Investigating Minimally Strict Functions in Functional Programming

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur
(Dr.-Ing.)

der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Jan Christiansen

Kiel

2011

1. Gutachter	Prof. Dr. Rudolf Berghammer
2. Gutachter	Priv.-Doz. Dr. Frank Huch
Datum der mündlichen Prüfung	12. Januar 2012

Contents

Acknowledgments

First of all I am very grateful to Olaf Chitil for introducing me to his excellent idea of checking whether a function is unnecessarily strict. Furthermore, I want to thank Rudolf Berghammer for his support, for tolerating my occasional loose tongue, and for not loosing patience with me. I am indebted to Fabian Reck for being an excellent officemate and a wine expert, and for always lending me an ear. Besides, I want to thank Daniel Seidel for being a great workmate and host and for even becoming a good friend. I owe thank to Janis Voigtländer for introducing me to the idea of free theorems and for collaborating with me in general. With respect to the work at hand I want to thank Ole Cordsen, Nikita Danilenko, Sebastian Fischer, Hauke Fuhrmann, Frank Kupke, Björn Peemöller, Matthias Quednau, Fabian Reck, Daniel Seidel, and Gabriel Wicke for reading parts of this thesis. I want to thank the "gang from the seventh floor", which, besides some members already mentioned, includes Bernd Braßel, Stefan Bolus, Michael Hanus, Thomas Heß, Frank Huch, Ina Pfannschmidt, Peter Pichol, and Ulrike Pollakowski for providing a nice working environment and sharing lunch. In particular, I would like to single out the benefits that we all enjoy by having a great secretary like Ulrike. Moreover, thanks to the countless number of students for giving me the opportunity to teach them the basic parts of functional programming in several first term practical courses. Also, I would like to thank the easily countable number of students that took my advanced courses. Finally, a very big thanks to the team of the cafetaria of the "Studentenwerk Schleswig-Holstein" without whose constant supply with coffee and sugar in the form of muffins and donuts this work could not have been finished.

1 Introduction

In this thesis we consider the non-strict, functional programming language Haskell (Peyton Jones 2003). First of all, functional programing languages belong to the class of declarative programming languages. While in an imperative programming language the programmer specifies a procedure to get a solution to a problem, in a declarative language the programmer instead specifies the structure of a solution. In other words, while in an imperative language we define how a solution is computed, in a declarative language we rather define what is computed. Besides, Haskell has a quite strong type system, which is able to catch some bugs at compile-time, while these bugs do not emerge until run-time in a language with a weaker type system or no type system at all. And, finally, features like higher-order functions and overloading allow programmers to reuse code, which results in programs that are smaller and easier to maintain.

However, even with a comparatively high-level language like Haskell, programmers may write programs that are not optimal with respect to a certain criterion. Obviously, the most important criterion is correctness. That is, we prefer a program that meets our specifications over a program that does not. There are several approaches that assist programmers in writing correct Haskell programs like property-based testing (Claessen and Hughes 2000) or even automatic proving (Sonnex et al. 2011). Nonetheless, there are other important criteria to rate programs. For example, we could classify several implementations of the same algorithm in one language by means of run-time or memory consumption. This thesis investigates another criterion, namely, rating Haskell programs by means of strictness.

Intuitively, in contrast to a strict programming language, in a non-strict programming language like Haskell a function only inspects the part of its argument that is needed to yield a certain part of its result. In the extreme case of a constant function, for example, the function does not evaluate its argument at all as the result of the function does not depend on the argument. Nevertheless, we can still define a constant function in Haskell that unnecessarily inspects its argument. More generally speaking, in Haskell a function definition rather non-declaratively specifies how an argument is inspected. There are often several implementations of functions with the same basic behavior that inspect different parts of their argument. While in the case of a constant function it is quite easy to observe that it does not have to inspect its argument, we will see that in general it is more difficult to observe whether a function can yield the same result by inspecting a smaller part of its argument.

To rate functions with respect to strictness, we consider the following definition. A function f is less strict than a function g if the function f never inspects a larger part of its argument than the function g does, to yield a certain result. In addition, there exists a result such that f inspects a strictly smaller part of its argument than g does. We call a function f with these properties less strict than g. This "definition" is quite informal, and we will later give a formal definition of the less-strict relation by

means of a denotational semantics. For now this intuitive view should be sufficient for this introduction and will be helpful later on.

As an example for the interest in less strict functions, let us consider the standard Haskell function *partition*. Waldmann (2000) as well as Russell (2000) have stated that the implementation of this function was more strict than necessary. The function *partition* takes a predicate — a unary funtion that yields a Boolean value — and a list — an ordered collection of values of equal type — and yields a pair — a container for two values. The pair holds two lists where the first list contains the elements that satisfy the predicate and the second list contains the elements that do not. It has been observed that *partition*, at that time, was unnecessarily strict because the implementation did not comply with an abstract specification given in the Haskell language report (Peyton Jones 2003).

The language report specifies the behavior of *partition* by means of the standard Haskell function *filter*. The function *filter* takes a list and a predicate and yields a list of all elements that satisfy the predicate. The behavior of *partition* is specified by the following equivalence where p is an arbitrary predicate and xs an arbitrary list. Furthermore, the symbol $\equiv$ denotes that the left-hand and the right-hand side yield the same results when evaluated.

$$\mathit{partition}\ p\ xs \equiv (\mathit{filter}\ p\ xs, \mathit{filter}\ (\mathit{not} \circ p)\ xs)$$

The application of a function is denoted by juxta-position, for example, *partition p xs* denotes the application of the function *partition* to the arguments p and xs. The parenthesis together with the comma on the right-hand side of the equivalence denote the construction of a pair. That is, *partition* applied to some predicate p and a list xs yields the same result as the pair that consists of an application of *filter* to the predicate and the list and an application of *filter* to the negation of the predicate and the list. The operator $\circ$ denotes function composition and *not* is the Boolean negation, so the term $\mathit{not} \circ p$ denotes a predicate that first applies the original predicate p and negates its Boolean result.

Waldmann (2000) named the following example, which demonstrates, that the original implementation of *partition* does not comply with the behavior of two applications of *filter*.

$$\textbf{let}\ (xs, ys) = \mathit{partition}\ \mathit{odd}\ [1\,..]\ \textbf{in}\ (\mathit{take}\ 3\ xs, \mathit{take}\ 3\ ys)$$

The expression $[1\,..]$ denotes an infinite list of increasing natural numbers that starts with one. Because Haskell is a non-strict programming language we can use infinite data structures as a function might only inspect a small part of the infinite structure. The infinite list is partitioned into odd and even numbers by using *partition* and the predicate *odd*, which is satisfied if its argument is an odd number. The **let** expression simply introduces names for the components of the result, which is a pair, of the function *partition*. By equating a pair constructed of the variables xs and ys with the application $\mathit{partition}\ \mathit{odd}\ [1\,..]$, the variables xs and ys become names for the first and the second component of the result of *partition*, respectively. We can use the variables xs and ys on the right-hand side of the keyword **in** as abbreviations for

these components. Finally, the applications *take* 3 *xs* and *take* 3 *ys* yield the prefixes of length three of the lists *xs* and *ys*, respectively.

The evaluation of this **let** expression does not terminate if we use the original implementation of *partition*. At that time the implementation of *partition* inspected its whole argument, which is the infinite list $[1..]$ in this example. In contrast, if we define *partition* by means of *filter* the evaluation of the expression above yields the pair $([1,3,5],[2,4,6])$. Here, the square brackets denote a list and each pair of consecutive elements of the list is separated by a comma. That is, a definition of *partition* by means of *filter* yields a list with the first three odd numbers and a list with the first three even numbers as this definition inspects only a small part of the infinite list $[1..]$. More precisely, the application *take* 3 *xs* as well as *take* 3 *ys* only inspect the first three elements of *xs* and *ys*, respectively. The application *filter odd* $[1..]$ inspects the first five elements of $[1..]$ to yield the list $[1,3,5]$. Moreover, the application *filter* $(not \circ odd)$ $[1..]$ inspects the first six elements of $[1..]$ to yield the list $[2,4,6]$. Thus, both applications of *filter* inspect only a finite part of the infinite list.

The implementation of *partition* has been improved by introducing a so-called lazy pattern, which delays a check whether a value has a certain structure. Essentially, the definition of *partition* has to check whether the result of its recursive application is a pair to access the components of the pair. The original definition of *partition* accessed the components of the recursive application before yielding the first part of its result. To improve the behavior of *partition* with respect to strictness, the check for the pair structure of the recursive application has to be delayed.

Waldmann (2000) has already proposed to develop means to formally reason if a function can be improved by adding such a lazy pattern as well as tools that support the programmer in checking whether a function can be improved this way. One contribution of this thesis is a part-solution to his proposals. First, we present a criterion to check if a function is overly strict. This criterion is an answer to a more general question than the question whether a function can be improved by a lazy pattern. Furthermore, we present a tool called Sloth, which assists the programmer in checking whether a function is unnecessarily strict. In particular, the tool states that a function is overly strict in the case that it can be improved by introducing a lazy pattern. For example, in Section 6.6, using Sloth, we improve an implementation of the enumeration of elements of a tree in breadth-first order by introducing a lazy pattern. However, the tool only states whether a function is too strict and is not able to discover how a function can be improved. The tool, in particular, does not state whether a function can be improved by introducing a lazy pattern.

As another example for the interest in less strict implementations, let us consider the standard function *permutations*, which takes a list and yields a list of all permutations of the argument. The current definition of *permutations* emerged from a discussion initiated by van Laarhoven (2007).

Via a thorough refinement process the contributors of the discussion have defined a quite efficient implementation of *permutations* that, moreover, is less strict than the naive implementation they first came up with. More precisely, they have specified how little strict they expected an implementation of *permutations* to be. Intuitively, they stated that *permutations* is supposed to enumerate $n!$ permutations, each of length n, by inspecting at most the first n elements of its argument list.

This property can be considered as a kind of generalization of the concept of *on-line* behavior by Pippenger (1997). A function that takes a list and yields a list provides on-line behavior if the function inspects at most the first n elements of its argument to yield n elements of the result. The property of *permutations*, stated above, in fact, generalizes this idea to a function that yields a list of lists as result and to the special case of a function that enumerates permutations.

As a side note, functions with on-line behavior play an important role with respect to the expressiveness of pure functional programs with respect to complexity. Pippenger (1997) has shown that there are certain functions with on-line behavior that have a linear complexity in a strict functional language with destructive updates — he considers LISP — while in a strict language without destructive updates we always get an additional logarithmic factor. Bird et al. (1997) have provided an implementation of the same function in a lazy functional language that has linear complexity, too. That is, a lazy purely functional programming language is in a certain sense more expressive than a strict purely functional programming language. In this context the notion pure refers to the absence of destructive updates. Bird et al. (1997) make use of a less strict matrix transposition in order to get the desired on-line behavior in the lazy language. We consider this function in more detail in Section 8.2 and observe that they use a more liberal notion of less-strict than it is used in this thesis.

The notion of *boundedness* in the context of pretty printers (Wadler 2003) can also be considered as a kind of special case of on-line behavior. In the area of functional programming a pretty printer is a program or a library that takes a document that is represented by a tree structure and converts it into a string. The output has to reflect the structure that is represented by the tree structure of the document. Additionally, a pretty printer takes a width that restricts the number of characters in the string before each line break. A pretty printer is supposed to be optimal in the sense that it uses the smallest number of lines with respect to the given width, that is, line breaks are only added if necessary. Furthermore, a pretty printer is called bounded if it uses a look-ahead of at most w characters to pretty print a document with width w.

That is, boundedness aims for an implementation of pretty printers that inspects only a small part of its argument to yield a certain part of its result in the same way as the property of *permutations* and the more general term of on-line behavior. In other words, all these concepts aim for implementations whose strictness does not exceed a certain degree. In contrast, in this thesis we aim for implementations that are as little strict as possible. A function that is as little strict as possible is called minimally strict. For example, we can observe that *permutations* is actually minimally strict, at least for lists up to six element, by using the tool that is presented in this thesiss. For most functions we can only make a statement whether the function is minimally strict for a restricted number of arguments where we use a notion of size for the restriction. Sloth is only able to verify that *permutations* is as little strict as possible for lists up to six elements as the result of *permutations* has factorial many elements in the number of elements in the argument list. Thus, checking the result of each application is quite expensive simply because of the number of elements.

Finally, there are two case studies that use a heap profiling tool for a lazy functional programming language to improve the memory usage of a program. In both

cases, the memory usage of one of the considered functions is improved by using a less strict implementation. First, Runciman and Wakeling (1993a) have applied their profiling tool to check the memory usage of a compiler for the programming language Lazy ML, a lazy dialect of the strict functional programming language ML. In this case study they present an example of a function whose memory usage can be improved by using a less strict definition. However, they do not provide enough background information to relate this example to the results presented in this thesis. Furthermore, Runciman and Wakeling (1993b) have used the profiling tool to investigate a program that takes a propositional formula and yields a clausal normal form of this formula. We investigate this example in detail in Section 8.2 and observe that their notion of less-strict goes beyond the notion of less-strict considered in this thesis.

The related works considered so far make use of concepts that are similar to the idea of minimally strict functions. Besides these works the most important related work is a draft paper by Chitil (2006)[1]. He has presented the idea to automatically check whether a function is as little strict as possible. Furthermore, he has presented the idea of how to generally observe that a function is unnecessarily strict. The tool presented in this thesis basically still applies the same idea. Chitil (2006) has even presented a tool called StrictCheck that checks whether a function is as little strict as possible. However, StrictCheck proposes functions that are only implementable using parallel features like the ambiguous choice operator by McCarthy (1961). We can model this operator in Haskell using non-pure features, which functional programmers want to avoid most of the time. Ambiguous choice and non-pure features break referential transparency in the sense that we can no longer replace equals by equals.

Finally, Coutts et al. (2007), as a side-product, have developed a tool that is related to the concept of minimally strict functions and Sloth. They have developed a library for stream processing and implemented standard list functions on basis of this library. Similar to the *partition* example they have used specifications from the language report to check whether their implementations agree with the standard definitions of these functions. To compare the functions they have implemented a tool for checking whether two functions agree for all possible inputs up to a specific size similar to SmallCheck by Runciman et al. (2008). However, in contrast to SmallCheck they also consider that an argument might contain an error or a non-terminating expression. This approach circumvents the problem of parallel functions as two functions are compared with each other. Thus, if the function that acts as specification is not parallel, the tool will not propose a parallel implementation. Nevertheless, in most cases we do not have a specification that can be used to check a function and, furthermore, we still do not know whether the function can be improved with respect to strictness as the specification might be unnecessarily strict as well. For example, in Section 3.2 we observe that the implementation of the function *intersperse* as it is given in the language report is unnecessarily strict.

The rest of this thesis is organized as follows.

[1] An extended version is available as technical report (Chitil 2011).

- Chapter 2 introduces the main features of the lazy, purely functional programming language Haskell (Peyton Jones 2003). Furthermore, it provides a denotational semantics for a simple non-strict, first-order, functional programming language without polymorphism. This language is used as basis for a mathematical model of minimal strictness in Chapter 4.

