} = \\ \text{lift}_\sigma(R_1, \dots, R_n) \\ \text{lift}_\sigma^\perp(R_1, \dots, R_n) \\ \text{lift}_\sigma^\perp(R_1, \dots, R_n) \end{array} \frac{}{\sqsubseteq; \text{lift}_\sigma^\perp(R_1, \dots, R_n)}$$

The mapping $\llbracket \cdot \rrbracket_\eta^{(basic,=)}$ instantiates the mapping $\llbracket \cdot \rrbracket_\eta$ assumed in the definition of $lift$, and, for every model $m \in M \setminus \{(basic, =)\}$, the mapping $\llbracket \cdot \rrbracket_\eta^m$ instantiates the mapping $\llbracket \cdot \rrbracket_\eta^\perp$ assumed in the definition of $lift^\perp$. Note that this does not cause circularities for permissible type constructors.

- For every $\tau_1, \tau_2 \in T$ which are closed by η , let $R_1 = \llbracket \tau_1 \rrbracket_\eta^m$ and $R_2 = \llbracket \tau_2 \rrbracket_\eta^m$. Then:

$$\frac{\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_\eta^m}{\begin{array}{l} \text{basic} \\ \text{fix} \\ \text{seq} \end{array}} \Bigg| \begin{array}{l} = \\ R_1 \rightarrow^{(basic,=)} R_2 \\ R_1 \rightarrow R_2 \\ R_1 \rightarrow^{(seq,=)} R_2 \end{array} \frac{R_1 \rightarrow R_2}{R_1 \rightarrow^{(seq,\sqsubseteq)} R_2} \frac{}{\sqsubseteq}$$

- For every $\Delta \subseteq D_*$, every $\alpha \in V$ and for every $\tau' \in T$ such that η closes $\forall_\Delta \alpha. \tau'$:

$$\llbracket \forall_\Delta \alpha. \tau' \rrbracket_\eta^m = \left(\forall_\Delta R \in Rel^m. \llbracket \tau' \rrbracket_{\eta[R/\alpha]}^m \right)$$

The mapping $\llbracket \cdot \rrbracket_{\emptyset[R/\alpha]}^m$ instantiates the mapping $\llbracket \cdot \rrbracket_{[R/\alpha]}^m$ assumed for Definition 23 on the preceding page. Note that this does not cause circularities for permissible type constructors and permissible type classes. \square

The construction of the relational actions along with the requirements for the environment guarantee that all relations occurring in the logical relation for a model $m \in M$ are elements of the set Rel^m . Especially, that relations obtained from the relational action of type abstractions are bottom-reflecting in the $(seq, =)$ model and total in the (seq, \sqsubseteq) model can now be shown with indirect reasoning in a similar way as in [JV06], by using a suitable identity relation in the $(seq, =)$ model and a suitable relation \sqsubseteq in the (seq, \sqsubseteq) model.

Note that the relational action of the function type constructor is the same in both the $(fix, =)$ model and the (fix, \sqsubseteq) model, although the other relational actions of these two models differ. The reason for this is as follows. Every relation in the (fix, \sqsubseteq) model

has to be left-closed in addition to be admissible already required by the $(fix, =)$ model. As a consequence, extra requirements need to be added to the relational actions of type constructors and type abstraction. In contrast, the relational action of the function type constructor used by these two models propagates left-closedness along with strictness and continuity from its relation arguments to the resulting relation, and thus these models can share that relational action.

For every model in $M \setminus \{(basic, =)\}$ and for every type constructor having a rank greater than zero, it was already indicated in Section 3.5 that the according relational action is chosen to be the largest relation to allow for infinite structures. The following example demonstrates this in detail using the standard Haskell list type constructor, whose pseudo-declaration can be found in Section 2.1 on page 12.

Example 9. Consider the $(fix, =)$ model and the type expression $\tau = [\gamma]$, where $\gamma \in V$ is a type variable. Let $R \in Rel^{(fix, =)}$ and $\eta(\gamma) = R$ for an environment η . Furthermore, let $\bar{\eta}$ be an environment such that $\bar{\eta}(\alpha) = R$ where $\alpha \in V$ is the single type variable of the list type constructor $[\]$. The logical relation corresponding to τ is then $\llbracket \tau \rrbracket_{\eta}^{(fix, =)} = lift_{[\]}^{\perp}(R)$ which can be written in set notation according to Definition 16 on page 32 as follows, where $\beta \in V \setminus \{\alpha\}$ is a new type variable and $\eta' = \bar{\eta}[lift_{[\]}^{\perp}(R)/\beta]$.

$$\begin{aligned} lift_{[\]}^{\perp}(R) &= \{(\perp, \perp)\} \cup \{([\], [\])\} \\ &\cup \left\{ (x : xs, y : ys) \mid (x, y) \in \llbracket \alpha \rrbracket_{\eta'}^{(fix, =)} \wedge (xs, ys) \in \llbracket \beta \rrbracket_{\eta'}^{(fix, =)} \right\} \\ &= \{(\perp, \perp), ([\], [\])\} \cup \{(x : xs, y : ys) \mid (x, y) \in R \wedge (xs, ys) \in lift_{[\]}^{\perp}(R)\} \end{aligned}$$

The derived set notation may be read as follows. Two lists are related if all elements at corresponding positions are related. Additionally, related lists must have the same length if they are finite or partial or they must be both infinite. Note that infinite lists are only covered because $lift_{[\]}^{\perp}(R)$ is required to be the largest relation.

Observe that, if R is the graph of a Haskell function f , the relation $lift_{[\]}^{\perp}(R)$ equals the graph of the function $(map\ f)$ (see Figure 2.4 on page 13 for the definition of map). In Chapter 4, this observation helps to simplify free theorems.

Fixing $R = id_{Int}^{\perp}$ in the relation $lift_{[\]}^{\perp}(R)$ corresponds to the logical relation of the type expression $[Int]$. Then, the pair $(1 : 2 : [\], 1 : 2 : [\])$ of finite lists is related as well as the pair $(3 : \perp, 3 : \perp)$ of partial lists. Moreover, even the pair $(1 : ones, ones)$ of infinite lists is related where $ones$ is the Haskell function given in Section 2.1 on page 14. \square

The definition of the relational actions of n -ary type constructors, where $n \in \mathbb{N}$, is well-defined because permissible type constructors cannot be declared mutually recursively. The following example demonstrates this by means of two permissible type constructors.

Example 10. Consider the following declarations where $\alpha \in V$ is a type variable.

$$\begin{aligned} \mathbf{data}\ M\ \alpha &= E \mid S\ \alpha \\ \mathbf{newtype}\ N\ \alpha &= D\ (M\ \alpha) \end{aligned}$$

3 Parametricity

Let $R \in Rel^{(fix, \sqsubseteq)}$ be a relation, let η be an environment such that $\eta(\alpha) = R$ and let $\beta \in V \setminus \{\alpha\}$ be a type variable distinct to α . Then, the structural lifting of M in the (fix, \sqsubseteq) model is as follows.

$$\begin{aligned} lift_M^\perp(R) &= \{(\perp, \perp)\} \cup \{(E, E)\} \cup \left\{ (S\ x, S\ y) \mid (x, y) \in \llbracket \alpha \rrbracket_{\eta[lift_M^\perp(R)/\beta]}^{(fix, \sqsubseteq)} \right\} \\ &= \{(\perp, \perp), (E, E)\} \cup \{(S\ x, S\ y) \mid (x, y) \in R\} \end{aligned}$$

Taking a relation R , an environment η and a type variable β as before, the structural lifting of N is the following relation.

$$\begin{aligned} lift_N^\perp(R) &= \left\{ (D\ x, D\ y) \mid (x, y) \in \llbracket M\ \alpha \rrbracket_{\eta[lift_N^\perp(R)/\beta]}^{(fix, \sqsubseteq)} \right\} \\ &= \{(D\ x, D\ y) \mid (x, y) \in \sqsubseteq; lift_M^\perp(R)\} \quad \square \end{aligned}$$

Since permissible type classes are not mutually depending and their type class hierarchy is acyclic, the predicates used in the relational action of type abstractions are well-defined (see especially Definition 21 on page 34). The following example demonstrates this by means of three permissible type classes.

Example 11. Consider the following declarations of three permissibly type classes B , C and D .

```

class B  $\alpha$ 
class C  $\alpha$  where  $f :: \forall_{\{B\}} \beta. \beta \rightarrow \beta$ 
class B  $\alpha \Rightarrow D\ \alpha$ 

```

Let $m = (basic, =)$, let $\tau_1, \tau_2 \in T_c$ and let $R \in Rel^m(\tau_1, \tau_2)$ be a relation. If τ_1 and τ_2 are instances of B , then the predicate $P(m, R, B)$ holds. Let $f' :: \forall_{\{C\}} \alpha. \forall_{\{B\}} \beta. \beta \rightarrow \beta$ be the constrained version of the class method f of C . The predicate $P(m, R, C)$ holds, if τ_1 and τ_2 are instances of C and if $(f'_{\tau_1}, f'_{\tau_2})$ is an element of $\llbracket \forall_{\{B\}} \beta. \beta \rightarrow \beta \rrbracket_{\emptyset[R/\alpha]}^m$. Note that the latter relies on the predicate $P(m, R, B)$. Finally, if τ_1 and τ_2 are instances of D , then τ_1 and τ_2 are also instances of B and $P(m, R, B)$ is true; thus also $P(m, R, D)$ holds. \square

The last example of this section shows how to derive a relation for a type expression step-by-step.

Example 12. Consider the type expression $\tau = (\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha)$. The logical relation obtained from τ in the $(seq, =)$ model and using an empty environment is then as follows.

$$\begin{aligned}
 & \llbracket \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \rrbracket_{\emptyset}^{(seq,=)} \\
 = & \forall_{\emptyset} R_1 \in Rel^{(seq,=)}. \llbracket \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \rrbracket_{\emptyset[R_1/\alpha]}^{(seq,=)} \\
 = & \forall_{\emptyset} R_1 \in Rel^{(seq,=)}. \forall_{\emptyset} R_2 \in Rel^{(seq,=)}. \llbracket \alpha \rightarrow (\beta \rightarrow \alpha) \rrbracket_{\emptyset[R_1/\alpha, R_2/\beta]}^{(seq,=)} \\
 = & \forall_{\emptyset} R_1, R_2 \in Rel^{(seq,=)}. \llbracket \alpha \rrbracket_{\emptyset[R_1/\alpha, R_2/\beta]}^{(seq,=)} \rightarrow^{(seq,=)} \llbracket \beta \rightarrow \alpha \rrbracket_{\emptyset[R_1/\alpha, R_2/\beta]}^{(seq,=)} \\
 = & \forall_{\emptyset} R_1, R_2 \in Rel^{(seq,=)}. \llbracket \alpha \rrbracket_{\emptyset[R_1/\alpha, R_2/\beta]}^{(seq,=)} \rightarrow^{(seq,=)} \left(\llbracket \beta \rrbracket_{\emptyset[R_1/\alpha, R_2/\beta]}^{(seq,=)} \rightarrow^{(seq,=)} \llbracket \alpha \rrbracket_{\emptyset[R_1/\alpha, R_2/\beta]}^{(seq,=)} \right) \\
 = & \forall_{\emptyset} R_1 \in Rel^{(seq,=)}. \forall_{\emptyset} R_2 \in Rel^{(seq,=)}. R_1 \rightarrow^{(seq,=)} \left(R_2 \rightarrow^{(seq,=)} R_1 \right) \quad \square
 \end{aligned}$$

3.7 Parametricity theorems

From each of the previously described logical relations, a *parametricity theorem* [Wad89, JV06], which is called abstraction theorem in [Rey83], can be derived. To simplify matters, they are simultaneously given here as follows.