- Chapter 3 illustrates the concept of non-strict evaluation and presents a practical example that backs the interest in less strict functions. More precisely, it presents an application whose memory usage can be improved by a factor of 20,000 by using a less strict implementation of a standard Haskell function, namely above-mentioned *intersperse*.

- Chapter 4 first revisits the approach by Chitil (2006). Furthermore, it presents a criterion to check whether a function is minimally strict and rule out parallel implementations by employing the notion of sequentiality. This chapter as well as the underlying denotational semantics only consider monomorphic functions. Finally, we prove that the presented criterion is necessary and sufficient for the existence of a less strict, sequential function.

- Chapter 5 presents the basics of a light-weight tool, that is, a Haskell library, called Sloth, that is based on the mathematical model of Chapter 4. In particular, the chapter illustrates certain approximations that are applied to gain a light-weight tool. Parts of the results of this chapter have already been presented elsewhere (Christiansen 2011).

- Chapter 6 generalizes the concept of minimally strict functions to polymorphic functions. We show how we can check whether a polymorphic function is minimally strict by checking a specific monomorphic instance of the polymorphic function. Furthermore, we prove that we can reduce the number of test cases of a polymorphic function significantly by employing this approach. The results of this chapter have already been presented (Christiansen and Seidel 2011).

- Chapter 7 presents some case studies of checking whether functions are minimally strict. More precisely, we consider Peano arithmetic, arithmetic for a representation of binary numbers by Braßel et al. (2008), and a Haskell library called split (Yorgey 2011). Besides, we also present an approach to derive less strict functions in certain cases and exemplarily discuss the connection between less strictness and space usage by means of the function *reverse* that reverses the order of the elements of a list.

Finally, we want to make some organizational remarks. All proofs that are omitted in Chapter 4 and Chapter 6 are presented in Appendix A and Appendix B, respectively. For all run-time and memory usage measurements we have used an Apple MacBook Pro 2.3GHz with an Intel Core i5 and 4GB RAM. When we benchmark Haskell programs, we use the Glasgow Haskell Compiler (GHC) version 7.0.3 in single core mode. At last, we want to make a remark about mathematical rigorousness. The mathematical model in Chapter 4 is based on the denotational semantics presented in Section 2.2. In contrast, Chapter 6 as well as Chapter 7 use a more

loose approach and do not strictly separate syntax and semantics. This approach is quite common in the context of functional programming as the semantics of a functional language is very closely related to the syntax of the language. We only have to take extra care to consider the cases that an expression yields an error or does not terminate. Chapter 6 gives more details about this approach to reasoning.

2 Preliminaries

This section presents some preliminaries for the remainder of this thesis. The first part of this section presents the main features of the functional programming language Haskell (Peyton Jones 2003). The reader who is familiar with Haskell may safely skip Section 2.1. In Section 7.3 and Section 8.2.2 we also give an outlook to applications of the presented approach to the functional logic programming language Curry (Hanus 2006), but we do not introduce the features of this language till we need them.

Section 2.2 presents a simplified functional programming language and provides a standard denotational model for this language. This denotational model is used in Chapter 4 as the basis for a mathematical model of minimally strict functions. The reader who is familiar with denotational semantics of non-strict functional languages may only inspect the syntax of the considered language presented in Figure 2.2.1 and note that we do not use arbitrary complete partial orders as domains but bounded complete, algebraic cpos. Definitions of these concepts are found in Section 2.2.

2.1 Introduction to Haskell

While in an imperative programming language the model of programming is based on destructively modifying the contents of memory cells, in a functional programming language the model of programming is based on the definition and application of functions. In other words, while in an imperative programming language a variable is a name for a location in the memory, in a functional programming language a variable is a shortname for a more complex expression. That is, while we can change the value that is assigned to a variable in an imperative language we cannot change it in a functional language. In this section we introduce the main features of the functional programming language Haskell. We start by showing how data is represented in Haskell, more precisely, we introduce the concept of *algebraic data types*.

2.1.1 Algebraic Data Types

A data type definition is introduced by the keyword **data** followed by the name of the algebraic data type, which has to start with a capital letter. One of the simplest algebraic data types, but a quite essential one is the type of Boolean values *Bool*.

data $Bool = False \mid True$

In the case of the data type of Boolean values the type is called *Bool*. The name is followed by an equality sign and an enumeration of the constructors of the type,

separated by vertical bars. In the case of *Bool* the constructors are called *False* and *True*. The name of a constructor has to start with a capital letter as well. A constructor is used to construct a value of the corresponding type. That is, the term *False* as well as the term *True* construct a corresponding value of type *Bool*.

While the constructors of *Bool* are both constants we can define constructors that take additional arguments. For example, the following data type *Partial* defines a constant *None* and an unary constructor *Some*, which takes an integer as argument.

$$\textbf{data}\ \mathit{Partial} = \mathit{None} \mid \mathit{Some}\ \mathit{Int}$$

Besides algebraic data types, Haskell provides primitive data types, for example, *Int* for integers, *Integer* for arbitrary precision integers, *Float* and *Double* for floating point numbers, and *Char* for characters. To construct a value of type *Partial* we can either use *None* or a term like *Some* 42, for example. That is, we apply a constructor to an argument by juxtaposition of the constructor name and the argument.

A data type like *Partial* can be used to represent the result of a function if the function is only defined for some inputs. For example, consider a function that takes some kind of mapping from integers to integers and looks up a key in this mapping. If the mapping does not contain a binding for a key, the result of the function is *None*. If the mapping contains a binding, the result is *Some i* where *i* is the integer that is associated with the provided key. Obviously, we potentially also need a data type like *Partial* where the *Some* constructor contains a Boolean value, for example, instead of an integer. As this data type would be very similar to *Partial*, Haskell provides a mechanism to abstract over the concrete data type whose values may be used for arguments of the *Some* constructor.

Instead of the data type *Partial* for integers and another data type for Boolean values, Haskell provides the following data type, called *Maybe*.

$$\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

The α in this equation is called *type variable* and denotes that the type *Maybe* takes a type as argument. Types that take a type as argument are called *type constructors*. Instead of the type *Partial* we can as well use the type *Maybe Int*, that is, we apply the type constructor *Maybe* to the type *Int*. This way we instantiate the type variable α with the type *Int* and get a type *Maybe Int*, whose values are *Just* applied to some integer and *Nothing*. For example, the term *Just* 42 is a value of type *Maybe Int*. In contrast, the term *Just False* constructs a value of type *Maybe Bool*.

Besides the simple data type definitions, considered so far, we can define more complex data types by employing recursion. The following algebraic data type defines a type of ordered collections of values of equal type, called lists.

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

For an arbitrary type τ we can construct an empty list of type *List* τ by using the term *Nil*. In the following we use the names α, β, γ for type variables and τ, τ_1, τ_2, τ_3 as meta-variables for concrete types. We can construct a non-empty list of type *List* τ by using the constructor *Cons* that takes a value of type τ and a list with

elements of type τ as second argument. For example, the term *Cons* 1 *Nil* constructs a list of integers, that is, a list of type *List Int*, that contains one element, namely 1. Furthermore, the term *Cons* 1 (*Cons* 2 *Nil*) constructs a list with the elements 1 and 2. Note that a list like *Cons* 1 (*Cons False Nil*) is not a value of any list type. The list *Cons False Nil* is an element of type *List Bool*. Therefore, *Cons* applied to this value may only take a value of type *Bool* as first argument as the constructor *Cons* states that its first argument has the same type as the elements of the list of its second argument.

As lists are a very commonly used data type in Haskell, the actual syntax of the list data type is prettier than the one above. The following definition is not valid Haskell syntax, but it illustrates the syntax of lists in Haskell. The constructors of this data type are $[\,]$ and ":", where the latter one is used infix.

$$\textbf{data}\ [\alpha] = [\,] \mid \alpha : [\alpha]$$

We can construct the empty list by $[\,]$ and a list of type $[Int]$ with the element 1 by $1 : [\,]$. Furthermore, a list with 1 and 2 is constructed by $1 : 2 : [\,]$. Note that the infix list constructor ":" is right associative, that is, the term $1 : 2 : [\,]$ stands for $1 : (2 : [\,])$. There is also a pretty syntax for the construction of lists. For example, the term $[False, True]$ denotes the list $False : True : [\,]$ and the term $[1, 2, 3, 4]$ denotes the list $1 : 2 : 3 : 4 : [\,]$.

Type constructors are not restricted to a single argument. The following algebraic data type defines a type of pairs by using a two-ary type constructor. Note that the name of the type may agree with the name of a constructor.

$$\textbf{data}\ Pair\ \alpha\ \beta = Pair\ \alpha\ \beta$$

The constructor *Pair* takes two arguments. The first argument is a value of the type that is passed as first argument to the type constructor *Pair*. The second argument of the constructor *Pair* is a value of the type that is passed as second argument to the type constructor *Pair*. For example, the term *Pair False* 42 constructs a pair of type *Pair Bool Int*.

Haskell uses a pretty syntax for pairs, too. In contrast to the list data type, pairs are not constructed by an infix constructor but by a mixfix constructor. Again, the following definition is not valid Haskell syntax but illustrates the syntax of pairs in Haskell.

$$\textbf{data}\ (\alpha, \beta) = (\alpha, \beta)$$

The pair that we have considered above is constructed by $(False, 42)$ and it is an element of the type $(Bool, Int)$.

2.1.2 Functions

As mentioned before, functions are quite essential in a functional programming language. In Haskell, functions are defined by means of *pattern matching*, where a pattern is a kind of inverse of a constructor. While we can use a constructor to assemble data, we can use a pattern to disassemble it again. As an example, we consider the following function, which disassembles a value of type *Maybe Int*.

$$fromJust\ (Just\ x) = x$$

A function definition starts with the name of the function, in this case *fromJust*. The first character of a function name has to be lower case to distinguish function names from constructor and type names. The name of a function is followed by a pattern. In the case of *fromJust* the pattern matches if the argument is a value constructed by using the constructor *Just*. All variables used in a pattern are bound to the corresponding value in the argument, the function is applied to. Note that we can distinguish variables from constructors because variable names start with a lower case letter. For example, if we apply *fromJust* to *Just* 42, then the variable x is bound to 42 and the result of this application is 42 as well.

The application of a function to an argument, like the application of a constructor to an argument, is denoted by juxtaposition of function name and argument. That is, $fromJust\ (Just\ 42)$ denotes the application of *fromJust* to the argument $Just\ 42$. If we apply *fromJust* to *Nothing*, we get a run-time error as *fromJust* does not provide a rule for this case.

To define a function that matches more than one constructor, we can use multiple rules, written below each other. For example, the following function defines the Boolean negation by using two rules, one for the case that the argument is *False* and one for the case that the argument is *True*.

$$\begin{aligned} not\ False &= True \\ not\ True\ &= False \end{aligned}$$

The rules of a function are processed from top to bottom. Furthermore, variables are also valid patterns. For example, we can as well define the Boolean negation as follows.

$$\begin{aligned} not\ False &= True \\ not\ b\ \ \ \ &= False \end{aligned}$$

Here the first rule matches if the argument is *False* and the second rule matches in all other cases, that is, if the argument is *True*. If a variable that is introduced by a pattern is not used on the right-hand side of the equality sign, as it is the case for the second rule of *not*, we can use an underscore instead of the variable in the pattern as well.

A function can also take multiple arguments where the arguments are separated by whitespaces. The following definition of *andL* defines the Boolean conjunction in Haskell.

$$\begin{aligned} andL\ False\ \_ &= False \\ andL\ True\ \ b\ &= b \end{aligned}$$

If the first argument of *andL* is *False*, the result of the function is *False* as well, independent of the second argument. If the first argument is *True*, then *andL* yields its second argument.

By means of this simple example we can already observe that there are often several ways to define the same basic behavior. For example, we can define the following implementation of the Boolean conjunction, called *andR*, which performs pattern matching on its second argument.

```
andR _ False = False
andR b True  = b
```

Besides, there is one implementation of the Boolean conjunction, called *and*, that explicitly lists all cases as follows.

```
and False False = False
and False True  = False
and True  False = False
and True  True  = True
```

Later we will observe that all these ways to define a Boolean conjunction lead to different behavior with respect to strictness.

The predefined Boolean conjunction in Haskell is an infix operator called "&&". The definition of an infix operator does not follow the general rule of a function definition. In contrast to a function definition, the name of an infix operator is written between its two arguments.

```
False && _ = False
True  && b = b
```

If we want to use an infix operator in the way as we can use a two-ary function, we enclose its name in parentheses. For example, this way we can define the infix operator (&&) by first stating its name.

```
(&&) False _ = False
(&&) True  b = b
```

In the same way as we can transform an infix operator into a normal, two-ary function by enclosing it in parentheses we can as well use a two-ary function as infix operator by enclosing it in quotes. For example, the term *False* `` `and` `` *True* applies the two-ary function *and* to the arguments *False* and *True*.

Besides pattern matching in function rules, we can use pattern matching in so-called **case** expressions. By using a **case** expression a function can behave differently depending on the value of an expression in the same way as a function rule specifies the behavior of a function depending on the argument of the function. The following definition is an alternative to the definition of the negation function by means of several rules.

```
not b =
  case b of
       False → True
       True  → False
```

The keyword **case** is followed by the expression over which we branch, in this case the variable b. This expression is also called the scrutinee of the **case** expression and it is itself followed by the keyword **of**. The keyword **of** is followed by pairs of pattern and expression, also called branches, where each pattern is separated by $\rightarrow$ from the corresponding expression[1]. Like function rules, the patterns of a **case** expression determine the right-hand side that is used, depending on the value of the scrutinee.

The branches of a **case** expression can be separated by semicolons and enclosed in braces to allow unambiguous definitions of cascaded **case** expressions. Alternatively, as used in the example above, we can use a so-called *offside rule* (Landin 1966). That is, the leftmost character of the first pattern introduces a column that is used to identify the next pattern. All tokens that begin in this column are considered as patterns and start a new branch. All tokens that begin to the right of this column are still assigned to the previous rule while all tokens that begin to the left of this column close the definition of the branches of the **case** expression. This way we can leave out the braces and semicolons in the definition of a **case** expression.

Haskell also provides an **if-then-else** expression that can be considered as syntactic sugar for a **case** expression that branches over a Boolean value. More precisely, the following semantic equivalence holds where we use braces and semicolons to explicitly separate branches instead of using the offside rule.

$$\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2 \equiv \textbf{case } b \textbf{ of } \{\mathit{True} \rightarrow e_1; \mathit{False} \rightarrow e_2\}$$

Note that here and in the following we always use the symbol $\equiv$ for semantic equivalence. In Section 2.2 we provide a formal definition of semantic equivalence for a simple functional language, which can be considered as a subset of Haskell.

Before we go on, we introduce two other syntactical constructs that are often used to define functions in Haskell. These constructs are called *guards* and **where** clause. Instead of using an equality sign in the definition of a function we can use a vertical bar to introduce a definition by means of guards. Guards are Boolean expressions that, similar to a **case** expression, are used to branch over the value of an expression. The vertical bar is followed by a Boolean expression and the expression is followed by an equality sign and the corresponding right-hand side. For example, the following definition implements the signum function that takes an arbitrary integer and yields -1 if the integer is negative, 0 if the integer is 0, and 1 if the integer is positive.

$$\begin{array}{l} \mathit{signum}\ i \\ \quad \mid i < 0 \quad\ = -1 \\ \quad \mid i == 0 \quad = 0 \\ \quad \mid \mathit{otherwise} = 1 \end{array}$$

Guards are processed from top to bottom. The operator $(<)$ is a two-ary Boolean predicate that checks whether an integer is strictly smaller than another integer and the operator $(==)$ checks whether two integers are equal. In Section 2.1.6 we will

[1] Here and in the following, we use the symbol $\rightarrow$ for the token ->.

observe that the operator $(==)$ actually does not only work for integers but for a much larger class of values. The constant *otherwise* is synonym for the Boolean constant *True*. Therefore, *signum* yields -1 if i is smaller than zero, 0 if i is equal to zero and 1 otherwise.

Finally, we want to present so-called **where** clauses. A **where** clause introduces local function and variable definitions. We can add a **where** clause to each rule of a function. The variables introduced by the patterns of the rule are visible in the right-hand sides of the definitions in a **where** clause. Functions and variables defined in a **where** clause are only visible in the right-hand side of the corresponding rule. For example, consider the following implementation of the *reverse* function that uses an accumulating parameter to achieve a linear complexity.

$$
\begin{array}{l}
reverse = rev\ [\,] \\
\quad \textbf{where} \\
\quad\quad rev\ xs\ [\,] \quad\quad = xs \\
\quad\quad rev\ xs\ (y : ys) = rev\ (y : xs)\ ys
\end{array}
$$

We define a local function called *rev* in the **where** clause. The function *rev* is neither visible in additional rules of *reverse* nor in other top-level definitions. By top-level definitions we denote function and variable definitions that are not defined within a **where** clause or a **let** expression.

Note that we call it a **where** *clause* and not an expression as it does not play the same role as an expression. For example, a **where** clause cannot be the argument of an application while an expression can. However, there is also a counterpart of the **where** clause in the family of expressions, called **let** expression. A **let** expression consists of two keywords **let** and **in**. Between these keywords we can define functions and variables in the same way as we would define them using a **where** clause. These functions and variables are only visible in the expression that follows the keyword **in**. We can define the *reverse* function by employing a **let** expression as follows.

$$
\begin{array}{l}
reverse = \\
\quad \textbf{let}\ rev\ xs\ [\,] \quad\quad = xs \\
\quad\quad\ \ rev\ xs\ (y : ys) = rev\ (y : xs)\ ys \\
\quad \textbf{in} \\
\quad rev\ [\,]
\end{array}
$$

Finally, note that the **where** clause as well as the **let** expression make use of offside rules similar to the offside rule of the **case** expression. We abstain from a more thorough explanation as we do not need a more precise understanding of offside rules in the following and refer the interested reader to the Haskell language report (Peyton Jones 2003).

2.1.3 Types

So far we have stated that each constructor belongs to a certain type. That is, we can assign a type to every constructor. However, we can even assign a type to every function and, moreover, to every expression in a Haskell program.

Haskell has a strong, static type system with type inference. The type system is said to be strong because there will be no type errors at run-time if we run a well-typed program. Sometimes people also refer to a language as strongly typed if there are no implicit coercions. For example, the expression $\mathit{False} + 2$ is not well-typed in Haskell because $(+)$ expects two values of type *Int*, and *False* is of type *Bool*. Note that there are typed languages that use an implicit coercion to apply $(+)$ to *False* and 2, for example, by considering the Boolean value *False* as the integer 0. The type system of Haskell is said to be static because the compiler determines at compile time whether a program is well-typed and rejects the program if it is not. In contrast, in a dynamically typed language like python we might get a type error when we execute the program. Finally, type inference means that the compiler is able to determine the type of an arbitrary expression (Damas and Milner 1982). Therefore, we do not have to annotate any types at all and the compiler is still able to check whether the program is well-typed. Nevertheless, for documentation purposes, we annotate a type to every top-level Haskell definition in the following.

We annotate a type to an expression by separating the expression and the type by two colons. For example, the term $\mathit{False} :: \mathit{Bool}$ annotates the type *Bool* to the constructor *False*. If we annotate a wrong type, for example, $\mathit{False} :: \mathit{Int}$, the compiler reports a type error.

We annotate a type to a function by preceding the function definition with the function name, two colons, and the type of the function. We annotate the type $\mathit{Bool} \to \mathit{Bool}$ to the function *not* as follows.

$$\begin{aligned}&\mathit{not} :: \mathit{Bool} \to \mathit{Bool}\\&\mathit{not}\ \mathit{False} = \mathit{True}\\&\mathit{not}\ \mathit{True}\ = \mathit{False}\end{aligned}$$

Here the right-arrow $\to$ is a two-ary type constructor that constructs a function type. That is, *not* is a function that takes a Boolean value as argument and yields a Boolean value as result. Consider the two-ary Boolean conjunction (&&) as another example.

$$\begin{aligned}&(\&\&) :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}\\&\mathit{False}\ \&\&\ \_ = \mathit{False}\\&\mathit{True}\ \ \&\&\ b = b\end{aligned}$$

The type of this function is $\mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$. More precisely, if a function takes more than one argument the additional arguments are separated by $\to$ from each other. For example, a function that takes three arguments of types τ_1, τ_2, and τ_3 has the type $\tau_1 \to \tau_2 \to \tau_3 \to \tau_4$ where τ_4 is the result type of the function. Note that the function arrow is right-associative, that is, $\tau_1 \to \tau_2 \to \tau_3$ stands for $\tau_1 \to (\tau_2 \to \tau_3)$. Because the function arrow is right-associative, function application is left-associative, thus, $f\ e_1\ e_2\ e_3$ stands for $((f\ e_1)\ e_2)\ e_3$.

2.1.4 Parametric Polymorphism

In fact, we have already introduced parametric polymorphism when we have talked about algebraic data types. Parametric polymorphism means that a data type or a

function is parametric over a type. For example, the list data type is parametric over the type of the elements of the list. However, parametric polymorphism is not restricted to data types but can also be used for function definitions. In the following we present several examples of polymorphic functions.

Many functions, especially functions over collection types like lists or trees, behave equally for different element types. For example, counting the number of elements in a list does not depend on the type of the elements in the list. Consider the following definition of the standard Haskell function *length* that takes a list and yields its length. Here and in the following, to avoid parentheses, we employ the convention that a function, written prefix, has a higher precedence than an infix operator. Hence, the expression $1 + \mathit{length}\ xs$ is short for $1 + (\mathit{length}\ xs)$.

$$\begin{aligned} &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (\_ : xs) &&= 1 + \mathit{length}\ xs \end{aligned}$$

Obviously, the function *length* has the type $[\mathit{Bool}] \to \mathit{Int}$ as it can calculate the length of a list of Boolean values. However, it seems equally valid to assign the type $[\mathit{Int}] \to \mathit{Int}$ to *length* as it can also calculate the length of a list of integers. And, indeed, both types are valid types for *length*. The compiler does not report a type error if we annotate one of these types to the function *length*. Nonetheless, there is a more general type in the sense that we can apply the *length* function to more arguments without risking the type checker to complain. In the same way as we have used type variables to express that a data type is parametric, we use type variables to express that *length* calculates the length of a list of an arbitrary element type. More precisely, we can annotate the type $[\alpha] \to \mathit{Int}$ to *length*.

In Haskell the type of a function is always implicitly quantified over all type variables that occur in it. Furthermore, the variables are quantified at the beginning of the type. For example, the type $[\alpha] \to \mathit{Int}$ stands for the type $\forall\alpha.[\alpha] \to \mathit{Int}$. It is just a question of convenience to leave out the quantifier. Though, note that it is quite important that all variables are quantified at the beginning of a type. For example, the type $\forall\alpha.[\alpha] \to \mathit{Int}$ and the type $[\forall\alpha.\alpha] \to \mathit{Int}$ mean totally different things. The latter is also referred to as a higher-rank type (Odersky and Läufer 1996). The former type denotes a function that takes a list of an arbitrary type τ and yields an integer or, more precisely, we can choose some type τ and get a function of type $[\tau] \to \mathit{Int}$. In contrast, the latter type denotes a function that takes a list of polymorphic values as argument and yields an integer. That is, if we assign the type $[\forall\alpha.\alpha] \to \mathit{Int}$ to *length* we can only apply it to lists where every element has the type $\forall\alpha.\alpha$. For example, this is not the case for a list of Boolean values. Although Haskell provides extensions to higher-rank types (Peyton Jones et al. 2007), in the following we only consider rank-1 polymorphism. In other words, all type variables of function types are implicitly quantified at the beginning of the type definition.

A more sophisticated example of a polymorphic function is the list concatenation $(+\!\!+)$. In this case we employ a type variable to state that the element type of the result list is the same as the element type of the argument list. Furthermore, as both arguments are of type $[\alpha]$, we know that both argument lists have to have the same element type, so we cannot concatenate a list of Boolean values with a list of integers.

$$
\begin{aligned}
&(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha] \\
&[\,] \quad +\!\!+ ys = ys \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

Note that we cannot define a concatenation function that takes one list of type $[\alpha]$ and one list of type $[\beta]$ because of the definition of the list data type. More precisely, the application of the list constructor $(:)$ to x and the result of $xs +\!\!+ ys$ in the second rule of $(+\!\!+)$ implies that x has the same type as the elements of the result of $(+\!\!+)$. Furthermore, the first rule of $(+\!\!+)$ implies that the list ys has the same type as the result of $(+\!\!+)$.

We can as well use multiple type variables in a function type to specify more complex relationships. For example, the following function takes two lists of potentially different element types and combines these lists point-wise to a list of pairs.

$$
\begin{aligned}
&zip :: [\alpha] \to [\beta] \to [(\alpha, \beta)] \\
&zip\ (x : xs)\ (y : ys) = (x, y) : zip\ xs\ ys \\
&zip\ \_ \quad \_ \quad = [\,]
\end{aligned}
$$

By employing type variables we express that the elements of the first list have the same type as the values of the first component of the pair and the elements of the second list have the same type as the values of the second component of the pair. Note that we could as well annotate the type $[\alpha] \to [\alpha] \to [(\alpha, \alpha)]$ to *zip*, but in this case the function is less general. For example, while we can apply *zip* to the lists $[False]$ and $[42]$ this application would not be well-typed if we annotate the less general type because the less general type only allows argument lists of equal type.

Finally, we want to introduce a function that, in particular, is intensively used in Chapter 6. The list indexing function $(!!) :: [\alpha] \to Int \to \alpha$ takes a list and an index in the list (starting with zero) and yields the element at the corresponding position.

$$
\begin{aligned}
&(!!) :: [\alpha] \to Int \to \alpha \\
&[\,] \quad !!\ \_ \quad = indexError \\
&(x : xs)\ !!\ i \\
&\quad \mid i < 0 \quad = indexError \\
&\quad \mid i == 0 \quad = x \\
&\quad \mid otherwise = xs\ !!\ (i - 1)
\end{aligned}
$$

If the list is empty or the index is smaller than zero, the function yields a run-time error denoted by *indexError*. Using a guard we distinguish the cases that the index i is smaller zero, equal to zero, or greater than zero. In the second case the first element of the given list is the element we are looking for and in the third case we have to project to position $i - 1$ of the list without the first element, denoted by xs.

2.1.5 Higher-Order

In Haskell, functions are first class citizens. That means a function can be used, like any other data type, as argument or result of a function or even as argument of a constructor.

As we have observed before, the function arrow $\rightarrow$ is right associative. Therefore, the type of a two-ary function $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, in fact, stands for the type $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. That is, if we apply the function to a value of type τ_1 we get a function of type $\tau_2 \rightarrow \tau_3$. We can even partially apply any function that takes more than one argument. If we apply a function that takes n arguments to m arguments, where m is smaller than n, we get a function that takes another $n - m$ arguments. For example, the term *and True* denotes a partial application of the function *and* to the argument *True*. More precisely, the term *and True* denotes a function of type $Bool \rightarrow Bool$ that is semantically equivalent to the identity function of type $Bool \rightarrow Bool$. In the same way we can apply a function of type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$ to a value of type τ_1 and get a function of type $\tau_2 \rightarrow \tau_3 \rightarrow \tau_4$. If we, again, apply the resulting function to a value of type τ_2, we get a function of type $\tau_3 \rightarrow \tau_4$.

To partially apply an infix operator we can use so-called left and right sections. We enclose an infix operator together with one argument in parentheses and indicate by the position of the argument to the left or to the right of the infix operator if we partially apply the operator to its first or its second argument. For example, the expressions (*True* &&) and (&& *True*) partially apply (&&) to *True* and the resulting function still awaits its second and first argument, respectively.

As mentioned before, functions cannot only yield functions as result but also take functions as argument. Consider the following function definition that takes a predicate and a list and removes all elements from the list that do not satisfy the predicate. Here, the variable *pred* stands for a function of type $\tau \rightarrow Bool$, and, therefore, we can apply it to the list element x. When we apply *pred* to x, we get a Boolean value as result. Thus, we can use a guard to branch over the result of this application. Note that we may not leave out the parentheses in the type of *filter* this time as the function arrow is right-associative.

$$
\begin{array}{l}
filter :: (\alpha \rightarrow Bool) \rightarrow [\alpha] \rightarrow [\alpha] \\
filter\ \_ \quad [\,] \quad = [\,] \\
filter\ pred\ (x : xs) \\
\quad \mid pred\ x \quad\ = x : filter\ pred\ xs \\
\quad \mid otherwise = filter\ pred\ xs
\end{array}
$$

A mathematically natural higher-order function is the function composition $(\circ)$. The function composition takes a function of type $\tau_2 \rightarrow \tau_3$, a function of type $\tau_1 \rightarrow \tau_2$ and a value of type τ_1 and applies these functions one after the other to the value.

$$
\begin{array}{l}
(\circ) :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\
(f \circ g)\ x = f\ (g\ x)
\end{array}
$$

Note that this definition shows how we can define an infix operator that takes more than two arguments by using parentheses. As an example, the term $not \circ not$ is the identity function of type $Bool \rightarrow Bool$ as it takes a Boolean argument and applies *not* twice to this argument.

Higher-order functions are a means to define abstractions that can be employed in many contexts. The function *map* is one example for a frequently used abstraction

of this kind. It takes a function as higher-order argument and applies this function point-wise to all elements of a list.

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

As an example, we can apply *map* to *not* and the list $[False, True, True]$ and get the list $[True, False, False]$ as result.

Higher-order functions can be used to generalize first-order functions. For example, by employing a higher-order function we can generalize the definition of *zip*, presented in the previous section. Instead of constructing pairs from the elements of the argument lists, we apply an arbitrary two-ary function to each pair of elements. This two-ary function is provided in the form of a higher-order argument.

$$\begin{aligned}
&zipWith :: (\alpha \to \beta \to \gamma) \to [\alpha] \to [\beta] \to [\gamma] \\
&zipWith\ f\ \ (x : xs)\ (y : ys) = f\ x\ y : zipWith\ f\ xs\ ys \\
&zipWith\ _\ _\ \quad\ _\ \quad\ = [\,]
\end{aligned}$$

Note that we have $zip \equiv zipWith\ (,)$ where $(,)$ denotes the two-ary function that takes two arguments and constructs a pair.