Theorem 1. For every model $m \in M$, for every closed type expression $\tau \in T_c$ and for every closed term $t :: \tau$ such that both τ and t respect the restrictions of m (see the beginning of this chapter), the following property holds.

$$(t, t) \in \llbracket \tau \rrbracket_{\emptyset}^m \quad \square$$

The remainder of this section discusses Theorem 1 and gives reasons for the requirements on relations made in Definition 14 on page 30.

In the $(fix, =)$ model, every relation has to be admissible. Neither strictness nor continuity may be dropped as is shown by the following example taken from [Wad89, Böh06].

Example 13. The parametricity property of the function fix (see Figure 2.4 on page 13) is as follows.

$$\begin{aligned}
 & (fix, fix) \in \llbracket \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rrbracket_{\emptyset}^{(fix,=)} \\
 \Leftrightarrow & (fix, fix) \in (\forall R \in Rel^{(fix,=)}. (R \rightarrow R) \rightarrow R) \\
 \Leftrightarrow & \forall \tau_1, \tau_2 \in T_c, R \in Rel^{(fix,=)}(\tau_1, \tau_2), f :: \tau_1 \rightarrow \tau_1, g :: \tau_2 \rightarrow \tau_2. \\
 & (\forall (x, y) \in R. (f x, g y) \in R) \Rightarrow (fix_{\tau_1} f, fix_{\tau_2} g) \in R
 \end{aligned}$$

Strictness is then required as can be seen by indirect reasoning. Fix $\tau_1 = Bool$ and $\tau_2 = Bool$. Furthermore, let $R = \{(x, True) \mid x \in \langle Bool \rangle^{\perp}\}$. Thus R is continuous, but not strict. Let the two functions be defined as $f x = False$ and $g x = x$ for every $x \in \langle Bool \rangle^{\perp}$. Then, the condition $(f x, g y) \in R$ holds for every $(x, y) \in R$. However, the pair $(fix_{Bool} f, fix_{Bool} g)$ equals $(False, \perp)$ which is not in R .

3 Parametricity

A similar indirect argumentation can be given for continuity. Fix $\tau_1 = [Int]$ and $\tau_2 = Bool$. Let $R = \{(x, \perp) \mid \text{partialOrFinite}(x)\}$ where the predicate $\text{partialOrFinite}(x)$ is true if x is a partial or finite list. Thus, R is strict, but not continuous. Furthermore, let the two functions be defined as $f\ x = 1 : x$ and $g\ y = y$ for every $x \in \langle [Int] \rangle^\perp$ and for every $y \in \langle Bool \rangle^\perp$. Then, for every pair $(x, y) \in R$, the condition $(f\ x, g\ y) \in R$ holds, but $(\text{fix}_{[Int]} f)$ is an infinite list and therefore $(\text{fix}_{[Int]} f, \text{fix}_{Bool} g)$ is not in R . \square

The $(\text{fix}, \sqsubseteq)$ model can be considered as a modification of the $(\text{fix}, =)$ model allowing inequational parametricity results. At first sight, adding left-closedness as an extra condition seems to weaken the parametricity results, but it turns out that this approach allows to derive even more results than in the equational case. This is demonstrated in detail in the Chapter 4.

Example 14. Reconsider the function fix . Its parametricity result in the $(\text{fix}, \sqsubseteq)$ model is as follows.

$$\begin{aligned} & (\text{fix}, \text{fix}) \in \llbracket \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rrbracket_\emptyset^{(\text{fix}, \sqsubseteq)} \\ \Leftrightarrow & (\text{fix}, \text{fix}) \in (\forall R \in \text{Rel}^{(\text{fix}, \sqsubseteq)}. (R \rightarrow R) \rightarrow R) \\ \Leftrightarrow & \forall \tau_1, \tau_2 \in T_c, R \in \text{Rel}^{(\text{fix}, \sqsubseteq)}(\tau_1, \tau_2), f :: \tau_1 \rightarrow \tau_1, g :: \tau_2 \rightarrow \tau_2. \\ & ((\forall (x, y) \in R. (f\ x, g\ y) \in R) \Rightarrow (\text{fix}_{\tau_1} f, \text{fix}_{\tau_2} g) \in R) \end{aligned}$$

Note that this result resembles the one obtained in the $(\text{fix}, =)$ model, except that here, the relation R is under stronger restrictions. Therefore and according to the preceding example, admissibility is a necessary condition on all relations in the $(\text{fix}, \sqsubseteq)$. \square

The $(\text{seq}, =)$ model extends the $(\text{fix}, =)$ model by allowing the use of the selective strictness primitive seq in any term. Since this primitive allows to distinguish $(\backslash x \rightarrow \perp)$ and \perp , bottom-reflectiveness is required additionally to admissibility.

Finally, the $(\text{seq}, \sqsubseteq)$ model may be considered as a combination of the $(\text{seq}, =)$ model and the $(\text{fix}, \sqsubseteq)$ model. Every relation needs to be admissible, and, for the same reasons as in the $(\text{fix}, \sqsubseteq)$ model, left-closedness is introduced. Bottom-reflectiveness as required in the $(\text{seq}, =)$ model, however, can be relaxed to totality. See [JV06] for detailed explanations.

4 Free theorems

This chapter gives detailed examples for free theorems in every of the five models. In particular, it demonstrates how free theorems can be obtained from closed type expressions by applying Theorem 1 on page 39 and the definitions of the preceding chapter. Along with the examples, simplifications are identified and generalised.

4.1 The basic model

Starting from a small example in the $(basic, =)$ model, this section exposes the fundamental steps and simplifications in deriving free theorems. It turns out that the described simplifications can be generalised to the other equational models, one of them even to the inequational models.

Example 15. Consider the function $id :: \forall \alpha. \alpha \rightarrow \alpha$, which is presented in Figure 2.4 on page 13. The parametricity property of id in the $(basic, =)$ model may be obtained as follows.

$$\begin{aligned} & (id, id) \in \llbracket \forall \alpha. \alpha \rightarrow \alpha \rrbracket_{\emptyset}^{(basic, =)} \\ \Leftrightarrow & (id, id) \in (\forall_{\emptyset} R \in Rel^{(basic, =)}. R \rightarrow^{(basic, =)} R) \\ \Leftrightarrow & \forall \tau_1, \tau_2 \in T_c, R \in Rel^{(basic, =)}(\tau_1, \tau_2), (x, y) \in R. (id_{\tau_1} x, id_{\tau_2} y) \in R \end{aligned}$$

For any two closed type expressions $\tau_1, \tau_2 \in T_c$, a relation $R \in Rel^{(basic, =)}(\tau_1, \tau_2)$ may be specialised to a Haskell function $h :: \tau_1 \rightarrow \tau_2$ which respects the restrictions of the $(basic, =)$ model. Then, the following weaker result is implied.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, x :: \tau_1, y :: \tau_2. (x, y) \in h \Rightarrow (id_{\tau_1} x, id_{\tau_2} y) \in h$$

Using the common notation for functions, this can be rewritten to the following result.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, x :: \tau_1, y :: \tau_2. (h x = y) \Rightarrow (h (id_{\tau_1} x) = id_{\tau_2} y)$$

Finally, by replacing y with $(h x)$ in the consequence of the implication, the following free theorem is obtained.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, x :: \tau_1. (h (id_{\tau_1} x) = id_{\tau_2} (h x))$$

Equivalent to this is the following free theorem in point-free notation, which can also be found in [Wad89].

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2. (h \circ id_{\tau_1} = id_{\tau_2} \circ h)$$

4 Free theorems

Note that \circ stands for the Haskell function composition operator. Its definition is given in Figure 2.4 on page 13. \square

On its way to the last free theorem, this example showed three different simplifications, each of which is now reconsidered and generalised.

Simplification 1. Let $\tau_1, \tau_2 \in T_c$ be two closed type expressions, and, for every relation $R \in Rel$, let $P(R)$ be a predicate relying on R . Then, the following implications specify the specialisation of a relation R to a Haskell function h .

$$\begin{aligned}
\forall R \in Rel^{(basic,=)}(\tau_1, \tau_2). P(R) &\implies \forall h :: \tau_1 \rightarrow \tau_2. P(h) \\
\forall R \in Rel^{(fix,=)}(\tau_1, \tau_2). P(R) &\implies \forall h :: \tau_1 \rightarrow \tau_2, h \text{ strict}. P(h) \\
\forall R \in Rel^{(fix,\sqsubseteq)}(\tau_1, \tau_2). P(R) &\implies \forall h :: \tau_1 \rightarrow \tau_2, h \text{ strict}. P(h; \sqsubseteq) \\
\forall R \in Rel^{(fix,\sqsupset)}(\tau_1, \tau_2). P(R) &\implies \forall h :: \tau_2 \rightarrow \tau_1. P(\sqsubseteq; h^{-1}) \\
\forall R \in Rel^{(seq,=)}(\tau_1, \tau_2). P(R) &\implies \forall h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total}. P(h) \\
\forall R \in Rel^{(seq,\sqsubseteq)}(\tau_1, \tau_2). P(R) &\implies \forall h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total}. P(h; \sqsubseteq) \\
\forall R \in Rel^{(seq,\sqsupset)}(\tau_1, \tau_2). P(R) &\implies \forall h :: \tau_2 \rightarrow \tau_1, h \text{ strict}. P(\sqsubseteq; h^{-1}) \quad \square
\end{aligned}$$

Justifying that these implications hold amounts to showing that every Haskell function h or its composition with \sqsubseteq , together with the restrictions on the right-hand side, fulfills the conditions of the corresponding relation R on the left-hand side. Note that every Haskell function h is continuous. Thus, if h is strict, the graph of h is admissible. If h is additionally total, then its graph is bottom-reflecting. The argumentation for the remaining implications is as follows. Let $\tau_1, \tau_2 \in T_c$ be two closed type expressions and let $h :: \tau_1 \rightarrow \tau_2$ be a Haskell function. The relation $\sqsubseteq; h^{-1} = \{(x, y) \mid x \sqsubseteq h y\}$ is then admissible and left-closed, and, if h is strict, it is additionally total. If h is strict, then the relation $h; \sqsubseteq = \{(x, y) \mid h x \sqsubseteq y\}$ is admissible and left-closed. It is additionally total, if h is total. Note that the reasoning for the mentioned properties relies on h being continuous and monotonic.