In the context of higher-order functions, it is often useful to define a function without having to introduce a name for the function. Therefore, Haskell provides the possibility to define anonymous functions. An anonymous function is introduced by the symbol λ followed by patterns that are separated by whitespaces.[2] The patterns are followed by a right arrow and the right-hand side of the anonymous function. For example, the expression $\lambda x\ xs \to xs +\!\!+ [x]$ defines an anonymous function that takes an element x and a list xs and appends the element at the end of the list. As an example of the application of an anonymous function, the application $filter\ (\lambda i \to (i > 0)\ \&\&\ (i < 10))$ defines a function on lists of integers that removes all elements from the list that are smaller than one or greater than nine. Note that *filter* takes over the elements of the list for which the predicate is satisfied.

By using higher-order functions we can even define control structures that usually are built-in in other languages. For example, the following higher-order function is a functional implementation of the while loop, which is known from imperative programming languages. The first argument of *while* is a function that takes the current "state" of the loop and yields a Boolean value that determines whether the loop is executed once more. The second argument of *while* is a state transformation that is applied in every execution of the body to get a new state from the old one.

$$\begin{aligned}
&while :: (\alpha \to Bool) \to (\alpha \to \alpha) \to \alpha \to \alpha \\
&while\ pred\ trans\ x \\
&\quad \mid pred\ x \quad\ = while\ pred\ trans\ (trans\ x) \\
&\quad \mid otherwise = x
\end{aligned}$$

As an example, we can use $while\ (\lambda(c, \_) \to c > 0)\ (\lambda(c, f) \to (c - 1, c * f))\ (5, 1)$ to calculate the factorial of 5. More precisely, this application yields the pair $(0, 120)$

[2] In Haskell programs the symbol λ is, in fact, represented by a backslash.

where the first component represents a counter that is decreased in every execution of the loop body while the second component is the factorial of the initial counter, in our example 5. The state transformation $(\lambda(c,f) \rightarrow (c-1, c*f))$ takes a counter and the current factorial, multiplies the factorial with the current counter, and decreases the counter.

A last comment about higher-order functions is in order to illustrate their power. If we consider a language that does only provide primitive recursive functions, we get an expressive power that is beyond primitive recursion if we add higher-order functions to the language (Hutton 1999). For example, we can even define the Ackermann function in a language with primitive recursion and higher-order functions.

2.1.6 Type Classes

Type classes (Wadler and Blott 1989) are an implementation of ad-hoc polymorphism, in the context of object-oriented programming languages also called overloading, in Haskell. Like a parametric polymorphic function can be applied to arguments of different types, an ad-hoc polymorphic function can be applied to arguments of different types as well. However, while a parametric polymorphic function behaves equally no matter which instance of the polymorphic function we consider, an ad-hoc polymorphic function, in most cases, behaves differently for different argument types. That is, we can provide several implementations of the same function that behave differently for different types. When we apply an overloaded function, the types of the arguments determine the concrete implementation that is used.[3]

Let us consider a function that tests two values for equality. Obviously, we cannot provide an equality check that works for arguments of different types. For example, we would like to implement an infix operator $(==_{Bool})$ that is able to check two Boolean values for equality as follows.

$$
\begin{array}{l}
(==_{Bool}) :: Bool \rightarrow Bool \rightarrow Bool \\
False \;==_{Bool}\; False = True \\
True \;==_{Bool}\; True \;= True \\
\_ \quad ==_{Bool}\; \_ \quad = False
\end{array}
$$

In the same way as we have defined a function that checks Booleans for equality, we would like to define the following infix operator $(==_{[Bool]})$ that checks two lists of Boolean values for equality.

$$
\begin{array}{l}
(==_{[Bool]}) :: [Bool] \rightarrow [Bool] \rightarrow Bool \\
[\,] \quad\quad ==_{[Bool]}\; [\,] \quad\quad = True \\
(b1 : bs1)\; ==_{[Bool]}\; (b2 : bs2) = (b1 \;==_{Bool}\; b2)\;\&\&\;(bs1 \;==_{[Bool]}\; bs2) \\
\_ \quad\quad ==_{[Bool]}\; \_ \quad\quad = False
\end{array}
$$

In Haskell a type class called *Eq* is used to implement equality checks like $(==_{Bool})$ and $(==_{[Bool]})$ and use the same name for both functions. The predefined type class *Eq* provides a function $(==)$ that implements an equality check.[4]

[3] In Haskell we can even choose the implementation depending on the result type of the function.

[4] In fact, *Eq* also provides an inequality check $(/=) :: \alpha \rightarrow \alpha \rightarrow Bool$ that is left out for simplicity.

```
class Eq α where
  (==) :: α → α → Bool
```

The keyword **class** starts the definition of a type class, followed by the name of the class. Like constructors and types, the names of type classes have to start with a capital letter. The name of the class is followed by a type variable that denotes the type that will be an instance of this type class. After the keyword **where** we enumerate the functions that have to be implemented to make a type an instance of this type class. In the case of the type class *Eq*, a type τ becomes an instance of the class if we provide a function $(==)$ of type $\tau \to \tau \to Bool$. For example, the following code implements an instance of *Eq* for the type *Bool*.

```
instance Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False
```

To make a type an instance of a type class we use the keyword **instance**. The name of the type class is followed by the specific type that becomes an instance of the type class, in this case *Bool*. The type is followed by the keyword **where** and by implementations of the functions of the type class, in this case $(==)$.

If we reconsider the equality check for lists of Booleans, we observe a more general structure. We define the equality check of lists by testing the elements of the list point-wise for equality. The equality check for lists of integers will look very similar to the equality check for lists of Booleans. We just have to replace all occurrences of $(==_{Bool})$ by an equality check for integers. That is, instead of using a specific instance to check the equality of the elements (in the case of $(==_{[Bool]})$ the specific instance is $(==_{Bool})$) we would like to check the elements with the equality check of the element type of the list. We can use a *type class constraint* to define an instance like this. Using a type class constraint we make all types that satisfy a certain precondition an instance of the type class. The following instance makes all types of the form $[\tau]$ an instance of *Eq*, if the type τ is an instance of *Eq*.

```
instance Eq α ⇒ Eq [α] where
  []     == []     = True
  (x:xs) == (y:ys) = (x == y) && (xs == ys)
  _      == _      = False
```

The applications of $(==)$ on the right-hand side are, in fact, applications of different functions. More precisely, while the left application $x == y$ invokes a function of type $\tau \to \tau \to Bool$, the application $xs == ys$ invokes a function of the type $[\tau] \to [\tau] \to Bool$. Note that, although we assume that these functions implement equality checks on the corresponding types, in fact, we do not know anything about these functions except for their types and their name.

Type class constraints are also used to restrict the arguments of certain functions. For example, consider that we want to define a function that takes a value and a list and checks whether the value is an element of the list. We obviously need an

equality test on the element type of the list to implement this function. That is, we want to define a function of type $\alpha \to [\alpha] \to Bool$ but we do not want to instantiate α with arbitrary types τ but only with types that are instances of *Eq*. We can express this circumstance by adding the constraint $Eq\ \alpha$ to the front of the type signature. The following definition of the membership test uses the Boolean disjunction implemented by the infix operator $(||)$.

$$\begin{aligned} &elem :: Eq\ \alpha \Rightarrow \alpha \to [\alpha] \to Bool \\ &elem\ \_\ [\,] \quad\ = False \\ &elem\ x\ (y : ys) = (x == y)\ ||\ elem\ x\ ys \end{aligned}$$

If we omit the type class constraint in the type of *elem*, the compiler even reports a type error. By the use of $(==)$, the compiler knows that x and y must be values of a type that is an instance of *Eq*.

We cannot only define type classes whose instances are types, but we can also define type classes whose instances are type constructors. This kind of classes is called type constructor classes (Jones 1993). For example, the following type class *Functor* is a quite prominent type constructor class.

$$\begin{aligned} &\textbf{class}\ Functor\ \varphi\ \textbf{where} \\ &\quad fmap :: (\alpha \to \beta) \to \varphi\ \alpha \to \varphi\ \beta \end{aligned}$$

To make a type τ an instance of *Functor*, the type has to be a type constructor because the variable φ is applied to a type, in this case type variables α and β. The type class *Functor* provides a function called *fmap*, which is a generalization of *map*. The function *fmap* takes a function from some type to another type and a value of some type constructor applied to the argument type of the function. Typically, an instance of *Functor* is some kind of collection that is parametric over the type of the elements, for example, a list, a tree, or some abstract data type like a set. In this case *fmap* can be considered as a function that applies its argument f to every member of the collection.

In the case of the list type constructor the function *fmap* is equivalent to the function *map*, presented before.

$$\begin{aligned} &\textbf{instance}\ Functor\ [\,]\ \textbf{where} \\ &\quad fmap\ \_\ [\,] \quad\ = [\,] \\ &\quad fmap\ f\ \ (x : xs) = f\ x : fmap\ f\ xs \end{aligned}$$

Note that the token $[\,]$ in the first line of the definition denotes a type constructor that takes the element type of a list as argument and constructs the type of lists with this element type. In contrast, the occurrences of the token $[\,]$ in the second line of the definition denote the pattern and the constructor for the empty list, respectively.

We can also regard a tuple as a collection with respect to one of its components. That is, for some type τ we can define an instance of *Functor* for the partially applied type constructor $((,)\ \tau)$ as follows. In this case we have to provide a function *fmap* of type $(\alpha \to \beta) \to (\tau, \alpha) \to (\tau, \beta)$. The following code implements an instance of *Functor* for $((,)\ \tau)$ where τ is an arbitrary type.

```
instance Functor ((,) α) where
  fmap f (x,y) = (x,f y)
```

As mentioned before, instances of *Functor* are typically collections, but they don't have to be. For example, consider the two-ary type constructor $\rightarrow$ that constructs function types. If we partially apply $\rightarrow$ to some type τ, denoted by $((\rightarrow)\ \tau)$, the resulting unary type constructor takes the result type of the function type as argument. To make this type constructor an instance of the type class *Functor*, we have to provide an implementation of *fmap* of type $(\alpha \rightarrow \beta) \rightarrow (\tau \rightarrow \alpha) \rightarrow (\tau \rightarrow \beta)$. This type is an instance of the type of the function composition $(\circ)$. Therefore, we can define an instance of *Functor* for a partially applied function type as follows.

```
instance Functor ((→) α) where
  fmap f g = f ∘ g
```

Nobody restrains us from defining "silly" instances of *Functor*. For example, the following implementation is also a valid implementation of *fmap* for the list type constructor. This instance ignores its arguments and yields the empty list.

```
instance Functor [] where
  fmap _ _ = []
```

The type class *Functor* is inspired by a concept with the same name in category theory. There, a functor is a homomorphism, that is, a structure preserving mapping, between two categories. More precisely, a functor is a mapping F from a category C to a category D that assigns every object $X \in C$ an object $F(X) \in D$ and every morphism $f : X \rightarrow Y \in C$ a morphism $F(f) : F(X) \rightarrow F(Y) \in D$. Furthermore, F has to satisfy the following laws where $\circ$ denotes morphism composition.

$$F(id) = id$$
$$F(f \circ g) = F(f) \circ F(g)$$

In contrast, in Haskell the functor F is split into two parts, the type constructor that maps an object (in the case of Haskell a type) to an object and the function *fmap* that maps a morphism (in the case of Haskell a function) to a morphism. Moreover, for all instances of *Functor*, the function *fmap* is supposed to satisfy the following laws.

$$fmap\ id \equiv id$$
$$fmap\ (f \circ g) \equiv fmap\ f \circ fmap\ g$$

For example, our "silly" list instance breaks the first law. However, note that Haskell does not check whether an instance of *Functor* satisfies these laws.

2.2 Denotation of a Simple Functional Language

In this section we define a denotational semantics for a simple, non-strict, first-order, monomorphic functional language. This semantics is used in Chapter 4 to present a

$$
\begin{array}{lcl}
\tau & ::= & \mathit{Bool} \\
 & | & \mathit{List}\ \tau \\
\sigma & ::= & \tau_1 \times \cdots \times \tau_n \\
P & ::= & D\ P \\
 & | & \epsilon \\
D & ::= & f :: \sigma \to \tau; f\langle x_1, \ldots, x_n\rangle = e \\
e & ::= & x \\
 & | & \mathit{False} \\
 & | & \mathit{True} \\
 & | & \mathit{Nil}_\tau \\
 & | & \mathit{Cons}\langle e_1, e_2\rangle \\
 & | & f\langle e_1, \ldots, e_n\rangle \\
 & | & \textbf{case}\ e\ \textbf{of}\ \{\mathit{False} \to e_1; \mathit{True} \to e_2\} \\
 & | & \textbf{case}\ e\ \textbf{of}\ \{\mathit{Nil}_\tau \to e_1; \mathit{Cons}\langle x, xs\rangle \to e_2\} \\
 & | & \mathit{undefined}_\tau
\end{array}
$$

Figure 2.2.1: Syntax of a Simple First Order Functional Language

mathematical model of minimally strict functions. Figure 2.2.1 presents the syntax of the considered language. For simplicity we only consider Booleans and lists as data types, but the results in Chapter 4 hold for arbitrary algebraic first-order data types. The language presented here is similar to languages considered elsewhere. For example, Plotkin (1977) presents a simple functional language called PCF (programming language for computable functions). In contrast to the language considered here PCF does not provide lists but provides integers instead, and PCF uses an explicit fixpoint combinator while we use a system of recursive function definitions.

Here and in the following, by τ we denote a type that is either *Bool* or a list type whose argument type is a type of the form τ again. By σ we denote tuple types of arbitrary arity whose components are types of the form τ. Tuple types only occur as argument types of functions and constructors.

A program P is a sequence of declarations D where ϵ represents the empty program. Furthermore, we use an uncurried notation for functions, that is, a function f takes a tuple $\langle e_1, \ldots, e_n\rangle$ as argument. We do not use Haskell's curried notation as we do not consider higher-order functions and some of the following definitions are simplified by an uncurried notation. Function application is denoted by juxtaposition, for example, $f\langle e_1, \ldots, e_n\rangle$. We identify $f\langle\rangle$ with f and $f\langle x\rangle$ with $f\ x$. Instead of using semicolons to separate type annotation and function definition as well as braces and semicolons to separate case branches, for simplicity, we employ Haskell-like offside rules (Landin 1966). For convenience, we add an expression $\mathit{undefined}_\tau$ to the considered language that denotes a run-time error and is used to model partially defined case expressions.

We assume that the reader is familiar with the basic concepts of partially ordered sets and complete partial orders. More precisely, we consider the concepts of chain-complete partial orders and directed set complete partial orders and use these terms synonymously as these two concepts are equivalent. In the following we use the abbreviation cpo for complete partial order. Furthermore, by $\bigsqcap$ and $\bigsqcup$ we denote the greatest lower bound (infimum) and the least upper bound (supremum) with respect to a corresponding ordering denoted by $\sqsubseteq$ and $\sqsupseteq$, respectively.