Simplification 2. Let f and g be two closed terms, let $R \in Rel$ be a relation and let $h :: \tau \rightarrow \tau'$ be a Haskell function, where $\tau, \tau' \in T_c$ are closed type expressions. The following equivalences are given for the $(basic, =)$ model, the $(fix, =)$ model and the $(seq, =)$ model, in that order.

$$\begin{aligned}
(f, g) \in (h \rightarrow^{(basic,=)} R) &\iff \forall x :: \tau. (f x, g (h x)) \in R \\
(f, g) \in (h \rightarrow R) &\iff \forall x :: \tau. (f x, g (h x)) \in R \\
(f, g) \in (h \rightarrow^{(seq,=)} R) &\iff (f \neq \perp \iff g \neq \perp) \wedge (\forall x :: \tau. (f x, g (h x)) \in R)
\end{aligned}$$

A special case for h is the identity function, whose graph may be considered to be either the relation id_τ or the relation id_τ^\perp for a closed type expression $\tau \in T_c$. In that case, the term $(h x)$ on the right-hand side can be further simplified to just x . \square

These equivalences can be shown by applying Definition 17 on page 33, Definition 18 on page 34 or Definition 19 on page 34, depending on the model, and using the idea shown in Example 15 on page 41. Based on the same definitions are also most of the following equivalences, while some of them only establish common notation.

Simplification 3. Let $\tau, \tau', \tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$ be closed type expressions, let x and y be closed terms and let $f :: \tau_1 \rightarrow \tau'_1$ and $g :: \tau_2 \rightarrow \tau'_2$ be Haskell functions. The following equivalences are given in the $(seq, =)$ model, but they can easily be adjusted to the $(basic, =)$ and the $(fix, =)$ model by omitting the conditions $(x \neq \perp)$ and $(y \neq \perp)$ on the right-hand sides.

$$\begin{aligned}
(x, y) \in f &\iff f x = y \\
(x, y) \in id_{\tau}^{\perp} &\iff x = y \\
(x, y) \in (f \rightarrow^{(seq, =)} g) &\iff (x \neq \perp \iff y \neq \perp) \wedge (g \circ x = y \circ f) \\
(x, y) \in (id_{\tau}^{\perp} \rightarrow^{(seq, =)} g) &\iff (x \neq \perp) \wedge (g \circ x = y) \\
(x, y) \in (f \rightarrow^{(seq, =)} id_{\tau}^{\perp}) &\iff (y \neq \perp) \wedge (x = y \circ f) \\
(x, y) \in (id_{\tau}^{\perp} \rightarrow^{(seq, =)} id_{\tau'}^{\perp}) &\iff x = y
\end{aligned}$$

Note that the last three equivalences may lead to further simplifications similar to those given in Simplification 2 on the preceding page. \square

Since, in the third to last equivalence, y equals $g \circ x$ and $g \circ x$ is always different to \perp , the restriction $(y \neq \perp)$ was dropped. A similar argumentation applies to the next to last equivalence. Note that the left-hand side of the last equivalence corresponds to $(x, y) \in id_{\tau \rightarrow \tau'}^{\perp}$, and, with the second equivalence, this may be reduced to $x = y$.

4.2 The *fix* models

This section shows free theorems for the Haskell function *length* (see Figure 2.4 on page 13) both in the $(fix, =)$ and in the (fix, \sqsubseteq) model. Similarly to the preceding section, simplifications identified in the examples are generalised to other suitable models.

Example 16. The parametricity property of *length* in the $(fix, =)$ model is as follows.

$$\begin{aligned}
&(length, length) \in \llbracket \forall \alpha. [\alpha] \rightarrow Int \rrbracket_{\emptyset}^{(fix, =)} \\
&\iff (length, length) \in (\forall_{\emptyset} R \in Rel^{(fix, =)}. lift_{\square}^{\perp}(R) \rightarrow id_{Int}^{\perp}) \\
&\iff \forall \tau_1, \tau_2 \in T_c, R \in Rel^{(fix, =)}(\tau_1, \tau_2). (length, length) \in (lift_{\square}^{\perp}(R) \rightarrow id_{Int}^{\perp}) \\
&\iff \forall \tau_1, \tau_2 \in T_c, R \in Rel^{(fix, =)}(\tau_1, \tau_2), (x, y) \in lift_{\square}^{\perp}(R). length_{\tau_1} x = length_{\tau_2} y
\end{aligned}$$

Note that Simplification 3 was already applied in the last line to get the common notation of equality. Instead of simplifying the last line, the next steps concentrate on the next to last one. With Simplification 1, the following weaker result is obtained.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict}. (length, length) \in (lift_{\square}^{\perp}(h) \rightarrow id_{Int}^{\perp})$$

4 Free theorems

In Example 9 on page 37, it was already observed that $\text{lift}_{\square}^{\perp}(h)$ equals the graph of $(\text{map } h)$. Thus, since $(\text{map } h)$ is a Haskell function, Simplification 3 on the preceding page can be applied to yield the following free theorem in point-free notation.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict} . \text{length}_{\tau_1} = \text{length}_{\tau_2} \circ (\text{map}_{\tau_1 \tau_2} h) \quad \square$$

The observation of Example 9 on page 37 can be generalised to other models as well. In the $(\text{basic}, =)$ and $(\text{seq}, =)$ model, the argumentation is similar to that of Example 9. Justifications for the $(\text{seq}, \sqsubseteq)$ model are in [JV06], which apply in the same way for the $(\text{fix}, \sqsubseteq)$ model.

Simplification 4. Let $\tau_1, \tau_2 \in T_c$ be two closed type expressions and let $f :: \tau_1 \rightarrow \tau_2$ be a Haskell function. Then, the following equalities and set inclusions hold.

$$\begin{aligned} \text{lift}_{\square}(f) f &= \text{map}_{\tau_1 \tau_2} \\ \text{lift}_{\square}^{\perp}(f) f &= \text{map}_{\tau_1 \tau_2} \\ \sqsubseteq; \text{lift}_{\square}^{\perp}(f; \sqsubseteq) &\subseteq (\text{map}_{\tau_1 \tau_2} f); \sqsubseteq \\ \sqsubseteq; \text{lift}_{\square}^{\perp}(\sqsubseteq; f^{-1}) &\subseteq \sqsubseteq; (\text{map}_{\tau_1 \tau_2} f)^{-1} \end{aligned}$$

Additionally, let x and y be two closed terms and, for every two closed terms u and v , let $P(u, v)$ be a predicate depending on u and v . Then, the following implications hold.

$$\begin{aligned} \forall (x, y) \in \sqsubseteq; \text{lift}_{\square}^{\perp}(f; \sqsubseteq) . P(x, y) &\Rightarrow \forall l :: [\tau_1] . P(l, \text{map}_{\tau_1 \tau_2} f l) \\ \forall (x, y) \in \sqsubseteq; \text{lift}_{\square}^{\perp}(\sqsubseteq; f^{-1}) . P(x, y) &\Rightarrow \forall l :: [\tau_1] . P(\text{map}_{\tau_1 \tau_2} f l, l) \quad \square \end{aligned}$$

Note that structural liftings of other permissible type constructors may also be replaced with functions. In particular, Simplification 4 may easily be adjusted to instances of the Haskell type class *Functor*, because the class method *fmap* of *Functor* generalises the idea of *map*. For example, the type constructor *Maybe* (see Figure 2.4 on page 13) is an instance of *Functor*.

The second part of this section is concerned with free theorems in the $(\text{fix}, \sqsubseteq)$ model.

Example 17. Reconsider the function *length* from the preceding example. In the $(\text{fix}, \sqsubseteq)$ model, its parametricity property is as follows.

$$\begin{aligned} (\text{length}, \text{length}) &\in \llbracket \forall \alpha . [\alpha] \rightarrow \text{Int} \rrbracket_{\emptyset}^{(\text{fix}, \sqsubseteq)} \\ \Leftrightarrow (\text{length}, \text{length}) &\in (\forall_{\emptyset} R \in \text{Rel}^{(\text{fix}, \sqsubseteq)} . \sqsubseteq; \text{lift}_{\square}^{\perp}(R) \rightarrow \sqsubseteq) \\ \Leftrightarrow \forall \tau_1, \tau_2 \in T_c, R \in \text{Rel}^{(\text{fix}, \sqsubseteq)}(\tau_1, \tau_2), (x, y) \in \sqsubseteq; \text{lift}_{\square}^{\perp}(R) . &(\text{length}_{\tau_1} x, \text{length}_{\tau_2} y) \in \sqsubseteq \end{aligned}$$

Establishing the common infix notation of partial orders, the following equivalent result is obtained.

$$\forall \tau_1, \tau_2 \in T_c, R \in \text{Rel}^{(\text{fix}, \sqsubseteq)}(\tau_1, \tau_2), (x, y) \in \sqsubseteq; \text{lift}_{\square}^{\perp}(R) . \text{length}_{\tau_1} x \sqsubseteq \text{length}_{\tau_2} y$$

Simplification 1 offers the choice to specialise the relation R to either $h; \sqsubseteq$ or to $\sqsubseteq; h^{-1}$. Consider first the first option, which yields, together with Simplification 4 on the preceding page, the following weaker result.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict}, x :: [\tau_1]. \text{length}_{\tau_1} x \sqsubseteq \text{length}_{\tau_2} (\text{map}_{\tau_1 \tau_2} h x)$$

Similarly to the equational case (see Example 16 on page 43), there is also a point-free notation, which is as follows.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict}. \text{length}_{\tau_1} \sqsubseteq \text{length}_{\tau_2} \circ (\text{map}_{\tau_1 \tau_2} h)$$

Consider now the second option, that is, specialise R to $\sqsubseteq; h^{-1}$. Applying similar simplifications as before, the following free theorem is obtained.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_2 \rightarrow \tau_1, y :: [\tau_2]. \text{length}_{\tau_1} (\text{map}_{\tau_2 \tau_1} h y) \sqsubseteq \text{length}_{\tau_2} y$$

The corresponding point-free notation is as follows.

$$\forall \tau_1, \tau_2 \in T_c, h :: \tau_2 \rightarrow \tau_1. \text{length}_{\tau_1} \circ (\text{map}_{\tau_2 \tau_1} h) \sqsubseteq \text{length}_{\tau_2}$$

Finally, the following conciser notation of the obtained free theorems can be obtained by properly choosing τ_1 , τ_2 and h .

$$\begin{aligned} \text{length}_{\tau_1} &\sqsubseteq \text{length}_{\tau_2} \circ (\text{map}_{\tau_1 \tau_2} h) && (h \text{ strict}) \\ \text{length}_{\tau_1} &\supseteq \text{length}_{\tau_2} \circ (\text{map}_{\tau_1 \tau_2} h) \end{aligned}$$

Note that the combination of these two results equals the free theorem given in Example 16 on page 43. \square

This example uses simplifications which resemble those given in Simplification 3 on page 43, but some of which are implications and not equivalences anymore. As a consequence, not all of the following simplifications can be applied in every situation, as demonstrated later in Section 4.3.

Simplification 5. Let x and y be closed terms, let $\tau, \tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$ be type expressions and let $f :: \tau_1 \rightarrow \tau'_1$ and $g :: \tau_2 \rightarrow \tau'_2$ be Haskell functions. The following equivalences then hold in both the $(\text{fix}, \sqsubseteq)$ and the $(\text{seq}, \sqsubseteq)$ model.

$$\begin{aligned} (x, y) \in \sqsubseteq &\iff x \sqsubseteq y \\ (x, y) \in f; \sqsubseteq &\iff f x \sqsubseteq y \\ (x, y) \in \sqsubseteq; f^{-1} &\iff x \sqsubseteq f y \end{aligned}$$

Although given in the $(\text{fix}, \sqsubseteq)$ model, the following four implications can easily be adjusted to the $(\text{seq}, \sqsubseteq)$ model. In particular, the condition $(x \neq \perp \Rightarrow y \neq \perp)$ may be

added as a conjunct to the conclusions of the implications.

$$\begin{aligned}
 (x, y) \in (f; \sqsubseteq \rightarrow g; \sqsubseteq) &\implies g \circ x \sqsubseteq y \circ f \\
 (x, y) \in (f; \sqsubseteq \rightarrow \sqsubseteq; g^{-1}) &\implies x \sqsubseteq g \circ y \circ f \\
 (x, y) \in (\sqsubseteq; f^{-1} \rightarrow g; \sqsubseteq) &\implies g \circ x \circ f \sqsubseteq y \\
 (x, y) \in (\sqsubseteq; f^{-1} \rightarrow \sqsubseteq; g^{-1}) &\implies x \circ f \sqsubseteq g \circ y
 \end{aligned}$$

Note that these implications get simpler, if f or g or both are identity functions. \square

4.3 The *seq* models

The Haskell function *filter*, whose definition is given in Figure 2.4 on page 13, serves as the running example of this section, in which free theorems for both the (*seq*, =) and the (*seq*, \sqsubseteq) model are derived. In comparison with the two *fix* models, the two *seq* models introduce additional restrictions to parametricity properties (see Definition 19 on page 34 and Definition 20 on page 34). This section describes under which conditions these restrictions can be dropped. In the end, further simplifications specific to the two inequational models are collected.

Example 18. Consider the Haskell function $filter :: \forall \alpha. (\alpha \rightarrow Bool) \rightarrow [\alpha] \rightarrow [\alpha]$. Its parametricity property in the (*seq*, =) model is as follows.

$$\begin{aligned}
 &\forall \tau_1, \tau_2 \in T_c, R \in Rel^{(seq,=)}(\tau_1, \tau_2), f :: \tau_1 \rightarrow Bool, g :: \tau_2 \rightarrow Bool. \\
 &(filter_{\tau_1} \neq \perp \Leftrightarrow filter_{\tau_2} \neq \perp) \\
 &\wedge (((f \neq \perp \Leftrightarrow g \neq \perp) \wedge (\forall (x_1, y_1) \in R. f x_1 = g y_1)) \\
 &\Rightarrow ((filter_{\tau_1} f \neq \perp \Leftrightarrow filter_{\tau_2} g \neq \perp) \\
 &\quad \wedge (\forall (x_2, y_2) \in lift_{\sqsubseteq}^{\perp}(R). (filter_{\tau_1} f x_2, filter_{\tau_2} g y_2) \in lift_{\sqsubseteq}^{\perp}(R))))
 \end{aligned}$$

With Simplification 1 on page 42, the relation R can be specialised to a function h giving the following weaker result.

$$\begin{aligned}
 &\forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total}, f :: \tau_1 \rightarrow Bool, g :: \tau_2 \rightarrow Bool. \\
 &(filter_{\tau_1} \neq \perp \Leftrightarrow filter_{\tau_2} \neq \perp) \\
 &\wedge (((f \neq \perp \Leftrightarrow g \neq \perp) \wedge (\forall (x_1, y_1) \in h. f x_1 = g y_1)) \\
 &\Rightarrow ((filter_{\tau_1} f \neq \perp \Leftrightarrow filter_{\tau_2} g \neq \perp) \\
 &\quad \wedge (\forall (x_2, y_2) \in lift_{\sqsubseteq}^{\perp}(h). (filter_{\tau_1} f x_2, filter_{\tau_2} g y_2) \in lift_{\sqsubseteq}^{\perp}(h))))
 \end{aligned}$$

The predicate $((f \neq \perp \Leftrightarrow g \neq \perp) \wedge (\forall (x_1, y_1) \in h. f x_1 = g y_1))$ in the third line is equivalent to $(f, g) \in (h \rightarrow^{(seq,=)} id_{Bool}^{\perp})$. This can be rewritten equivalently with Simplification 3 on page 43, and thus, the preceding result is equivalent to the following

one.

$$\begin{aligned}
& \forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total, } f :: \tau_1 \rightarrow \text{Bool}, g :: \tau_2 \rightarrow \text{Bool}. \\
& (\text{filter}_{\tau_1} \neq \perp \Leftrightarrow \text{filter}_{\tau_2} \neq \perp) \\
& \wedge (((g \neq \perp) \wedge (f = g \circ h)) \\
& \Rightarrow ((\text{filter}_{\tau_1} f \neq \perp \Leftrightarrow \text{filter}_{\tau_2} g \neq \perp) \\
& \wedge (\forall (x_2, y_2) \in \text{lift}_{\square}^{\perp}(h). (\text{filter}_{\tau_1} f x_2, \text{filter}_{\tau_2} g y_2) \in \text{lift}_{\square}^{\perp}(h))))
\end{aligned}$$

Note that the conjunction in the last two lines of this result is equivalent to

$$(\text{filter } f, \text{filter } g) \in (\text{lift}_{\square}^{\perp}(h) \xrightarrow{(\text{seq}, =)} \text{lift}_{\square}^{\perp}(h))$$

After applying Simplification 4 on page 44, Simplification 3 on page 43 can be used again to obtain the following equivalent result.

$$\begin{aligned}
& \forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total, } f :: \tau_1 \rightarrow \text{Bool}, g :: \tau_2 \rightarrow \text{Bool}. \\
& (\text{filter}_{\tau_1} \neq \perp \Leftrightarrow \text{filter}_{\tau_2} \neq \perp) \\
& \wedge (((g \neq \perp) \wedge (f = g \circ h)) \\
& \Rightarrow ((\text{filter}_{\tau_1} f \neq \perp \Leftrightarrow \text{filter}_{\tau_2} g \neq \perp) \\
& \wedge ((\text{map}_{\tau_1 \tau_2} h) \circ (\text{filter}_{\tau_1} f) = (\text{filter}_{\tau_2} g) \circ (\text{map}_{\tau_1 \tau_2} h))))
\end{aligned}$$

The restriction for bottom-reflectiveness of *filter* and of *filter* applied to one argument can be dropped yielding the following weaker parametricity result.

$$\begin{aligned}
& \forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total, } f :: \tau_1 \rightarrow \text{Bool}, g :: \tau_2 \rightarrow \text{Bool}. \\
& ((g \neq \perp) \wedge (f = g \circ h)) \Rightarrow ((\text{map}_{\tau_1 \tau_2} h) \circ (\text{filter}_{\tau_1} f) = (\text{filter}_{\tau_2} g) \circ (\text{map}_{\tau_1 \tau_2} h))
\end{aligned}$$

Since *f* equals $(g \circ h)$, every occurrence of *f* can be replaced by $g \circ h$, which results in the following free theorem.

$$\begin{aligned}
& \forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total, } g :: \tau_2 \rightarrow \text{Bool}. \\
& (g \neq \perp) \Rightarrow ((\text{map}_{\tau_1 \tau_2} h) \circ (\text{filter}_{\tau_1} (g \circ h)) = (\text{filter}_{\tau_2} g) \circ (\text{map}_{\tau_1 \tau_2} h)) \quad \square
\end{aligned}$$

Dropping the bottom-reflectiveness restrictions as shown in this example is an aspect of logical deductions, a more general notation of which is given as follows.

Simplification 6. Let *P*, *P'*, *Q* and *Q'* be predicates. In conjunctions, any conjunct can be dropped resulting in a weaker statement.

$$\begin{aligned}
P \wedge Q & \Longrightarrow P \\
P \wedge Q & \Longrightarrow Q
\end{aligned}$$

Strengthening the premise of an implication or weakening the conclusion of an implication results in a weaker implication.

$$\begin{aligned}
(P \Rightarrow Q) & \Longrightarrow (P' \Rightarrow Q) & (\text{if } P' \Rightarrow P) \\
(P \Rightarrow Q) & \Longrightarrow (P \Rightarrow Q') & (\text{if } Q \Rightarrow Q')
\end{aligned} \quad \square$$

Example 19. Now, consider the function *filter* in the (seq, \sqsubseteq) model. Its parametricity result is as follows.

$$\begin{aligned} & \forall \tau_1, \tau_2 \in T_c, R \in Rel^{(seq, \sqsubseteq)}(\tau_1, \tau_2), f :: \tau_1 \rightarrow Bool, g :: \tau_2 \rightarrow Bool. \\ & (filter_{\tau_1} \neq \perp \Rightarrow filter_{\tau_2} \neq \perp) \\ & \wedge (((f \neq \perp \Rightarrow g \neq \perp) \wedge (\forall (x_1, y_1) \in R. f x_1 \sqsubseteq g y_1)) \\ & \Rightarrow ((filter_{\tau_1} f \neq \perp \Rightarrow filter_{\tau_2} g \neq \perp) \\ & \wedge (\forall (x_2, y_2) \in \sqsubseteq; lift_{\sqsubseteq}^{\perp}(R). (filter_{\tau_1} f x_2, filter_{\tau_2} g y_2) \in \sqsubseteq; lift_{\sqsubseteq}^{\perp}(R)))) \end{aligned}$$

In a similar way as in Example 18 on page 46, the totality restrictions for *filter* and for *filter* applied to a function may be dropped, which gives the following weaker result.

$$\begin{aligned} & \forall \tau_1, \tau_2 \in T_c, R \in Rel^{(seq, \sqsubseteq)}(\tau_1, \tau_2), f :: \tau_1 \rightarrow Bool, g :: \tau_2 \rightarrow Bool. \\ & ((f \neq \perp \Rightarrow g \neq \perp) \wedge (\forall (x_1, y_1) \in R. f x_1 \sqsubseteq g y_1)) \\ & \Rightarrow (\forall (x_2, y_2) \in \sqsubseteq; lift_{\sqsubseteq}^{\perp}(R). (filter_{\tau_1} f x_2, filter_{\tau_2} g y_2) \in \sqsubseteq; lift_{\sqsubseteq}^{\perp}(R)) \end{aligned}$$

According to Simplification 1 on page 42, the relation R can be specialised in two ways. Consider first the specialisation to $h; \sqsubseteq$.

$$\begin{aligned} & \forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total}, f :: \tau_1 \rightarrow Bool, g :: \tau_2 \rightarrow Bool. \\ & ((f \neq \perp \Rightarrow g \neq \perp) \wedge (\forall x_1 :: \tau_1, y_1 :: \tau_2. (h x_1 \sqsubseteq y_1) \Rightarrow (f x_1 \sqsubseteq g y_1))) \\ & \Rightarrow (\forall (x_2, y_2) \in \sqsubseteq; lift_{\sqsubseteq}^{\perp}(h; \sqsubseteq). (filter_{\tau_1} f x_2, filter_{\tau_2} g y_2) \in \sqsubseteq; lift_{\sqsubseteq}^{\perp}(h; \sqsubseteq)) \end{aligned}$$

With Simplification 4 on page 44 and Simplification 5 on page 45, the following weaker result is obtained.

$$\begin{aligned} & \forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total}, f :: \tau_1 \rightarrow Bool, g :: \tau_2 \rightarrow Bool. \\ & ((f \neq \perp \Rightarrow g \neq \perp) \wedge (\forall x_1 :: \tau_1, y_1 :: \tau_2. (h x_1 \sqsubseteq y_1) \Rightarrow (f x_1 \sqsubseteq g y_1))) \\ & \Rightarrow ((map_{\tau_1 \tau_2} h) \circ (filter_{\tau_1} f) \sqsubseteq (filter_{\tau_2} g) \circ (map_{\tau_1 \tau_2} h)) \end{aligned}$$