Instead of complete partial orders we interpret types by bounded complete, algebraic complete partial orders, also called Scott domains. Scott et al. (1989) show that flat cpos are bounded complete and algebraic, and that all standard domain constructions — which we employ here — preserve these properties.

A cpo is bounded complete if all subsets that have an upper bound also have a least upper bound. A bounded complete cpo is also a complete meet-semilattice, that is, the infima of all non-empty subsets exist. We can prove this statement by defining the infimum of a set as the supremum of all lower bounds of the set. This supremum exists because the cpo is bounded complete.

To establish algebraic cpos, we have to introduce the concept of *finite* (sometimes also called *compact*) elements. We prefer the term finite here because in the considered setting the informal notion of finiteness like in "finite list", for example, coincides with the following definition of finite elements.

Definition 2.2.1 (Finite Element): An element x of a cpo $(D, \sqsubseteq)$ is called *finite* if for all chains $\langle x_i \rangle_{i \in I}$ in D with $x \sqsubseteq \bigsqcup_{i \in I} x_i$ there exists $i \in I$ such that $x \sqsubseteq x_i$. □

For any set S the powerset 2^S together with the subset ordering $\subseteq$ is a cpo. The smallest element of a powerset cpo is the empty set, and the supremum of a chain of sets is given by the union of these sets. In the powerset cpo the finite elements are exactly the finite sets.

To prove the first direction, consider an infinite set X. Furthermore, consider a chain $\langle X_i \rangle_{i \in \mathbb{N}}$ of finite subsets of X such that $|X_i| = i$. The supremum of this chain is the infinite set X, namely, $\bigcup_{i \in \mathbb{N}} X_i = X$. As there is no set X_i such that $X \subseteq X_i$, the infinite set X is not a finite element of $(2^S, \subseteq)$.

To prove the missing implication we proceed as follows. If X is not finite, then there exists a chain $\langle X_i \rangle_{i \in I}$ such that $X \subseteq \bigcup_{i \in I} X_i$, and for all $i \in I$ we have $X \not\subseteq X_i$. If the index-set I was finite, then the chain $\langle X_i \rangle_{i \in I}$ would have a greatest element, which is a contradiction. Thus, the index-set I is infinite. Furthermore, as for every $i \in I$ there exists $x \in X$ such that $x \notin X_i$ the set X is infinite as well.

In an algebraic cpo every element x is the supremum of the finite elements below it. That is, if $(D, \sqsubseteq)$ is an algebraic cpo, then for all $x \in D$ we have

$$x = \bigsqcup \{y \mid y \in D, y \sqsubseteq x, y \text{ finite}\}.$$

For example, the powerset cpo is an algebraic cpo as every (especially every infinite) set is the supremum (in this case the union) of its finite subsets. In an algebraic cpo the result of a (continuous) function for a non-finite element is the supremum of the results for the finite elements below. In other words, the behavior of functions for non-finite arguments is determined by the behavior for finite arguments. Note that

$$
\begin{aligned}
[\![\sigma \to \tau]\!] &= \{f \mid f \in [\![\sigma]\!] \to [\![\tau]\!], f \text{ continuous}\} \\
[\![\tau_1 \times \cdots \times \tau_n]\!] &= \{\langle x_1, \ldots, x_n\rangle \mid x_1 \in [\![\tau_1]\!], \ldots, x_n \in [\![\tau_n]\!]\} \\
[\![\mathit{Bool}]\!] &= \{\bot, \mathit{False}, \mathit{True}\} \\
[\![\mathit{List}\ \tau]\!] &= \{\bot, \mathit{Nil}_\tau\} \cup \{\mathit{Cons}\ \langle x, xs\rangle \mid x \in [\![\tau]\!], xs \in [\![\mathit{List}\ \tau]\!]\}
\end{aligned}
$$

Figure 2.2.2: Type Semantics

this characteristic of algebraic cpos is closely related to the behavior of functions in a functional programming language. In a functional programming language we cannot explicitly check whether the argument of a function is an infinite data structure, and, therefore, the behavior of this function for infinite arguments is determined by its behavior for finite arguments. This illustrates that we can use algebraic cpos to model the semantics of a functional programming language.

Figure 2.2.2 presents the interpretation of the types we are considering. The function $[\![\cdot]\!]$ denotes type semantics as well as term semantics. We consider a standard type semantics, that is, we interpret functions as continuous mappings with a point-wise ordering. Here continuity means Scott-continuity, namely, monotonicity and preservation of suprema of chains. Functions are ordered point-wise, that means, we have $f \sqsubseteq g$ if and only if $f\ x \sqsubseteq g\ x$ for all x of appropriate type.

As we consider a first-order language, all functions take a single argument. We denote mathematical function application by juxtaposition, that is, by terms of the form $f\ x$ where f is a function and x is an argument. The argument type of a syntactic function has the form $\tau_1 \times \cdots \times \tau_n$ and is interpreted as non-lifted direct product. The elements of $[\![\tau_1 \times \cdots \times \tau_n]\!]$ are denoted by $\langle e_1, \ldots, e_n\rangle$.

Because it is determined by the context, we use the symbol $\bot$ for all least elements and mostly do not distinguish least elements of different domains, for example, $\bot_{\mathit{Bool}}$ and $\bot_{\mathit{List\ Bool}}$. The type of Boolean values is interpreted by the three-element flat ordered set $\{\bot, \mathit{False}, \mathit{True}\}$ where $\bot$ is the least element. Note that we use the terms *False* and *True* for syntactic as well as for semantic objects.

The interpretation of $\mathit{List}\ \tau$ for some type τ contains terms over the constant Nil_τ that represents an empty list of type τ and the two-ary constructor $\mathit{Cons}\ \langle\cdot,\cdot\rangle$ for non-empty lists. The first argument of $\mathit{Cons}\ \langle\cdot,\cdot\rangle$ is an element of the interpretation of τ while the second argument is an element of the interpretation of $\mathit{List}\ \tau$ again. These terms are ordered component-wise, that is, for elements t and t' of $[\![\mathit{List}\ \tau]\!]$ we have $t \sqsubseteq t'$ if we get t' by replacing subterms in t that are $\bot$ by arbitrary values of appropriate type. Figure 2.2.3 illustrates the ordering of some of the elements of $[\![\mathit{List}\ \tau]\!]$, where x and y are elements of the interpretation of τ.

The set of terms over *Nil* and $\mathit{Cons}\ \langle\cdot,\cdot\rangle$ and the interpretation of τ together with the component-wise ordering does not form a cpo. For example, consider the following infinite chain

$$\bot, \mathit{Cons}\ \langle x, \bot\rangle, \mathit{Cons}\ \langle x, \mathit{Cons}\ \langle x, \bot\rangle\rangle, \mathit{Cons}\ \langle x, \mathit{Cons}\ \langle x, \mathit{Cons}\ \langle x, \bot\rangle\rangle\rangle, \ldots$$

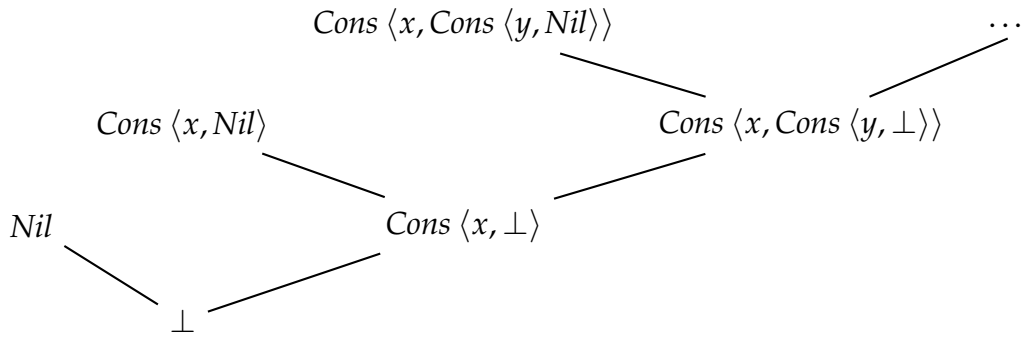


Figure 2.2.3: Ordering of Finite Lists

for some element x of the interpretation of τ. The supremum of this chain would be an infinite term. Therefore, we have to add representations for the infinite lists to $[\![List\ \tau]\!]$. We consider the least fixpoint of the recursive definition of $[\![List\ \tau]\!]$ that constitutes a cpo. Schmidt (1987) uses an inverse limit construction to show that such a least fixpoint exists. An element of $[\![List\ \tau]\!]$ is represented by an infinite tuple of different approximations of the element itself. For example, the list $Cons\ \langle x, Cons\ \langle x, \bot\rangle\rangle$ is represented by the infinite tuple

$$(\bot, Cons\ \langle x, \bot\rangle, Cons\ \langle x, Cons\ \langle x, \bot\rangle\rangle, Cons\ \langle x, Cons\ \langle x, \bot\rangle\rangle, \dots)$$

where the dots indicate that all following tuple components are $Cons\ \langle x, Cons\ \langle x, \bot\rangle\rangle$ as well. More precisely, we represent a finite list by an infinite tuple that becomes stationary at some position of the tuple. The components of the tuple contain approximations of the original lists that become more precise. That is, each consecutive pair of components of the tuple contains lists that are related by $\sqsubseteq$. The component where the tuple becomes stationary contains the original list. An infinite list is represented by an infinite tuple that contains approximations of the infinite list.

The inverse limit construction does not yield a solution to the recursive equation for the list type presented in Figure 2.2.2. However, we get a solution that is order-isomorphic to a solution of the concrete recursive definition. Therefore, in the following we consider the domain $[\![List\ \tau]\!]$ as the set of finite and infinite terms over Nil, $Cons\ \langle\cdot,\cdot\rangle$, and the interpretation of the content type τ. Schmidt (1987) has presented a detailed construction of solutions to recursive domain equations.

Figure 2.2.4 presents the term semantics for the language shown in Figure 2.2.1. The semantics of a variable x is its binding in the environment a. An environment is a mapping from the set of variables to semantic objects. By $a[x \mapsto v]$ we denote the modification of the environment a such that $a[x \mapsto v](x) = v$ and $a[x \mapsto v](y) = a(y)$ if $x \neq y$. Furthermore, we denote the modification of multiple variable bindings by $a[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$. If the semantics of an expression e is independent of the environment a, in other words, e has no free variables, then we use $[\![e]\!]$ instead of $[\![e]\!]_a$.

$$
\begin{aligned}
[\![x]\!]_a &= a(x) \\
[\![False]\!]_a &= False \\
[\![True]\!]_a &= True \\
[\![Nil_\tau]\!]_a &= Nil \\
[\![Cons\langle e_1, e_2\rangle]\!]_a &= Cons\ [\![\langle e_1, e_2\rangle]\!]_a \\
[\![f\langle e_1, \ldots, e_n\rangle]\!]_a &= [\![f]\!]\ [\![\langle e_1, \ldots, e_n\rangle]\!]_a \\
[\![\langle e_1, \ldots, e_n\rangle]\!]_a &= \langle [\![e_1]\!]_a, \ldots, [\![e_n]\!]_a\rangle \\
[\![f]\!]_a &= \lambda\langle v_1, \ldots, v_n\rangle.\ [\![e]\!]_{a[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]} \\
&\text{if } f :: \tau_1 \times \cdots \times \tau_n \to \tau; f\langle x_1, \ldots, x_n\rangle = e \in P \\
[\![\textbf{case } e \textbf{ of } \{False \to e_1; True \to e_2\}]\!]_a &= \begin{cases} \bot & \text{if } [\![e]\!]_a = \bot \\ [\![e_1]\!]_a & \text{if } [\![e]\!]_a = False \\ [\![e_2]\!]_a & \text{if } [\![e]\!]_a = True \end{cases} \\
[\![\textbf{case } e \textbf{ of } \{Nil_\tau \to e_1; Cons\langle x, xs\rangle \to e_2\}]\!]_a &= \begin{cases} \bot & \text{if } [\![e]\!]_a = \bot \\ [\![e_1]\!]_a & \text{if } [\![e]\!]_a = Nil_\tau \\ [\![e_2]\!]_{a[x \mapsto v_1, xs \mapsto v_2]} & \text{if } [\![e]\!]_a = Cons\ \langle v_1, v_2\rangle \end{cases} \\
[\![undefined_\tau]\!]_a &= \bot_\tau
\end{aligned}
$$

Figure 2.2.4: Term Semantics

The constants *False*, *True*, and *Nil* are interpreted by the corresponding mathematical objects. The semantics of a list whose outermost constructor is *Cons* is the semantical function $Cons\ \langle\cdot,\cdot\rangle$ applied to the semantics of the arguments of the list constructor. The term $\lambda a.\ b$, used in the semantics of a function f, introduces an anonymous mathematical function. Furthermore, in this rule we use a globally defined program called P. Recursive function definitions result in recursive definitions of the semantic function $[\![\cdot]\!]$. We use the least solution of the resulting system of equations.

The semantics of a case expression performs a case distinction over the semantics of the scrutinee. Note that, if the semantics of the scrutinee has the form $Cons\ \langle x, xs\rangle$, then we have to adapt the environment a accordingly.

Finally, the constant $undefined_\tau$ is interpreted by the least element of the interpretation of τ. This constant does not add expressiveness to the language as we could define $undefined_\tau$ as a non-terminating recursive function *loop* in the language itself.

To assign a type to a term we employ the typing rules presented in Figure 2.2.5. The typing rules are standard rules for a simply typed, monomorphic, functional language. A typing judgement has the form $\Gamma \vdash e :: \tau$ and states that an expression e has the type τ under a type assumption — also called type environment — Γ. A type assumption Γ has the form $\Gamma = \{x_1 :: \tau_1, \ldots, x_n :: \tau_n\}$ and assigns types to the free variables of an expression. By $\Gamma, x :: \tau$ we denote splitting a type environment into the binding of a variable x (here bound to a type τ) and an environment Γ that does not contain a binding for x.

$$\Gamma, x :: \tau \vdash x :: \tau \text{ (VAR)} \qquad \Gamma \vdash \mathit{False} :: \mathit{Bool} \text{ (FALSE)}$$

$$\Gamma \vdash \mathit{True} :: \mathit{Bool} \text{ (TRUE)} \qquad \Gamma \vdash \mathit{Nil}_\tau :: \mathit{List}\ \tau \text{ (NIL)}$$

$$\frac{\Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \mathit{List}\ \tau}{\Gamma \vdash \mathit{Cons}\langle e_1, e_2\rangle :: \mathit{List}\ \tau} \text{ (CONS)}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n :: \tau_n}{\Gamma \vdash \langle e_1, \ldots, e_n\rangle :: \tau_1 \times \cdots \times \tau_n} \text{ (TUPLE)}$$

$$\frac{\Gamma \vdash f :: \sigma \to \tau \qquad \Gamma \vdash e :: \sigma}{\Gamma \vdash f\ e :: \tau} \text{ (APP)} \qquad \frac{f :: \sigma \to \tau \in P}{\Gamma \vdash f :: \sigma \to \tau} \text{ (FUN)}$$

$$\frac{\Gamma \vdash e :: \mathit{Bool} \qquad \Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\mathit{False} \to e_1; \mathit{True} \to e_2\} :: \tau} \text{ (BCASE)}$$

$$\frac{\Gamma \vdash e :: \mathit{List}\ \tau \qquad \Gamma \vdash e_1 :: \tau' \qquad \Gamma, x :: \tau, xs :: \mathit{List}\ \tau \vdash e_2 :: \tau'}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\mathit{Nil}_\tau \to e_1; \mathit{Cons}\langle x, xs\rangle \to e_2\} :: \tau'} \text{ (LCASE)}$$

$$\Gamma \vdash \mathit{undefined}_\tau :: \tau$$

Figure 2.2.5: Typing Rules for the presented Functional Language

In the typing rule for a function we again use a globally defined program P. We call a program P *well-typed* if every type signature that is annotated to a function is correct with respect to the typing rules of Figure 2.2.5. More precisely, a program P is well-typed if we have $\emptyset \vdash f :: \sigma \to \tau$ for all functions $f :: \sigma \to \tau$ in the program P. In the following we assume that all programs P are well-typed.