Now, due to the restrictions given in Simplification 6 on the preceding page, Simplification 5 on page 45 is not applicable to simplify the premise of the implication. Instead, the function f may be specialised to $(g \circ h)$. Then, the right conjunct of the premise expresses precisely the monotonicity of g . Since this is always true, it may be omitted. Moreover, since f is now equal to $(g \circ h)$ and $(g \circ h)$ is different to \perp , the left conjunct may be simplified to $(g \neq \perp)$. Thus, the following free theorem is obtained.

$$\begin{aligned} & \forall \tau_1, \tau_2 \in T_c, h :: \tau_1 \rightarrow \tau_2, h \text{ strict and total}, g :: \tau_2 \rightarrow Bool. \\ & (g \neq \perp) \Rightarrow ((map_{\tau_1 \tau_2} h) \circ (filter_{\tau_1} (g \circ h)) \sqsubseteq (filter_{\tau_2} g) \circ (map_{\tau_1 \tau_2} h)) \end{aligned}$$

Simplification 1 on page 42 also allows R to be specialised to $\sqsubseteq; h^{-1}$. Together with Simplification 4 on page 44 and Simplification 5 on page 45, this yields the following

result.

$$\begin{aligned} & \forall \tau_1, \tau_2 \in T_c, h :: \tau_2 \rightarrow \tau_1, h \text{ strict}, f :: \tau_1 \rightarrow \text{Bool}, g :: \tau_2 \rightarrow \text{Bool}. \\ & ((f \neq \perp \Rightarrow g \neq \perp) \wedge (\forall x_1 :: \tau_1, y_1 :: \tau_2. (x_1 \sqsubseteq h y_1) \Rightarrow (f x_1 \sqsubseteq g y_1))) \\ & \Rightarrow ((\text{filter}_{\tau_1} f) \circ (\text{map}_{\tau_2 \tau_1} h) \sqsubseteq (\text{map}_{\tau_2 \tau_1} h) \circ (\text{filter}_{\tau_2} g)) \end{aligned}$$

Again, the same idea as in the other case can be applied. This time, the function g is specialised ($f \circ h$). Additionally requiring $(g \neq \perp)$ instead of $(f \neq \perp \Rightarrow g \neq \perp)$ strengthens the premise of the main implication. Thus, with Simplification 6 on page 47 and using the fact that $(f \circ h)$ is different to \perp , the following free theorem is obtained.

$$\begin{aligned} & \forall \tau_1, \tau_2 \in T_c, h :: \tau_2 \rightarrow \tau_1, h \text{ strict}, f :: \tau_1 \rightarrow \text{Bool}. \\ & (\text{filter}_{\tau_1} f) \circ (\text{map}_{\tau_2 \tau_1} h) \sqsubseteq (\text{map}_{\tau_2 \tau_1} h) \circ (\text{filter}_{\tau_2} (f \circ h)) \quad \square \end{aligned}$$

This example demonstrated, how proper specialisations may lead to simplifications based on the definition of monotonicity. The generalisation of this idea is as follows. Let $\tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$ be closed type expressions. For every Haskell function $h_1 :: \tau_1 \rightarrow \tau'_1$ and for every Haskell function $h_2 :: \tau_2 \rightarrow \tau'_2$, let $P(h_1, h_2)$ be a predicate depending on h_1 and h_2 . Furthermore, let $F \in \text{Rel}^{(\text{seq}, \sqsubseteq)}(\tau_1, \tau_2)$ and $G \in \text{Rel}^{(\text{seq}, \sqsubseteq)}(\tau'_1, \tau'_2)$ be two relations. Depending on these two relations, the following table gives simplifications possible in the $(\text{seq}, \sqsubseteq)$ model.

F	G	$\forall h_1 :: \tau_1 \rightarrow \tau'_1, h_2 :: \tau_2 \rightarrow \tau'_2. (h_1, h_2) \in (F \rightarrow^{(\text{seq}, \sqsubseteq)} G) \Rightarrow P(h_1, h_2)$
$f; \sqsubseteq$	$g; \sqsubseteq$	$\forall h :: \tau_2 \rightarrow \tau'_1. (h \circ f \neq \perp \Rightarrow g \circ h \neq \perp) \Rightarrow P(h \circ f, g \circ h)$
$f; \sqsubseteq$	$\sqsubseteq; g^{-1}$	$\forall h :: \tau_2 \rightarrow \tau'_2. (g \circ h \circ f \neq \perp \Rightarrow h \neq \perp) \Rightarrow P(g \circ h \circ f, h)$
$\sqsubseteq; f^{-1}$	$g; \sqsubseteq$	$\forall h :: \tau_1 \rightarrow \tau'_1. (h \neq \perp \Rightarrow g \circ h \circ f \neq \perp) \Rightarrow P(h, g \circ h \circ f)$
$\sqsubseteq; f^{-1}$	$\sqsubseteq; g^{-1}$	$\forall h :: \tau_1 \rightarrow \tau'_2. (g \circ h \neq \perp \Rightarrow h \circ f \neq \perp) \Rightarrow P(g \circ h, h \circ f)$

Note that, unless f or g or both are identity functions, most of the premises are true without further restriction and can be omitted. The only exception is the case that $F = f; \sqsubseteq$ and $G = \sqsubseteq; g^{-1}$, in which premise is equivalent to $(h \neq \perp)$. The following tables sum up these implications.

Simplification 7. Let $\tau, \tau', \tau_1, \tau'_1, \tau_2, \tau'_2 \in T_c$ be closed type expressions. For every Haskell function $h_1 :: \tau_1 \rightarrow \tau'_1$ and for every Haskell function $h_2 :: \tau_2 \rightarrow \tau'_2$, let $P(h_1, h_2)$ be a predicate depending on h_1 and h_2 . Furthermore, let $F \in \text{Rel}^{(\text{seq}, \sqsubseteq)}(\tau_1, \tau_2)$ and $G \in \text{Rel}^{(\text{seq}, \sqsubseteq)}(\tau'_1, \tau'_2)$ be two relations.

F	G	$\forall h_1 :: \tau_1 \rightarrow \tau'_1, h_2 :: \tau_2 \rightarrow \tau'_2. (h_1, h_2) \in (F \rightarrow^{(\text{seq}, \sqsubseteq)} G) \Rightarrow P(h_1, h_2)$
$f; \sqsubseteq$	$g; \sqsubseteq$	$\forall h :: \tau_2 \rightarrow \tau'_1. P(h \circ f, g \circ h)$
$f; \sqsubseteq$	$\sqsubseteq; g^{-1}$	$\forall h :: \tau_2 \rightarrow \tau'_2. (h \neq \perp) \Rightarrow P(g \circ h \circ f, h)$
$\sqsubseteq; f^{-1}$	$g; \sqsubseteq$	$\forall h :: \tau_1 \rightarrow \tau'_1. P(h, g \circ h \circ f)$
$\sqsubseteq; f^{-1}$	$\sqsubseteq; g^{-1}$	$\forall h :: \tau_1 \rightarrow \tau'_2. P(g \circ h, h \circ f)$

F	G	$\forall h_1 :: \tau_1 \rightarrow \tau'_1, h_2 :: \tau_2 \rightarrow \tau'_2. (h_1, h_2) \in (F \rightarrow^{(seq, \sqsubseteq)} G) \Rightarrow P(h_1, h_2)$
\sqsubseteq	$g; \sqsubseteq$	$\forall h :: \tau \rightarrow \tau'_1. P(h, g \circ h)$
\sqsubseteq	$\sqsubseteq; g^{-1}$	$\forall h :: \tau \rightarrow \tau'_2. (h \neq \perp) \Rightarrow P(g \circ h, h)$
$f; \sqsubseteq$	\sqsubseteq	$\forall h :: \tau_1 \rightarrow \tau'. (h \neq \perp) \Rightarrow P(h \circ f, h)$
$\sqsubseteq; f^{-1}$	\sqsubseteq	$\forall h :: \tau_1 \rightarrow \tau'. P(h, h \circ f)$
\sqsubseteq	\sqsubseteq	$\forall h :: \tau \rightarrow \tau'. P(h, h)$

Whenever τ occurs in the second table, it is equal to both τ_1 and τ_2 ; similarly, whenever τ' occurs, it is equal to both τ'_1 and τ'_2 .

Note that these simplifications can easily be adjusted to the (fix, \sqsubseteq) model by dropping the restrictions on h . \square

4.4 Type classes

This section consists solely of one example which demonstrates how type classes constrain parametricity results and free theorems. Especially, the following example illustrates Definition 21 on page 34.

Example 20. Consider the function *elem* given in Figure 2.4 on page 13. Applying the parametricity theorem in the $(fix, =)$ model yields the following result.

$$\begin{aligned}
& (elem, elem) \in \llbracket \forall_{\{Eq\}} \alpha. \alpha \rightarrow [\alpha] \rightarrow Bool \rrbracket_{\emptyset}^{(fix, =)} \\
& \Leftrightarrow (elem, elem) \in \left(\forall_{\{Eq\}} R. R \rightarrow (lift_{\perp}^{\perp}(R) \rightarrow id_{Bool}^{\perp}) \right) \\
& \Leftrightarrow \left(\forall \tau_1, \tau_2 \in T_{\{Eq\}}, R \in Rel^{(fix, =)}(\tau_1, \tau_2) . \right. \\
& \quad P((fix, =), R, \{Eq\}) \Rightarrow \\
& \quad \left. (\forall (x, y) \in R, (xs, ys) \in lift_{\perp}^{\perp}(R). elem_{\tau_1} x xs = elem_{\tau_2} y ys) \right)
\end{aligned}$$

According to Definition 21 on page 34 and Definition 22 on page 35, the predicate $P((fix, =), R, \{Eq\})$ represents the following formula.

$$((=)_{\tau_1}, (=)_{\tau_2}) \in \llbracket \alpha \rightarrow \alpha \rightarrow Bool \rrbracket_{\emptyset[R/\alpha]}^{(fix, =)} \wedge ((/=)_{\tau_1}, (/=)_{\tau_2}) \in \llbracket \alpha \rightarrow \alpha \rightarrow Bool \rrbracket_{\emptyset[R/\alpha]}^{(fix, =)}$$

Since the left and the right conjunct are similar, only the parametricity result for the left conjunct is given here.

$$\begin{aligned}
& \forall (x_1, y_1) \in R, (x_2, y_2) \in R. (=)_{\tau_1} x_1 x_2 = (=)_{\tau_2} y_1 y_2 \\
& \Leftrightarrow (\forall (x_1, y_1), (x_2, y_2) \in R. (x_1 =_{\tau_1} x_2) \Leftrightarrow (y_1 =_{\tau_2} y_2))
\end{aligned}$$

Applying Simplification 1 on page 42, Simplification 2 on page 42, Simplification 3 on page 43 and Simplification 4 on page 44 to the parametricity result of *elem* yields the following free theorem.

$$\begin{aligned}
& \forall \tau_1, \tau_2 \in T_{\{Eq\}}, h :: \tau_1 \rightarrow \tau_2, h \text{ strict} . \\
& \quad P((fix, =), h, \{Eq\}) \Rightarrow \\
& \quad (\forall x :: \tau_1, xs :: [\tau_1]. elem_{\tau_1} x xs = elem_{\tau_2} (h x) (map_{\tau_1 \tau_2} h xs))
\end{aligned}$$

Similar as before, the predicate $P((fix, =), h, \{Eq\})$ consists of two conjuncts, but again only the left one is given here.

$$\forall x_1 :: \tau_1, x_2 :: \tau_1. (x_1 ==_{\tau_1} x_2) \Leftrightarrow (h x_1 ==_{\tau_2} h x_2) \quad \square$$

4.5 Miscellaneous

The last simplification of this chapter originates in the following observation. Consider the $(fix, =)$ model and the closed type expression $[Char]$, which is equivalent to $String$. This type expression is interpreted by the logical relation (Definition 24 on page 35) to $lift_{[]}^{\perp}(id_{Char}^{\perp})$. Let now (x, y) be an element of that relation. Then, as described by the structural lifting of the constructor $[\]$, both x and y have to be infinite lists or, if they are finite or partial, both have to have the same length. Additionally, since the elements of x and y are related by the identity relation for $Char$, elements at equal positions in x and y have to be equal. As a consequence, both x and y have to be equal, and thus, the relation $lift_{[]}^{\perp}(id_{Char}^{\perp})$ is an identity relation. Note that this argumentation may be generalised to the other models, which then leads to the following simplifications.

Simplification 8. Let $n \in \mathbb{N}$, let $\sigma \in C_*^{(n)}$ be a permissible type constructor and let $\tau_1, \dots, \tau_n \in T_c$ be closed type expressions.

$$\begin{aligned} lift_{\sigma}(id_{\tau_1}, \dots, id_{\tau_n}) &= id_{(\sigma \tau_1 \dots \tau_n)} \\ lift_{\sigma}^{\perp}(id_{\tau_1}^{\perp}, \dots, id_{\tau_n}^{\perp}) &= id_{(\sigma \tau_1 \dots \tau_n)}^{\perp} \\ \sqsubseteq; lift_{\sigma}^{\perp}(\sqsubseteq_{\tau_1}, \dots, \sqsubseteq_{\tau_n}) &= \sqsubseteq_{(\sigma \tau_1 \dots \tau_n)} \quad \square \end{aligned}$$

Example 21. Consider the Haskell function $words :: String \rightarrow [String]$, which breaks up a string into a list of words. Before deriving a parametricity result for it, the type synonym $String$ has to be replaced. Therefore, the closed type expression of $words$ is equivalent to $[Char] \rightarrow [[Char]]$. Applying Theorem 1 on page 39 then yields the following result in the $(fix, =)$ model.

$$\forall (x, y) \in lift_{[]}^{\perp}(id_{Char}^{\perp}). (words x, words y) \in lift_{[]}^{\perp}(lift_{[]}^{\perp}(id_{Char}^{\perp}))$$

With Simplification 8, the relation $lift_{[]}^{\perp}(id_{Char}^{\perp})$ is identical to $id_{[Char]}^{\perp}$, and the relation $lift_{[]}^{\perp}(lift_{[]}^{\perp}(id_{Char}^{\perp}))$ is identical to $id_{[[Char]]}^{\perp}$. Therefore, the following result is equivalent to the previous one.

$$\forall (x, y) \in id_{[Char]}^{\perp}. (words x, words y) \in id_{[[Char]]}^{\perp}$$

With Simplification 3 on page 43, this results in the free theorem $(words = words)$. \square

5 Implementation

It was pointed out earlier that generating free theorems from types is a rewriting which can also be automated. The results of implementing an application capable of doing exactly that are described in this chapter.

To foster reusability, the application is split up in two parts. A library, called *free-theorems*, encapsulates the algorithms for generating free theorems from Haskell type signatures, while a separate user interface, named *ftshell*, allows for interacting with that library. Both parts are written entirely in Haskell, but rely on language features not covered in the Haskell 98 language report [Jon03], namely multi-parameter type classes [JJM97] and generics [LJ03]. Due to these non-portable features, especially generics, only the GHC [GHC06] currently supports *free-theorems* and *ftshell*. Section 5.1 will give the reason for still favouring the mentioned language features over supporting all established Haskell compilers and interpreters.

There exist at least two other implementations of the original paper on free theorems [Wad89], both usable as plug-ins for Lambdabot [Lam07, SC05]. One of these two implementations is described in detail in [Böh06] and served as a prototype implementation for the library *free-theorems* and the user interface *ftshell*. The other implementation [Bro06], which is intended solely as a Lambdabot plug-in, covers only parametricity results in which all relations are specialised to functions. Furthermore, it does not mention restrictions on those functions, as for example strictness, which suggests that this implementation restricts itself to what is described as (*basic*, =) model in this thesis.

This chapter is structured as follows. Section 5.1 explains the structure and the interface of the library *free-theorems* and discusses the difference and improvements compared to the prototype of [Böh06]. Afterwards, in Section 5.2, the user interface *ftshell* is introduced by listing the main commands and showing a simple session. Section 5.3 completes this chapter by presenting the libraries and applications used to develop both the library *free-theorems* as well as the user interface *ftshell*.

5.1 The library *free-theorems*

Theorem 1 on page 39 together with the logical relation of Definition 24 on page 35 and the relational actions defined in Section 3.5 describe rewriting rules for obtaining theorems from type signatures. These rules form the core of the library *free-theorems*. Additionally, the library contains functions and data structures for processing input data, which is referred to as front-end, as well as data structures and pretty printers for handling output data. The overall structure of the *free-theorems* library is shown in Figure 5.1.

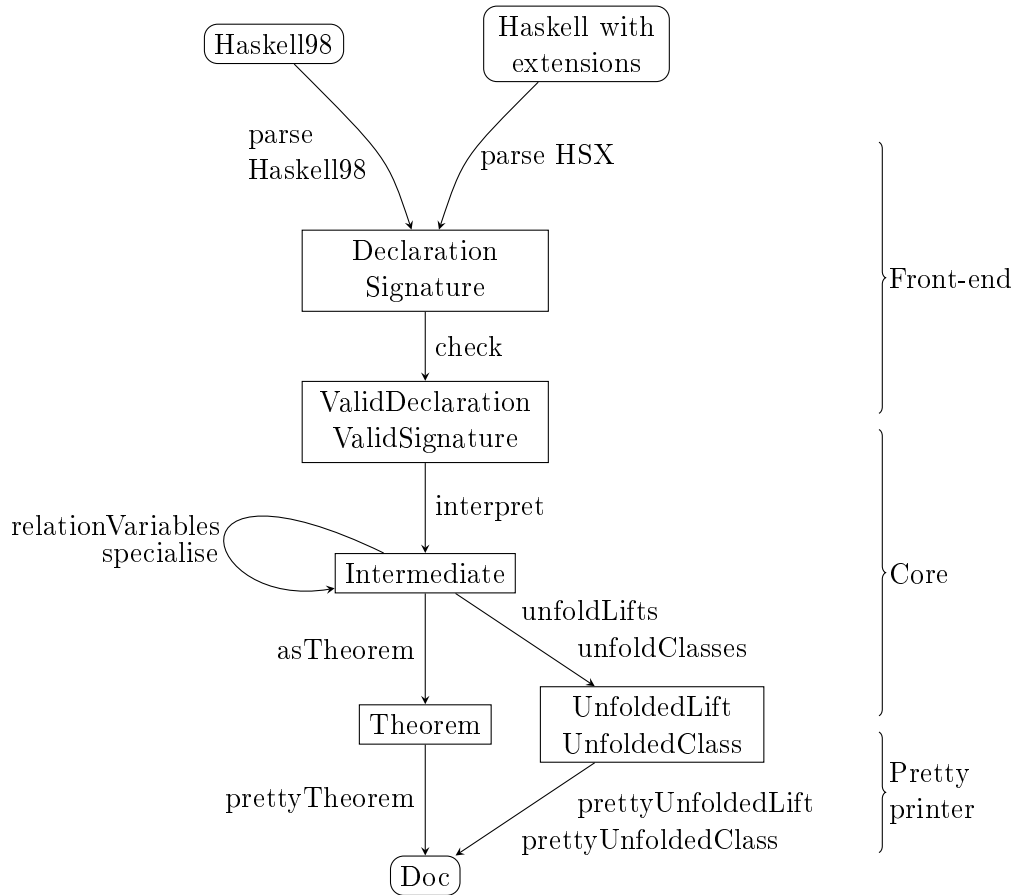


Figure 5.1: Overview of the interplay of the data structures and functions provided by the library *free-theorems*. The rectangular boxes contain data types specified by *free-theorems*, while rounded boxes describe external data types. The edge labels represent names of functions defined by *free-theorems*.

The *free-theorems* library works as follows. A list of *Declaration* elements, each of which represent a Haskell declaration in an abstract syntax, is checked according to the definitions of Section 3.3. This step filters valid declarations and signatures and creates suitable error messages for all invalid ones. Valid signatures can then be interpreted, which is essentially a rewriting into a relational representation according to Definition 24 on page 35 and Theorem 1 on page 39. To finally obtain a theorem, the definitions of Section 3.5 are applied to unfold the relational representation. Additionally, definitions of relations corresponding to non-nullary user-defined type constructors occurring in a theorem can be unfolded using Definition 16 on page 32. In a similar way, by applying Definition 21 on page 34, class constraints occurring in a theorem can be unfolded. Eventually, all data types of the *free-theorems* library can be transformed into human-readable text by means of a provided pretty printer.

The key feature of *free-theorems* is to encapsulate the theory presented in the pre-

ceding chapter and to provide these algorithms to several different applications. This is accomplished by providing two sets of data types, one belonging to the front-end and the other one representing generated theorems.

The front-end set of data types comprises types like *Declaration*, *Signature* as well as *TypeExpression*, which have been modelled after definitions of Section 3.2 and Section 3.3. Although there exist several other representations of Haskell’s abstract syntax, these are usually tightly connected with Haskell parsers and do not always provide all features required by *free-theorems*. To gain independence and good reusability, the library *free-theorems* has its own abstract syntax representation of (a subset of) Haskell. As a consequence, the *free-theorems* library can be used in conjunction with any source of abstract Haskell syntax, for example a Haskell parser, provided that this abstract syntax is transformed into data structures specified by *free-theorems*.

The second set of data types provided by *free-theorems* covers *Theorem*, *UnfoldedLift*, *UnfoldedClass* and other types these three are depending on. These types represent generated theorems and can be used to alter or pretty-print these theorems. For example, additionally to the already existing pretty printer, it is possible to implement another pretty printer creating L^AT_EX documents based on the described theorem types.

Since the *free-theorems* library is based on the prototype described in [Böh06], it has a similar interface. Internally however, *free-theorems* is mostly rewritten from scratch due to the increased requirements posed by the language subsets. Another reason which led to major internal differences is, that *free-theorems* can handle any user-defined declarations, while the prototype was restricted to a fixed, predefined set of declarations only. This made it necessary to implement checks to ensure the requirements defined in Section 3.3.

Being able to handle any declarations in the *free-theorems* library requires a suitable parser. The aforementioned prototype has its own parser implementation, but due to the high complexity of Haskell declarations, this was considered as too burdensome. Alternatively, there exists a well-documented parser for Haskell 98 available for all Haskell compilers and interpreters, which, however, does not support explicit quantifications of type variables. To still get the full power provided by the core of the *free-theorems* library, a similar, but extended implementation [HSX05] can be used. In fact, both the standardly available Haskell 98 parser as well as the extended Haskell parser are integrated in the front-end of the *free-theorems* library. Other available Haskell parsers, for example that of the GHC, may be added as well.