Example 2.2.1: As an example for the semantics, presented in Figure 2.2.4, we consider the function *andL*, which is frequently used as an example in the following. The function *andL* corresponds to the standard Boolean conjunction (&&) in Haskell and is defined as follows.

$$\begin{array}{l}
\mathit{andL} :: \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool} \\
\mathit{andL}\langle x, y\rangle = \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad\quad \mathit{False} \to \mathit{False} \\
\qquad\quad \mathit{True}\ \to y
\end{array}$$

For any environment a we calculate the semantics of *andL* as follows.

$$\begin{aligned}
[\![\mathit{andL}]\!]_a &= \lambda\langle v_1, v_2\rangle.[\![\textbf{case}\ x\ \textbf{of}\ \{\mathit{False} \to \mathit{False}; \mathit{True} \to y\}]\!]_{a[x\mapsto v_1, y\mapsto v_2]} \\
&= \lambda\langle v_1, v_2\rangle. \begin{cases} \bot & \text{if } [\![x]\!]_{a[x\mapsto v_1, y\mapsto v_2]} = \bot \\ [\![\mathit{False}]\!]_{a[x\mapsto v_1, y\mapsto v_2]} & \text{if } [\![x]\!]_{a[x\mapsto v_1, y\mapsto v_2]} = \mathit{False} \\ [\![y]\!]_{a[x\mapsto v_1, y\mapsto v_2]} & \text{if } [\![x]\!]_{a[x\mapsto v_1, y\mapsto v_2]} = \mathit{True} \end{cases}
\end{aligned}$$

$$= \lambda\langle v_1, v_2\rangle. \begin{cases} \bot & \text{if } v_1 = \bot \\ [\![\mathit{False}]\!]_{a[x \mapsto v_1, y \mapsto v_2]} & \text{if } v_1 = \mathit{False} \\ [\![y]\!]_{a[x \mapsto v_1, y \mapsto v_2]} & \text{if } v_1 = \mathit{True} \end{cases}$$

$$= \lambda\langle v_1, v_2\rangle. \begin{cases} \bot & \text{if } v_1 = \bot \\ \mathit{False} & \text{if } v_1 = \mathit{False} \\ v_2 & \text{if } v_1 = \mathit{True} \end{cases}$$

As the semantics of *andL* is independent of the environment a, we can use $[\![\mathit{andL}]\!]$ to refer to $[\![\mathit{andL}]\!]_a$. □

3 Non-Strict Evaluation

While we have already introduced the most important features of Haskell in Section 2.1, in this chapter we talk about its operational model. Haskell is a *call-by need* language, that is, programs are evaluated by so-called *lazy evaluation* (Wadsworth 1971; Launchbury 1993).

There are two main kinds of evaluation strategies, *strict* and *non-strict* evaluation. In a language with strict evaluation the arguments of a function are evaluated before the function is applied. In contrast, in a language with non-strict evaluation the arguments of a function are only evaluated if they are needed for the evaluation of the function application. Furthermore, there are two main kinds of non-strict evaluation strategies, *call-by name* and *call-by need*. In contrast to call-by name, the evaluation with call-by need avoids unnecessary re-evaluations, by sharing expressions. From a denotational point of view, these evaluation mechanisms are not distinguishable as they only differ with respect to efficiency. As we are mainly interested in the denotational aspects of evaluation, in the following, we only explain the call-by name aspects of non-strict evaluation in detail. Nevertheless, when we consider the effects of less strict implementations with respect to memory usage, as it is the case in Section 3.2 as well as in Chapter 7, we informally consider the effects of sharing.

In a language with call-by name evaluation, the arguments of a function are only evaluated if they are needed to determine the value of the application. For example, consider the following function, called *const*.

$$\begin{aligned} &const :: \alpha \to \beta \to \alpha \\ &const\ x\ \_ = x \end{aligned}$$

It takes a value of some type τ as first argument, ignores its second argument, and yields its first argument as result. When we evaluate an expression of the form *const False expensive_computation*, the expression *expensive_computation* is not evaluated as we can determine the value of the application without evaluating the second argument of *const*. More precisely, when *const False expensive_computation* is evaluated, *const* is replace by its right-hand side where the corresponding variables are substituted accordingly. Thus, the expression *const False expensive_computation* is replaced by *False*.

We can also use a run-time error or a non-terminating expression to observe that *const* does not cause the evaluation of its second argument. An expression that causes a run-time error or does not terminate when it is evaluated is denoted by $\perp$ in the following. For example, we have *const False* $\perp \equiv$ *False* where $\equiv$ denotes semantic equivalence. That is, even if we apply *const False* to a run-time error, this error is not raised because of non-strict evaluation. In the same way, even if the second argument of *const* is an expression whose evaluation does not terminate, the evaluation of the application of *const* terminates.

3.1 Advantages of Non-Strict Evaluation

From a practical point of view there are two main aspects of non-strict evaluation. First, we can make use of infinite data structures. Consider the following function, called *iterate*.

$$\begin{aligned} &\mathit{iterate} :: (\alpha \to \alpha) \to \alpha \to [\alpha] \\ &\mathit{iterate}\ f\ x = x : \mathit{iterate}\ f\ (f\ x) \end{aligned}$$

This function takes an initial value and constructs a list by repeated application of the provided function. Intuitively, we have $\mathit{iterate}\ f\ x \equiv [x, f\ x, f\ (f\ x), \ldots]$, where the dots denote that the list continues infinitely. More precisely, the evaluation of $\mathit{iterate}\ f\ x$ yields a list whose n-th element is generated by applying f exactly $n-1$ times to the initial value x. For example, consider the expression $\mathit{iterate}\ (+1)\ 0$. When we evaluate this expression, we get an infinite list of increasing integers, starting with zero.

An infinite list seems to be of little interest, however in a non-strict language like Haskell we can make use of infinite data structures. For example, we can take an initial segment of an infinite list by means of the function $\mathit{take} :: [\alpha] \to \mathit{Int} \to [\alpha]$. This function takes a list and an integer and yields the initial segment of the list with the given number of elements. For example, when we evaluate the expression $\mathit{take}\ 5\ (\mathit{iterate}\ (+1)\ 0)$ we get the list $[0, 1, 2, 3, 4]$.

Infinite data structures allow for elegant definitions of certain algorithms. For example, we can define the Fibonacci function as follows. First, we define a constant *fibs* as the infinite list of Fibonacci numbers. The first two elements of the list of Fibonacci numbers are zero and one. We get the third element of the list by adding the first Fibonacci number to the second one, that is, we add the first element of *fibs* to the second element of *fibs*. We get the fourth element of the list by adding the second element to the third one. In general we simply use $\mathit{zipWith}\ (+)$ to point-wise add the elements of *fibs* to $\mathit{tail}\ \mathit{fibs}$ where *tail* yields its argument list without the head.

$$\begin{aligned} &\mathit{fibs} :: [\mathit{Integer}] \\ &\mathit{fibs} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs}) \end{aligned}$$

Note that we use the arbitrary precision integer type *Integer* for Fibonacci numbers as these numbers grow quite fast. By means of *fibs* we define a function *fib* that takes an integer and yields the corresponding Fibonacci number as follows.

$$\begin{aligned} &\mathit{fib} :: \mathit{Int} \to \mathit{Integer} \\ &\mathit{fib}\ i = \mathit{fibs}\ !!\ i \end{aligned}$$

By employing the list indexing function $(!!)$, we simply project to a specific position and look up the corresponding Fibonacci number in the infinite list.

In the example of the Fibonacci numbers we can generate Fibonacci numbers without considering which number we are interested in. Thus, by using infinite data structures we can separate the generation from the consumption of data. Another

example for this style of programming is the calculation of the n-th prime number (Turner 1976). We define a function that generates the infinite list of prime numbers and get the n-th number by projecting to the n-th position of the list.

The other important practical aspect of non-strict evaluation has first been observed by Hughes (1989), namely, that non-strictness contributes to the modularity of programs. Haskell programs are often defined by means of predefined functions like *map*, *filter*, and *zipWith*. Hughes (1989) has observed that this style of programming heavily relies on non-strict evaluation.

For example, consider the function $splitWhen :: (\alpha \to Bool) \to [\alpha] \to [[\alpha]]$ from the Hackage[1] package *split* by Yorgey (2010). The function *splitWhen* splits a list into a list of sublists. The list is split at all positions that satisfy the predicate, and, furthermore, the elements that satisfy the predicate are removed. For example, we have

$$splitWhen\ (==\ \texttt{'a'})\ \texttt{"abcada"} \equiv [\texttt{""},\texttt{"bc"},\texttt{"d"},\texttt{""}].$$

Moreover, the predicate $isAlpha :: Char \to Bool$ from the standard library *Data.Char* yields *True* if and only if its argument is an alphabetic character. As a string is defined as a list of characters, we can use the application $splitWhen\ (not \circ isAlpha)$ to split a string into substrings that only contain alphabetical characters. For example, we have

$$splitWhen\ (not \circ isAlpha)\ \texttt{"a.,bc d"} \equiv [\texttt{"a"},\texttt{""},\texttt{"bc"},\texttt{"d"}].$$

Thus, by employing this function we can extract the words that occur in a text where the text is represented by a single string.

By means of *splitWhen* we can define the following function, called *wordCount*. It takes a word and a text and counts the number of occurrences of the word in the text.

$$\begin{aligned} &wordCount :: String \to String \to Int \\ &wordCount = length \circ filter\ (==word) \circ splitWhen\ (not \circ isAlpha) \end{aligned}$$

First, we split the text into single words via $splitWhen\ (not \circ isAlpha)$, then we remove all words that are not equal to the provided word by using *filter*, and, finally, we count the length of the filtered list.

Consider that we want to count occurrences of the string `"mathematics"` in a file that contains Shakespeare's collected works, which is freely available from Project Gutenberg. In a strict programming language the function *splitWhen* evaluates its argument completely. That is, the string that represents Shakespeare's collected works is placed in the memory. In contrast, in a non-strict programming language like Haskell the production of data by one function and the consumption of this data by another function can be interleaved. As soon as the data is consumed, the memory can be deallocated by the garbage collector. This way, functions in a modular programming style, which is very commonly used in Haskell, often have a small memory footprint (Hughes 1989). For example, Figure 3.1.1 shows the heap profile for counting occurrences of `"mathematics"` in Shakespeare's collected works, which has a size of about six megabyte.[2]

[1] Hackage is a database for Haskell libraries and is available at `http://hackage.haskell.org`.

[2] The word `"mathematics"` occurs three times in Shakespeare's collected works.

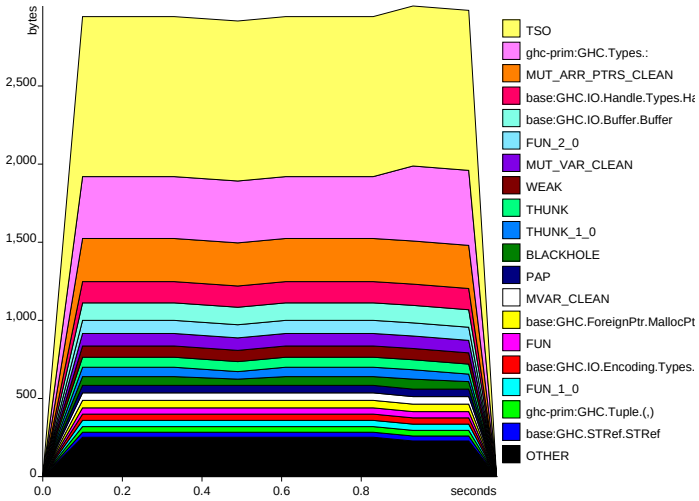


Figure 3.1.1: Heap Profile of Counting Occurrences of `"mathematics"` in Shakespeare's Collected Works.

Sansom and Peyton Jones (1995) have presented a tool to generate heap profiles for Haskell programs. The current implementation of the Glasgow Haskell Compiler (GHC) provides an implementation of this kind of profiler. A heap profile plots the occupied heap memory against the time. The GHC provides different kinds of profiling. In this thesis we use the most basic form of profiling that distinguishes memory occupied by different data constructors and different types of objects like functions (FUN) and thunks (THUNK). We use this basic form of profiling as we do not investigate the memory usage by means of profiling but only use the profiles for illustrative purposes.

The profile of counting occurrences of `"mathematics"` shows that it needs less than three kilobyte of memory to process the complete file. The different shades of gray — or colors respectively — in the heap profile distinguish heap objects of different kinds. For example, the topmost — or yellow — part of the heap represents the amount of memory that is occupied by the stack of the program. The topmost but one — or pink — part of the heap profile represents the amount of memory that is occupied by occurrences of the list constructor $(:)$ in the heap. The other parts of the profile are not of interest as these are mostly constant overheads caused by reading the content from a file and printing the result. In summary, we observe that the total amount of memory that is occupied at one time is quite small compared to the amount of data that is processed. And, even more impressively, the amount of memory that is occupied at one time stays the same even if we increase the size of the processed file.

3.2 Unnecessarily Strict Functions

Although Haskell is a non-strict programming language, functions can be unnecessarily strict. Consider the function *intersperse* from the standard library *Data.List*. It intersperses an element between all pairs of succeeding elements of a list.

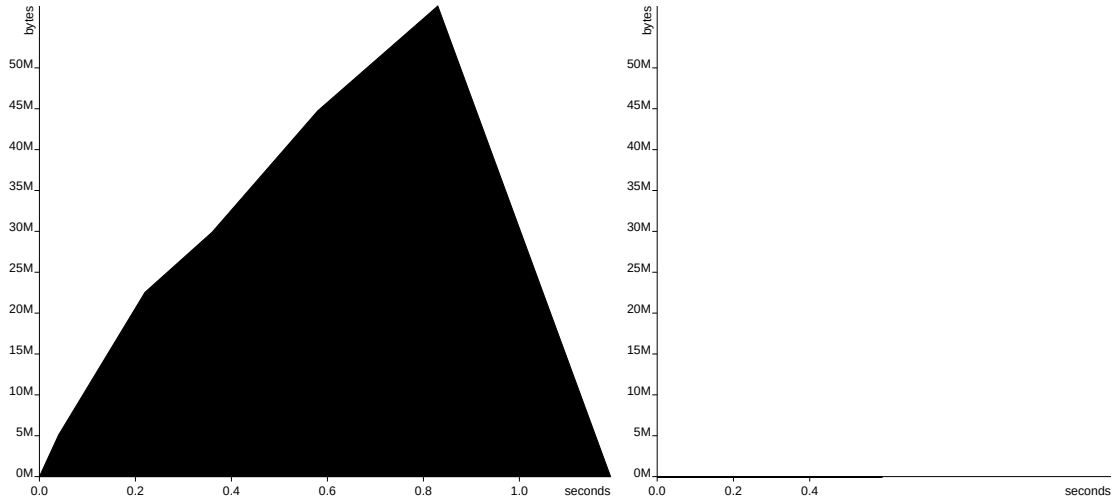


Figure 3.2.1: Heap Profile of Replacing Umlauts in Shakespeare's Work with the Standard (left) and the Less Strict (right) Implementation of *intersperse*.

$$
\begin{aligned}
&\mathit{intersperse} :: \alpha \to [\alpha] \to [\alpha] \\
&\mathit{intersperse}\ \_\ \ [\,] \qquad = [\,] \\
&\mathit{intersperse}\ \_\ \ [x] \qquad = [x] \\
&\mathit{intersperse}\ \mathit{sep}\ (x : \mathit{xs}) = x : \mathit{sep} : \mathit{intersperse}\ \mathit{sep}\ \mathit{xs}
\end{aligned}
$$

Furthermore, consider the function $\mathit{splitWhen} :: (\alpha \to \mathit{Bool}) \to [\alpha] \to [[\alpha]]$ from the Hackage package *split* again. By means of *intersperse* and *splitWhen* we define a function *replaceBy*, which replaces all occurrences of a specific element in a list by a given list.

$$
\begin{aligned}
&\mathit{replaceBy} :: \mathit{Eq}\ \alpha \Rightarrow \alpha \to [\alpha] \to [\alpha] \to [\alpha] \\
&\mathit{replaceBy}\ x\ \mathit{sep} = \mathit{concat} \circ \mathit{intersperse}\ \mathit{sep} \circ \mathit{splitWhen}\ (== x)
\end{aligned}
$$

First, by using *splitWhen* we split the argument list at all occurrences of the element x. Then, we intersperse the new separator *sep* between the elements of the resulting list, and, finally, we concatenate the resulting list via $\mathit{concat} :: [[\alpha]] \to [\alpha]$. For example, we have *replaceBy* `'a' "bb" "mathematics"` $\equiv$ `"mbbthembbtics"`.

Let us consider an application of the function *replaceBy*. In HTML we have to replace all occurrences of German umlauts by a specific HTML encoding. For this example we only replace all occurrences of the character `'\228'`, which is the German umlaut "ä", by the corresponding encoding `"&auml;"`. Again, we process the collected works by Shakespeare. Note that we consider the english version of Shakespeare's collected works, and, therefore, there are no umlauts. We will comment on this issue at the end of this section. The left-hand side of Figure 3.2.1 presents the heap profile for escaping umlauts in Shakespeare's collected works, again, this profile it generated by the GHC profiler.

The behavior of *replaceBy* is quite contrary to the expectations about memory usage in a non-strict programming language. The program uses more than 50 megabyte of memory while we would have expected a profile similar to Figure 3.1.1. In Figure 3.2.1 the black part of the heap profile represents the amount of memory that

is occupied by the list constructor (:). The profile does not contain other parts because parts that sum to a total of less than 1% of the profile are removed from a profile.

Unexpected consumption of memory like this is called a *space leak* (Wadler 1987). In this case the space leak is caused by an unnecessarily strict implementation of *intersperse*. We can use Sloth, the tool that is presented in this thesis, to observe that the current implementation of *intersperse* is too strict. As Sloth is a light-weight tool, we only have to add the following line to our module to check whether a function is as little strict as possible.

import *Test.Sloth*

This module provides a function called *strictCheck* to check whether a function is as little strict as possible for inputs up to a specific size. In the following we refer to a function that is as little strict as possible as minimally strict. In Chapter 4 we provide a formal definition of minimally strict functions. The size of a term is the number of constructors in the term. That is, $\mathit{strictCheck}\ (\mathit{intersperse} :: A \to [A] \to [A])\ 4$ checks whether the A-instance of *intersperse* is as little strict as possible for inputs whose size in sum is at most four. Sloth generates test cases, that is, elements of the argument type of a function, to check whether a function is minimally strict. In the case of a polymorphic function this is hardly possible as we would have to check every possible monomorphic instance of the polymorphic function. Instead, we instantiate all type variables of a polymorphic function type with the opaque data type A, which is provided by the module *Test.Sloth*. In Chapter 6 we prove that a polymorphic function is indeed minimally strict if and only if its A-instance is minimally strict.

Sloth reports the following result if we check whether *intersperse* is minimally

```
> strictCheck (intersperse :: A -> [A] -> [A]) 4
3: \a (b:⊥) -> b:⊥
5: \a (b:c:⊥) -> b:a:c:⊥
Finished 7 tests.
```

It presents two argument result pairs, which show that *intersperse* is unnecessarily strict. The first one states that *intersperse* yields $\bot$ if it is applied to a and $b : \bot$, where a and b are arbitrary values. A minimally strict implementation of *intersperse* yields $b : \bot$ for these arguments instead.[3] Sloth highlights the subterm on the right-hand side of -> where the tested function is too strict. That is, it highlights a term that is not an error, while the current implementation yields an error at the highlighted position instead. Moreover, there exists a less strict implementation that yields the highlighted value. Note that Sloth does not only observe that *intersperse* is too strict for the arguments a and $b : \bot$, but also that the first element of the result list has to be the first element of the argument list. This is possible due to the special treatment of polymorphic functions.

[3] For readibility we use $\bot$ in the output while Sloth actually uses the symbol _ instead.

In the following we refer to an argument together with the current result of the function and the proposed result as a counter-example. We use the term counter-example as these informations show that the function is not minimally strict. The second counter-example for *intersperse* states that *intersperse* yields $b : a : \bot$ if we apply it to a and $b : c : \bot$, while there exists a less strict implementation that yields $b : a : c : \bot$ for these arguments instead.

To provide a less strict implementation of *intersperse* we consider the following slightly different but equivalent implementation. We replace the pattern matching that checks whether the argument list has exactly one element by a **case** expression. Note that most compilers perform a similar transformation as many core languages use **case** expressions and rules that do not perform pattern matching. To transform *intersperse* into such a core language we would have to replace the pattern matching in the remaining rules by a **case** expression as well.

```
intersperse :: α → [α] → [α]
intersperse _   []       = []
intersperse sep (x : xs) =
  case xs of
       [] → x : []
       _  → x : sep : intersperse sep xs
```

Using the information provided by Sloth we can define a less strict implementation of *intersperse* by yielding the first element of the list "before" performing pattern matching on the tail.

```
intersperse' :: α → [α] → [α]
intersperse' _   []       = []
intersperse' sep (x : xs) =
  x : case xs of
           [] → []
           _  → sep : intersperse' sep xs
```

To get from *intersperse* to *intersperse'* we distribute the partial application $(x:)$ over the **case** expression. We refer to this transformation as **case** deferment. In general, **case** deferment does not always yield a less strict implementation, but we always get an implementation that is at least as little strict as the original implementation. In Chapter 7 we consider this transformation in more detail and develop conditions when **case** deferment yields a less strict implementation.

There we observe that *intersperse'* is indeed less strict than *intersperse* if there are arguments *sep* and $x : xs$ for which the evaluation of the scrutinee of the **case** expression yields an error. Furthermore, the context that is pulled over the case expression has to be more defined than an error if we apply the context to an error. More precisely, in this particular case, we have to provide arguments *sep* and $x : xs$ such that xs is equivalent to an error and $(x:)$ applied to an error is not an error. An arbitrary value *sep* of type τ and a list of the form $x : \bot$, where x is an arbitrary value of type τ and $\bot$ denotes an error, satisfy these requirements. The results from Chapter 7 show that *intersperse'* is, therefore, indeed less strict than *intersperse*.

The definition of *intersperse′* still has a shortcoming, and, in fact, the definition of *intersperse* has the same shortcoming. The function *intersperse′* performs the same pattern matching several times because *intersperse′* checks whether *xs* is the empty list and we perform the same check again in the recursive application. To calculate an improved implementation of *intersperse* we apply the worker/wrapper transformation by Gill and Hutton (2009). We do not present the calculation itself, but, for completeness, we provide the functions *wrap* and *unwrap*, used to derive the new implementation. We use the following functions.

```
wrap :: (α → [α] → [α]) → α → [α] → [α]
wrap _ _   []       = []
wrap f sep (x : xs) = x : f sep xs

unwrap :: (α → [α] → [α]) → α → [α] → [α]
unwrap _ _   []  = []
unwrap f sep xs = sep : f sep xs
```

Using these functions for the worker/wrapper transformation we can derive the following function *intersperse″*, which is equivalent to *intersperse′* and, therefore, less strict than *intersperse*.

```
intersperse″ :: α → [α] → [α]
intersperse″ _   []       = []
intersperse″ sep (x : xs) = x : work xs
  where
    work []       = []
    work (y : ys) = sep : y : work ys
```

Because we only know that *intersperse″* is less strict than *intersperse* but not whether *intersperse″* is as little strict as possible, we can use Sloth to check whether *intersperse″* is still unnecessarily strict. Sloth does not report any counter-examples for inputs up to size 100.

The heap profile on the right-hand side of Figure 3.2.1 shows the memory usage of escaping umlauts using the less strict implementation of *intersperse*. As both profiles in Figure 3.2.1 use the same scaling, the heap profile for the less strict implementation is barely notable. There is a small black line at the bottom of the graph which states that the process takes around 0.6 seconds. Figure 3.2.2 presents a magnified version of the heap profile of escaping umlauts using the less strict implementation *intersperse″*. In this case the program uses less than three kilobyte of memory. That is, one delay of a pattern matching improves the memory usage by a factor of 20,000.

Obviously, we seldomly generate HTML from a 6MB text file. Furthermore, we have chosen a file that does not contain any umlauts on purpose. The presented space leak is linear in the size of the longest substring that does not contain the character that is replaced. More precisely, *intersperse* does not yield a result until it is able to observe whether its argument has exactly one element. The function *splitWhen* checks whether any character of Shakespeare's collected works is an umlaut and when it finally arrives at the end the corresponding string yields a singleton list that contains Shakespeare's collected works. Therefore, *splitWhen* evaluates the string

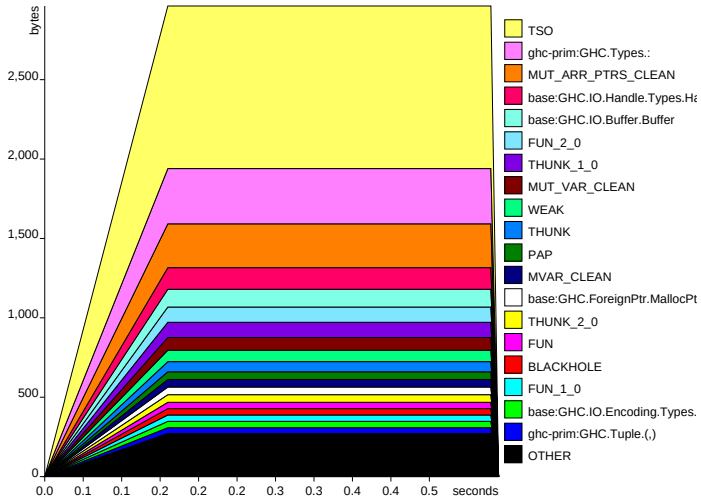


Figure 3.2.2: Magnified Version of the Heap Profile of Replacing Umlauts in Shakespeare's Work with the Less Strict Implementation of *intersperse*.

that represents Shakespeare's collected works completely before it yields a result with one element. Furthermore, *intersperse* keeps the result of *splitWhen* in the memory as it does not yield the first element of its result until it has checked whether its argument has exactly one element. Therefore, we get a space usage that is linear in the size of Shakespeare's collected works.

In contrast to *intersperse*, as soon as *splitWhen* has observed that a character of Shakespeare's collected works is no umlaut, the function *intersperse''* passes this character on. Thus, *replaceBy* passes Shakespeare's collected works character-wise to its surrounding context if we use *intersperse''*. Therefore, if we consider a well-behaved context, the garbage collector is able to deallocate parts of the string of Shakespeare's collected works immediately. Hence, this process never occupies a large amount of memory.

Although we have considered an input that does not contain umlauts, this is not an artificial example. The presented space leak is a modification of a space leak discovered by Fischer (2010) in his *searchstring* package (Fischer et al. 2010). Furthermore, a priori we often do not know whether a file contains a certain character or substring, and we also get a space leak if the character or substring occurs only rarely in the file. In summary, this example is supposed to emphasise that being too strict can have a tremendous effect with respect to memory behavior when processing large amounts of data.

4 Mathematical Model of Minimally Strict Functions

In this chapter we present the theoretical background for minimal strictness and the implementation of Sloth. In Section 4.1 we use the denotational semantics presented in Section 2.2 as the basis for the definition of least-strictness, which has been introduced by Chitil (2006) in an informal manner. We introduce the concept of sequentiality in Section 4.2, which is used in Section 4.3 to define minimal strictness. Furthermore, we argue that the conecpt of minimally strict functions is an improvement over the concept of least strict functions. Finally, we provide a criterion to check whether a function is minimally strict and prove that this criterion is sufficient (Section 4.3.1) and necessary (Section 4.3.2). As mentioned in the introduction all proofs omited in this chapter are presented in Appendix A.

4.1 Least Strict Functions

Chitil (2006) has originally presented the idea of checking whether a function is as little strict as possible. In this section we present his approach and argue that it has a shortcoming that is undesirable in practice.

We start by defining a less-strict relation for functions. On basis of the semantics of a type we define the set of total values of a type. A total value is a maximal element of the interpretation of a type.

Definition 4.1.1 (Total Value): For a partially ordered set $(P, \sqsubseteq)$ we define the set of all maximal elements $P{\uparrow} \subseteq P$ as follows.

$$P{\uparrow} := \{x \in P \mid \forall y \in P.\, x \sqsubseteq y \Longrightarrow x = y\} \qquad \square$$

In the following we refer to elements of $[\![\tau]\!]$ as values of type τ, to elements of $[\![\tau]\!]{\uparrow}$ as total values of type τ, and to elements of $[\![\tau]\!] \setminus [\![\tau]\!]{\uparrow}$ as partial values of type τ. Furthermore, the name of a variable indicates its kind, more precisely, we use variable names that start with v for values, pv for partial values, and tv for total values.

The next definition provides a formal treatment of the less-strict relation established by Chitil (2006). In other words, we define when we call a function f less or equally strict than a function g.