Although not defined in the Haskell 98 report, multi-parameter type classes proved to give important and often needed abstractions for various problems. One field of their successful application is to combine monads expressing states, error handling and logging facilities [Wad92] by the help of monad transformers [Jon95, LHJ95]. The *free-theorems* library makes also use of this feature, as for example in the function interpreting type expressions as relations. There, an environment mapping type variables to relation variables has to be maintained, while, at the same time, new relation variables can be generated based on a state.

In [Böh06], it was suggested to improve the automatic free theorems generator by the help of generics [LJ03]. This technique allows to avoid writing boilerplate code which traverses through huge data structures only to apply some modifications at special

```

checkArity :: Declaration → Bool
checkArity = everywhere (&&) (True `mkQ` hasCorrectArity)
where
  hasCorrectArity :: TypeExpression → Bool
  hasCorrectArity t = case t of
    TypeCon ConInt ts → length ts == 0
    TypeCon ConList ts → length ts == 1
    otherwise         → True

```

Figure 5.2: A simplified example taken from *free-theorems*, which demonstrates the usage of generics. The combinator *everywhere* applies the query *hasCorrectArity* at every suitable node and combines the results and the value *True* using the logical conjunction operator (*&&*). The case specified by *otherwise* covers, for example, function types and type abstractions. The complete function checks also other cases of type constructors and returns error messages instead of boolean values.

nodes. With the help of generics, not only the source code is getting more succinct, but also possible sources of errors are avoided. Unfortunately, this powerful feature is only supported by the GHC [GHC06].

The *free-theorems* library applies generics mainly in the front-end for checking restrictions of the abstract syntax. Without generics, that would mean to traverse more than ten different data types in several different ways, which gets easily complicated and hard to maintain. A dramatic simplification can be gained by taking a generic approach. Figure 5.2 demonstrates how checking the arity of type constructors everywhere in a declaration is expressible by just two functions. In the same way, the implementation of many similar problems is greatly alleviated by the help of generics which justifies the application of this technique.

5.2 The user interface *ftshell*

To allow users to interact with the *free-theorems* library, the shell-based, textual application *ftshell* is provided. The basic model of interaction with it is by entering commands to query information or to modify the internal state of *ftshell*. This is exactly the same concept as was applied in the prototypic implementation described in [Böh06]. In fact, most components of that prototype could be reused in *ftshell*, while some had to be modified or extended to fit to the interface of the *free-theorems* library.

The commands provided by the user interface *ftshell* can be grouped into two sets, general commands and theorem-related commands. The first and bigger set of general commands is always available. These commands allow for querying available declarations, changing the current language subset, modifying pretty-printing options and managing

5 Implementation

files which contain declarations. By contrast, theorem-related commands are available only after the user entered a type signature or selected a name from the list of loaded type signatures. This second set of commands is intended to modify the theorem generated for the chosen type signature by specialising relation variables to function variables or by querying definitions of lifted relations or class constraints occurring in a theorem. A selection of the most important commands, both general and theorem-related ones, is shown in Figure 5.3 on the facing page.

While type signatures can be entered directly in the shell, is this not possible for declarations of types and classes. To cope with that, declarations can be loaded from files. A basic file containing declarations and type signatures taken from the Haskell *Prelude* and five other standard Haskell modules, namely *Data.Complex*, *Data.List*, *Data.Maybe*, *Data.Monoid* and *Data.Ratio*, is loaded automatically when starting *ftshell*.

The remainder of this section presents a session, in which the free theorem for the Haskell function *length* in the $(fix, =)$ model is generated (see also Section 4.2). When starting *ftshell*, the following welcome message is printed.

```
FTshell (version 0.1) - Automatic generation of free theorems
Copyright 2007, Sascha Boehme
```

```
Press ':help' for help or ':quit' to quit.
```

```
Loading 'declarations.hs' ... found 187 declarations.
```

The application is now waiting for commands or type signatures. First, the $(fix, =)$ model is selected.

```
> :fix-equational
The current language subset is 'fix-equational'.
```

Next, the name of the function *length* is entered. As a result, the shell returns a theorem generated for that type signature.

```
> length
The free theorem for the type signature

length :: forall a . [a] -> Int

in the language subset 'fix-equational' is:

forall t1,t2 in TYPES, R1 in REL(t1,t2), R1 strict and
continuous.
  forall (x,y) in p-lift_{[]} (R1). length x = length y
```

Finally, the relation *R1* is specialised to a function. This results in a weaker theorem as follows.

:help Displays an overview of all available commands. Two additional help commands explain the usage of the shell and give an introduction to free theorems, respectively.

:quit Exits the application.

:all-declarations Displays the names of all available declarations. For every loaded file, algebraic data types, type renamings, type synonyms, type classes and type signatures are shown in separate groups.

:signatures Displays all available type signatures. This command does not only list the names of available signatures as **:all-declaration** does, but it shows also the corresponding type expression.

:declaration <name> Displays the declaration for the given name. Only the names displayed by the command **:all-declarations** can be given as an argument to this command.

:basic-subset Changes the current model to the (*basic*, =) model.

:fix-equational Changes the current model to the (*fix*, =) model.

:fix-inequational Changes the current model to the (*fix*, \sqsubseteq) model.

:seq-equational Changes the current model to the (*seq*, =) model.

:seq-inequational Changes the current model to the (*seq*, \sqsubseteq) model.

:relation-variables Displays all relation variables of the current theorem, which may be specialised. This does not include variables representing functions.

:specialise <relation-variable> Specialises the relation corresponding to the given relation variable to a function and updates the theorem. Only relation variables displayed by **:relation-variables** may be specialised.

:lifts Gives the definitions for lifted relations occurring in the current theorem.

:classes Displays the class constraints on relations and functions of the current theorem.

Figure 5.3: The most important commands of the user interface *ftshell* and their explanation.

```

length > :specialise R1
The free theorem for the type signature

    length :: forall a . [a] -> Int

in the language subset ‘fix-equational’ is:

forall t1,t2 in TYPES, f :: t1 -> t2, f strict.
  forall x :: [t1]. length x = length (map f x)

```

This final theorem is the point-wise notation of the theorem described in Example 16 on page 43.

5.3 Used applications and libraries

A number of applications and libraries aided in developing the library *free-theorems* and the shell application *ftshell*. This section presents these tools by shortly explaining what they do and how they have been applied in the implementation process.

Haddock

Haddock [Had05] is a tool to automatically generate documentation from annotated Haskell source code. It extracts all exposed declarations and type signature along with specially marked Haskell comments and creates hyperlinked HTML files. By that, Haddock greatly contributes to foster the reuse of Haskell code.

Although only few declared types and defined functions are exposed by *free-theorems*, every module of the library is completely documented with Haddock comments which improves the maintainability of the library.

QuickCheck

The QuickCheck library [Qui02] is an automatic testing tool for Haskell. It applies programmer-supplied properties to a large number of randomly generated test data.

Properties to be checked are written as ordinary Haskell functions. An example property taken from the test suite of *free-theorems* is the following one.

```

prop_closureVariables t vs =
  Set.null (set ‘Set.intersection‘ freeTypeVariables (closureFor set t))
  where
    types = (t :: TypeExpression, vs :: [TypeVariable])
    set = Set.fromList vs

```

This property checks that, after binding the type variables *vs* in the type expression *t* by adding additional type abstractions, these type variables do not occur free in the resulting type expression.

Altogether more than 40 properties have been written to increase reliability of the library *free-theorems*. Like the given example, the majority of these properties is concerned with checking the *free-theorems* front-end. There are also two tests based on ideas of [DJ04] which check that parsing a pretty-printed declaration yields the original declaration.

Opposed to the library *free-theorems*, no automatic tests were defined for the user interface *ftshell*. Instead, only manual tests were performed to find errors.

Cabal

The Common Architecture for Building Applications and Libraries, abbreviated by Cabal, defines a standard for packaging and building Haskell libraries and applications [Cab06].

Packages conforming to the Cabal consist of at least a package description and a setup module used to configure, build and install the library or application. Due to the infrastructure provided by the Cabal, the setup module is mostly not longer than five lines of Haskell source code. As an example, the content of the file `Setup.lhs`, which is part of the *free-theorems* library, is as follows.

```
#!/usr/bin/env runhaskell

> import Distribution.Simple
> main = defaultMain
```

A package description contains general information of the package like its name, its author and its license as well as information needed to build the package. The latter includes, for example, dependencies on other packages, exposed modules and language extensions used in the source code. An extract of the package description for the library *free-theorems* is shown in Figure 5.4 on the following page.

Darcs

Darcs [Dar06, Rou05] is a distributed revision control system. Revision control systems in general track changes of source code, handle conflicts caused by different modifications and allow for undoing any changes. While such systems usually depend on one central server to store all changes, darcs extends this model by tracking modifications locally for every user and, when needed, merging these modifications between users. To make this possible, a unique algebra of patches was especially developed for darcs.

Since most current Haskell projects apply darcs, it was used also in the implementation of the library *free-theorems* and the user interface *ftshell*. Due to the fact that only one developer worked on these projects, darcs gave no further benefit than serving as a backup tool for the source code.

```
name:          free-theorems
version:       0.1
license:       BSD3
author:        Sascha Boehme
synopsis:      Automatic generation of free theorems.
category:      Language
build-depends:
  base >= 1.0,
  mtl >= 1.0
  haskell-src >= 1.0,
  haskell-src-extensions >= 0.2
exposed-modules:
  Language.Haskell.FreeTheorems
  Language.Haskell.FreeTheorems.Syntax
  Language.Haskell.FreeTheorems.Parser.Haskell98
  Language.Haskell.FreeTheorems.Parser.Hsx
  Language.Haskell.FreeTheorems.Theorems
extensions:
  Generics
```

Figure 5.4: An extract of the package description file `free-theorems.cabal`, which is part of the *free-theorems* library.

Pretty printer

A pretty printer lays out structured data in a formatted way which makes the structure visible. In [Hug95], the design and implementation of a pretty printer library was explained, guided by a formal specification. This library was later improved (see [PJ97]) and is, by default, included in the Haskell base package of GHC [GHC06].

This pretty printer is applied by the *free-theorems* library to format declarations and theorems.

Shellac

Shellac [Doc07] is a library to aid in developing shell-based applications. It simplifies the complexity involved in implementing a shell in a way, that, essentially, only command names and functions implementing these commands are needed to set up a shell application. However, Shellac utilises language features provided only by GHC [GHC06] for now. Other Haskell compilers or interpreters are not supported.

6 Conclusions

This thesis describes free theorems in three sublanguages of the functional programming language Haskell. The considered sublanguages have been studied before already, but this thesis compiles the different notations used in the literature into one common scheme, which, in addition, is enriched in several ways. First, based on [Wad89], type classes are covered. Second, this thesis applies the idea of inequational results, which is described in [JV06], to a sublanguage of Haskell, where only equational results have been considered so far. Third, in extending [Böh06], all kinds of Haskell data type declaration are included in the scheme, in particular type renamings and algebraic data types with strictness flags. Finally, possible simplifications have been studied on examples and generalised.

Accompanying this thesis, the algorithm for deriving free theorems was implemented in Haskell, which resulted in the library *free-theorems* and the user interface *ftshell*. Although based on the prototype described in [Böh06], the library was mainly rewritten from scratch, using Haskell generics to keep the code maintainable and small. In fact, the library *ftshell* has roughly the same number of code lines as the prototype, but at the same time, the former implements much more functionality than the latter, in particular three additional models and fully user-defined data types. In contrast to the library *free-theorems* and its prototype, the user interface *ftshell* differs only slightly from the prototype's user interface; modifications were mostly caused by changes to the underlying library.

As is usual in software projects, implementations should at least be correct and extensible. In the case of the user interface *ftshell*, the first property is, to some degree, easy to establish, because just a few user interactions are necessary to test every feature. And, since *ftshell* is based on Shellac [Doc07], extensibility is guaranteed; providing further commands essentially requires to just define a new function for every command. Note also, that only slight modifications were necessary to turn the prototype interface into *ftshell*. Considering the implementation of the more complex library *free-theorems*, the situation is not that simple. To achieve correctness, several automatic tests have been written and manual checks on small examples have been performed. Since this method is not complete, that is, it does not guarantee to find all errors, it still helped a lot to detect several bugs and improved the implemented library. Extensibility of *free-theorems* was achieved in different degrees. The front-end parser and the pretty-printer are easily exchangeable, that is, alternatives for them can be implemented without modifications to the library's core. Extending the core, however, is more complex because of several dependencies in its inner structure. This is similar to the theory of Chapter 3, on which the library is based. Assume, for example, that the definition of type expressions would be extended. Then, nearly all concepts of Chapter 3 need to be reconsidered. In much the same way, even slight modifications to the core of *free-theorems* may cause several

6 Conclusions

changes in other places of the code.

While the algorithm to automatically generate free theorems has been implemented completely, the *free-theorems* library lacks some of the simplifications described in Chapter 4, especially point-free notations, parts of Simplification 4 on page 44 and Simplification 7 on page 49. Even more, since only simple examples have been considered in Chapter 4, other possible simplifications still remain to be identified.

Another missing feature of the implementation already suggested in [Böh06] is to output generated free theorems in L^AT_EX. This feature would greatly increase the readability of the obtained results. Note that the structure of the *free-theorems* library was designed to allow such an alternative pretty-printer.

This thesis does not cover type constructor classes [Jon93]. The adjustments necessary to incorporate them would have an impact on several parts of the theory presented in Chapter 3; besides redefining type expressions and class declarations, especially relational actions need to be reconsidered. Note, however, that type constructor classes, especially monads, have several applications in Haskell, and supporting them could possibly broaden the field of applications of free theorems.

Acknowledgments

First of all, I want to thank my supervisor Janis Voigtländer. I am grateful for his challenging tasks and for always demanding correct explanations. In addition, I owe him a lot for patiently helping me with all my question and for giving me good advices, but also for introducing me to the exciting field of functional programming.

I also want to thank Heiko Vogler for his guidance during my study years. In fact, five years ago, I would have never anticipated that a single bar of chocolate could pave my path until now. I am deeply grateful for his constant support and for offering me several opportunities to improve my knowledge and also my writing and presentation skills. Especially many thanks for letting me participate in two exchange trips to Szeged.

Many thanks also to Torsten Stüber for helpful questions, suggestions and mathematical discussions in general. At the same time, I am grateful that he spend some of his valuable time on listening to my explanations and commenting them.

Finally, I want thank my parents, my brother and Emmi for being there.

Bibliography

- [Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
- [BM98] Richard Bird and Lambert Meertens. Nested Datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422, pages 52–67. Springer-Verlag, Berlin, 1998.
- [Böh06] Sascha Böhme. Automatische Erzeugung freier Theoreme. Großer Beleg, Technische Universität Dresden, 2006.
- [Bro06] Andrew James Bromage. Free theorems lambdabot plugin, October 2006. Available at <http://andrew.bromage.org/darcs/freetheorems>.
- [Cab06] The Haskell Cabal: Common Architecture for Building Applications and Libraries. <http://www.haskell.org/cabal>, October 2006. Version 1.1.4.
- [Dar06] Darcs: A distributed revision control system. <http://darcs.net>, June 2006. Version 1.0.8.
- [DJ04] Nils Anders Danielsson and Patrik Jansson. Chasing Bottoms, A Case Study in Program Verification in the Presence of Partial and Infinite Values. In *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, pages 85–109. Springer-Verlag, July 2004.
- [Doc07] Robert Dockins. Shellac: A framework for building read-eval-print style shells. <http://www.eecs.tufts.edu/~rdocki01/shellac.html>, February 2007. Version 0.9.
- [GHC06] GHC: The Glasgow Haskell Compiler. <http://www.haskell.org/ghc>, April 2006. Version 6.4.2.
- [Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.
- [GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.
- [Had05] Haddock: A Haskell Documentation Tool. <http://www.haskell.org/haddock>, August 2005. Version 0.7.

- [Hin69] J. Roger Hindley. The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
- [HP06] Ralf Hinze and Ross Paterson. Finger Trees: A Simple General-purpose Data Structure. *Journal of Functional Programming*, 16(2):197–217, 2006.
- [HPF99] Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction to Haskell Version 98*, October 1999.
- [HSX05] Haskell source with extensions. <http://www.cs.chalmers.se/~d00nibro/haskell-src-exts>, April 2005. Version 0.2.
- [Hug95] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925. Springer Verlag, 1995.
- [JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop*, June 1997.
- [Joh02] Patricia Johann. A Generalization of Short-Cut Fusion and its Correctness Proof. *Higher Order and Symbolic Computation*, 15(4):273–300, 2002.
- [Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61. ACM Press, 1993.
- [Jon95] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995.
- [Jon03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, May 2003.
- [JV06] Patricia Johann and Janis Voigtländer. The Impact of *seq* on Free Theorems-Based Program Transformations. *Fundamenta Informaticae*, 69:63–102, 2006.
- [Lam07] Lambdabot. <http://haskell.org/haskellwiki/Lambdabot>, April 2007. Version 4.
- [LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 333–343. ACM Press, 1995.
- [LJ03] Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37. ACM Press, 2003.

Bibliography

- [LP96] John Launchbury and Ross Paterson. Parametricity and Unboxing with Unpointed Types. In *European Symposium on Programming, Proceedings*, pages 204–218. Springer-Verlag, 1996.
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [PJ97] Simon L. Peyton Jones. A pretty printer library in Haskell, 1997. Available at <http://research.microsoft.com/~simonpj/downloads/pretty-printer/pretty.html>.
- [Plo80] Gordon Plotkin. Lambda-definability in the full type hierarchy. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.
- [Qui02] Quickcheck: An Automatic Testing Tool for Haskell. <http://www.cs.chalmers.se/~rjmh/QuickCheck>, 2002.
- [Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.
- [Rey83] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier Science Publishers B. V. (North-Holland), 1983.
- [Rou05] David Roundy. Darcs: Distributed Version Management in Haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4. ACM Press, 2005.
- [SC05] Don Stewart and Manuel M. T. Chakravarty. Dynamic Applications From the Ground Up. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM Press, 2005.
- [Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, 1986.
- [Sta85] Richard Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65:85–97, 1985.
- [Str67] Christopher Strachey. Fundamental Concepts in Programming Languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 11–49, 2000.
- [Voi02] Janis Voigtländer. Concatenate, Reverse and Map Vanish For Free. In *International Conference on Functional Programming, Proceedings*, pages 14–25. ACM Press, 2002.

- [Wad89] Philip Wadler. Theorems for Free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
- [Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIG ACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.
- [WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

Index

This list gives the most important notions and the pages where these notions appear first or where they are defined.

A

admissible relation 18
algebraic data type 12, 25
 lifted 15

B

bottom-reflecting relation 18
bound type variable 21

C

class constraint 16
class method 16
closed
 term 10
 type expression 21
complete partial order 18
continuous relation 18

D

data constructor 12
directed graph 19

E

environment 31

F

free type variable 20
function type constructor 19

I

identity relation 31
instance

 of a polymorphic value 30
 of a type class 16, 30

L

left-closed relation 18
logical relation 35

M

model 17
monotonic function 19

N

natural numbers 18
nested type constructor 26

O

ordered set 18

P

parametricity theorem 39
pattern matching 14
permissible
 type class 28
 type constructor 27
position of a type constructor 25
primitive type constructor 23

R

ranked set 18
relational action
 function type constructor 33, 34
 type abstraction constructor 35

user-defined type constructor 32
 restricted, acyclic subset 19

S

strict relation 18
 strictness flag 14, 25

T

term 10
 total relation 18
 type abstraction 19
 type class 16, 19
 type constructor 12, 19
 of a type renaming 26
 of a type synonym 23
 of an algebraic data type 25
 type expression 19
 type instantiation 30
 type renaming 15, 26
 unlifted 15
 type synonym 15, 23
 type variable 19

U

user-defined type constructor 23

Symbols

This page gives the most important symbols and notations and the pages where they are defined. The following short list names some of the symbols occurring in the main list.

n, i, j	natural numbers	$\tau, \tau_1, \dots, \tau_n$	type expressions
m	model	σ	type constructor
R, R_1, \dots, R_n, S	relations	$\alpha_1, \dots, \alpha_n$	type variables
$\forall_{\Delta} R \in Rel^m. F(R)$	35	$Rel(\tau_1, \tau_2)$	30
$R_1 \rightarrow^{(basic, =)} R_2$	33	$Rel^{(basic, =)}$	31
$R_1 \rightarrow R_2$	34	$Rel^{(basic, =)}(\tau_1, \tau_2)$	30
$R_1 \rightarrow^{(seq, =)} R_2$	34	$Rel^{(fix, =)}$	31
$R_1 \rightarrow^{(seq, \sqsubseteq)} R_2$	34	$Rel^{(fix, =)}(\tau_1, \tau_2)$	31
id_{τ}	31	$Rel^{(fix, \sqsubseteq)}$	31
id_{τ}^{\perp}	31	$Rel^{(fix, \sqsubseteq)}(\tau_1, \tau_2)$	31
$lift_{\sigma}(R_1, \dots, R_n)$	32	$Rel^{(seq, =)}$	31
$lift_{\sigma}^{\perp}(R_1, \dots, R_n)$	32	$Rel^{(seq, =)}(\tau_1, \tau_2)$	31
η	31	$Rel^{(seq, \sqsubseteq)}$	31
$\llbracket \tau \rrbracket_{\eta}^m$	35	$Rel^{(seq, \sqsubseteq)}(\tau_1, \tau_2)$	31
M	17	D	19
$(basic, =)$	17	D_*	28
$(fix, =)$	17	C	19
(fix, \sqsubseteq)	17	C_a	23
$(seq, =)$	17	C_*	27
(seq, \sqsubseteq)	17	C_p	23
\mathbb{N}	18	C_r	23
\mathbb{N}_0	18	C_s	23
\sqsubseteq	18	(σ, i, j)	25
\sqsupseteq	18	$T(C, D)$	19
\perp	11, 18	$T_c(C, D)$	21
$\mathcal{P}(X)$	18	T	28
$R; S$	18	T_c	28
R^{-1}	18	T_{Δ}	30
Rel	31	$\tau[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$	21
		V	19
		$\langle \tau \rangle^{\perp}$	29
		$\langle \tau \rangle$	29

Erklärung

Hiermit erkläre ich, Sascha Böhme, dass ich die vorliegende Diplomarbeit selbständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Ort, Datum

Unterschrift