Definition 4.1.2 (Less-Strict Relation): Let $f, g \in [\![\sigma \to \tau]\!]$. The functions f and g are related by $\preceq$ if the following holds.

$$f \preceq g \;\; :\Longleftrightarrow \;\; \forall\, tv \in [\![\sigma]\!]{\uparrow}.\, f\, tv = g\, tv \;\; \wedge \;\; \forall\, v \in [\![\tau]\!].\, f\, v \sqsupseteq g\, v$$

If $f \prec g$, that is, $f \preceq g$ and $g \not\preceq f$, we say that f is less strict than g. □

The relation $\preceq$ is reflexive, antisymmetric, and transitive because $=$ and $\sqsupseteq$ are reflexive, antisymmetric, and transitive; thus, $\preceq$ is a partial order.

Note that two functions are only related by $\preceq$ if they agree for total values. That is, we assume that it is intended if a function yields a non-total result for a total argument as we illustrate in the following example.

Example 4.1.1: Consider the function *head* that takes a list and yields its first element. As our simple functional language does not provide polymorphism, we consider the monomorphic Boolean instance of the *head* function. Naturally the *head* function yields a run-time error if it is applied to an empty list.

$$
\begin{array}{l}
\mathit{head} :: \mathit{List\ Bool} \to \mathit{Bool} \\
\mathit{head}\ \mathit{xs} = \\
\quad \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nil}_{\mathit{Bool}} & \to \mathit{undefined}_{\mathit{Bool}} \\
\mathit{Cons}\langle y, \mathit{ys}\rangle & \to y
\end{array}
\end{array}
$$

We get the following semantics for *head*.

$$[\![\mathit{head}]\!] = \lambda v. \begin{cases} v_1 & \text{if } v = \mathit{Cons}\ \langle v_1, v_2\rangle \\ \bot & \text{otherwise} \end{cases}$$

Now consider the function *head*′ that yields a Boolean value, namely, *False*, in the case that its argument is the empty list.

$$
\begin{array}{l}
\mathit{head}' :: \mathit{List\ Bool} \to \mathit{Bool} \\
\mathit{head}'\ \mathit{xs} = \\
\quad \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nil}_{\mathit{Bool}} & \to \mathit{False} \\
\mathit{Cons}\langle y, \mathit{ys}\rangle & \to y
\end{array}
\end{array}
$$

We get the following semantics for *head*′.

$$[\![\mathit{head}']\!] = \lambda v. \begin{cases} \mathit{False} & \text{if } v = \mathit{Nil}_\tau \\ v_1 & \text{if } v = \mathit{Cons}\ \langle v_1, v_2\rangle \\ \bot & \text{otherwise} \end{cases}$$

If we omit the requirement that two functions f and g are only related by $\preceq$ if they agree for total values, then *head*′ is less strict than *head*. Furthermore, if we replace *False* in the definition of *head*′ by *True* we get another function that is less strict than *head*. Regarding a tool that checks whether a function is unnecessarily strict the tool would have to suggest an arbitrary total value as result for the empty list. However, more importantly, in the case of *head* the programmer intended to define a partial function. Therefore, we do not want to suggest a total function only for the sake of a less strict implementation. □

Here and in the following we regard the behavior of a function for total values as the one a programmer cares about. That is, we might change the behavior of a function with respect to partial values but preserve its behavior with respect to total values. In summary, we want to check whether a function is as little strict as possible with respect to partial values while we preserve its behavior with respect to total values. The following example illustrates the less-strict relation by means of several implementations of the Boolean conjunction.

Example 4.1.2: Let us consider the Boolean conjunction *andL* from Example 2.2.1 and its strict counterpart *and* that is defined as follows.

$$
\begin{array}{l}
\mathit{and} :: \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool} \\
\mathit{and}\langle x, y\rangle = \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad \mathit{False} \to \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\quad \mathit{False} \to \mathit{False} \\
\qquad\qquad\qquad\quad \mathit{True}\ \to \mathit{False} \\
\qquad \mathit{True}\ \to \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\quad \mathit{False} \to \mathit{False} \\
\qquad\qquad\qquad\quad \mathit{True}\ \to \mathit{True}
\end{array}
$$

We get the following semantics for *and*.

$$
[\![\mathit{and}]\!] = \lambda\langle v_1, v_2\rangle. \begin{cases} \bot & \text{if } v_1 = \bot \vee v_2 = \bot \\ \mathit{True} & \text{if } v_1 = \mathit{True} \wedge v_2 = \mathit{True} \\ \mathit{False} & \text{otherwise} \end{cases}
$$

For all total values of type $\mathit{Bool} \times \mathit{Bool}$, $\langle \mathit{False}, \mathit{False}\rangle$, $\langle \mathit{False}, \mathit{True}\rangle$, $\langle \mathit{True}, \mathit{False}\rangle$, and $\langle \mathit{True}, \mathit{True}\rangle$, the function *andL* yields the same results as *and*. For the partial value $\langle \mathit{False}, \bot\rangle$, the function *andL* yields *False* while *and* yields $\bot$. For all other partial values *andL* yields the same results as *and*. Therefore, by Definition 4.1.2 *andL* is less strict than *and*, that is, $\mathit{andL} \prec \mathit{and}$. For readability, here and in the following, we sometimes identify a function f with its semantics $[\![f]\!]$.

Let us consider the symmetric counterpart of *andL*, called *andR*. While *andL* performs pattern matching on its first argument, *andR* performs pattern matching on its second argument.

$$
\begin{array}{l}
\mathit{andR} :: \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool} \\
\mathit{andR}\langle x, y\rangle = \\
\quad \textbf{case}\ y\ \textbf{of} \\
\qquad \mathit{False} \to \mathit{False} \\
\qquad \mathit{True}\ \to x
\end{array}
$$

In comparison to the semantics of *andL*, in the semantics of *andR* the arguments v_1 and v_2 have changed their roles.

$$[\![andR]\!] = \lambda\langle v_1, v_2\rangle. \begin{cases} \bot & \text{if } v_2 = \bot \\ \mathit{False} & \text{if } v_2 = \mathit{False} \\ v_1 & \text{if } v_2 = \mathit{True} \end{cases}$$

In the same manner as before, we can observe that *andR* is less strict than *and*. However, *andL* and *andR* are incomparable. While *andL* yields a more defined result than *andR* for the argument $\langle \mathit{False}, \bot\rangle$, the function *andR* yields a more defined result than *andL* for the argument $\langle \bot, \mathit{False}\rangle$. Altogether we get $\mathit{andL} \prec \mathit{and}$ and $\mathit{andR} \prec \mathit{and}$, but $\mathit{andR} \not\preceq \mathit{andL}$ and $\mathit{andL} \not\preceq \mathit{andR}$. □

So, how can we check whether a function is least strict or not? Let $f \in [\![\sigma \to \tau]\!]$. Because f is monotonic, for every value v of type σ and every total value tv of type σ with $v \sqsubseteq tv$, we have $f\ v \sqsubseteq f\ tv$. Therefore, $f\ v$ is a lower bound of the set

$$\{f\ tv \mid tv \in [\![\sigma]\!]\uparrow, v \sqsubseteq tv\}.$$

As $f\ v$ is a lower bound of this set, it is less or equally defined than the corresponding greatest lower bound.

$$f\ v \sqsubseteq \bigsqcap \{f\ tv \mid tv \in [\![\sigma]\!]\uparrow, v \sqsubseteq tv\}$$

Chitil (2006) uses this inequality[1] to check whether a function is least strict. A least strict function is supposed to agree with this greatest lower bound for all possible inputs v. The following definition states when a function is called *least strict*.

Definition 4.1.3 (Least-Strictness): For a function $f \in [\![\sigma \to \tau]\!]$ we define a function $\inf_f : [\![\sigma]\!] \to [\![\tau]\!]$ as follows.

$$\inf_f\ v = \bigsqcap \{f\ tv \mid tv \in [\![\sigma]\!]\uparrow, v \sqsubseteq tv\}$$

The function f is called *least strict* if and only if $f = \inf_f$. □

We have to prove that the infimum in the previous definition exists. First of all, all domains used here are interpreted as bounded complete cpos. A bounded complete cpo is also a complete meet-semilattice. In a complete meet-semilattice the infima of all non-empty subsets exist. That is, we have to show that the set

$$\{f\ tv \mid tv \in [\![\sigma]\!]\uparrow, v \sqsubseteq tv\}$$

is non-empty for all $v \in [\![\sigma]\!]$. We consider the following non-empty set of upper bounds of v.

$$U := \{v' \in [\![\sigma]\!] \mid v \sqsubseteq v'\}$$

As $[\![\sigma]\!]$ is a cpo, every chain in U has a least upper bound. By monotonicity this least upper bound is itself an element of U. Furthermore, $(U, \sqsubseteq)$ is a partially ordered set

[1] Accidentally Chitil (2006) uses $\bigsqcup$ instead of $\bigsqcap$.

and, by the Lemma of Zorn, U has at least one maximal element. Let m be a maximal element of U. We show that we have $m \in [\![\sigma]\!]\uparrow$ by employing the definition of $[\![\sigma]\!]\uparrow$. Let $y \in [\![\sigma]\!]$ such that $m \sqsubseteq y$. By transitivity we get $v \sqsubseteq y$ and, therefore, $y \in U$. As m is maximal in U, we get $m = y$ and, thus, $m \in [\![\sigma]\!]\uparrow$. This shows that the set $\{f\ tv \mid tv \in [\![\sigma]\!]\uparrow, v \sqsubseteq tv\}$ is non-empty and, thus, has a greatest lower bound. In the following we frequently employ the fact that for every value v of type σ there exists a total value tv of type σ with $v \sqsubseteq tv$.

Note that f and $\inf_f$ agree for total values, that is, we have $f\ tv = \inf_f\ tv$ for all total values tv. Furthermore, if a function f is not least strict there exists a partial value pv such that $f\ pv \sqsubset \inf_f\ pv$. Thus, if f is not least strict the function $\inf_f$ is less strict than f. Hence, we are not only able to check whether a function is least strict but can also provide a less strict function, namely, $\inf_f$.

The definition of least-strictness has a shortcoming. Some functions that one would rank to be as little strict as possible are not least strict.

Example 4.1.3: Let us consider the function *andL* from Example 2.2.1 again. For the argument $\langle \bot, False \rangle$, the function *andL* yields $\bot$. On the other hand, we obtain the following equality.

$$\begin{aligned}
\inf_{[\![andL]\!]} \langle \bot, False \rangle &= \bigsqcap \{ [\![andL]\!]\ tv \mid tv \in [\![Bool \times Bool]\!]\uparrow, \langle \bot, False \rangle \sqsubseteq tv \} \\
&= \bigsqcap \{ [\![andL]\!]\ \langle False, False \rangle, [\![andL]\!]\ \langle True, False \rangle \} \\
&= \bigsqcap \{ False, False \} \\
&= False
\end{aligned}$$

That is, *andL* is not least strict because $[\![andL]\!]\ \langle \bot, False \rangle \neq \inf_{[\![andL]\!]}\ \langle \bot, False \rangle$. □

Besides the definition of least-strictness, Chitil (2006) has presented a tool, which is called StrictCheck, that checks whether a Haskell function f agrees with $\inf_f$ for all values up to a specific size. For example, the following application checks whether the Boolean conjunction (&&) is least strict for values up to size ten. The function *test2*, provided by StrictCheck, takes an integer that specifies the maximal size of the generated test cases as first argument and a two-ary function as second argument.

```
Main> test2 10 (&&)
Function seems not to be least strict.
Input(s): (⊥, False)
Current output: ⊥
Proposed output: False
```

The result presented by *test2* states that (&&) applied to an error (denoted by $\bot$) and *False* yields an error, but a least strict implementation is supposed to yield *False* instead. Note that (&&) is the Haskell function that corresponds to *andL*, which is defined in Example 2.2.1. That is, StrictCheck yields the result that we have observed in Example 4.1.3. Besides checking whether we have $[\![andL]\!] = \inf_{[\![andL]\!]}$ for all inputs, StrictCheck suggests

$$\inf_{[\![andL]\!]} \langle \bot, False \rangle = False$$

as result of $[\![andL]\!]$ for the argument $\langle \bot, False \rangle$ because we have

$$[\![andL]\!]\ \langle \bot, False \rangle \neq \inf_{[\![andL]\!]}\ \langle \bot, False \rangle.$$

Actually, *andL* is indeed not as little strict as possible. We can define a Boolean conjunction in Haskell that yields *False* for the argument $\langle False, \bot \rangle$ as well as for the argument $\langle \bot, False \rangle$. For example, we can use the *unamb* operator, presented by Elliott (2009). The function $unamb :: \alpha \to \alpha \to \alpha$ takes two arguments and yields the argument that is not equal to $\bot$. If both arguments are $\bot$, it yields $\bot$ as well. To obtain this behavior even if $\bot$ is a non-terminating computation, *unamb* uses concurrency to fork two threads that evaluate the arguments of *unamb* in parallel. If one of the two threads is successful, *unamb* yields the result of this thread.

Because Haskell is a pure functional language, concurrency, as it is provided by Concurrent Haskell (Peyton Jones et al. 1996), for example, is performed in the *IO* monad (Peyton Jones and Wadler 1993). Thus, to define a function of type $\alpha \to \alpha \to \alpha$ that uses concurrency we have to employ the function $unsafePerformIO :: IO\ \alpha \to \alpha$, which takes an I/O action and yields a non-monadic value. As its name suggests, *unsafePerformIO* is unsafe in the sense that it might destroy referential transparency. For example, the expression *unamb False True* might be evaluated to *False* or *True*, depending on the scheduling of the forked threads. Therefore, *unamb* breaks referential transparency as we cannot replace equals for equals. For example, the evaluation of **let** $x = unamb\ False\ True$ **in** (x, x) might yield another result than the evaluation of $(unamb\ False\ True, unamb\ False\ True)$ as the two occurrences of *unamb False True* might yield different results. Nevertheless, the function *unamb* is referential transparent if its arguments have a common least upper bound.

In practice we often want to know whether there is a less strict implementation that avoids the use of features like concurrency and *unsafePerformIO*. In fact, many functions are not least strict because there is often a bias towards one argument with respect to strictness. By using StrictCheck we have observed, that the functions $(\&\&)$, $(||)$, $(+\!\!+)$, *and*, *or*, *zip*, as well as the list instances of $(==)$ and *compare* are not least strict, for example.

As an example, we examine the monomorphic instance of the equality check $(==)$ for lists of Boolean values. StrictCheck yields the following counter-example if we check $(==) :: [Bool] \to [Bool] \to Bool$ for Boolean lists up to size 10.

```
Main> test2 10 ((==) :: [Bool] -> [Bool] -> Bool)
Function seems not to be least strict.
Input(s): ([False], [⊥, False])
Current output: ⊥
Proposed output: False
```

That is, we are supposed to define an equality check that yields *False* for the arguments $False : [\,]$ and $\bot : False : [\,]$. Furthermore, as the current implementation of $(==)$ yields *False* for the arguments $False : \bot$ and $True : \bot$, a less strict implementation of the equality check is supposed to yield *False* as well. However, we cannot define a less strict implementation that satisfies these requirements and does not use features like concurrency, and it is not necessarily easy to observe this fact.

In contrast to StrictCheck, Sloth, the tool presented in this thesis, identifies $(\&\&)$, $(||)$, $(+\!\!+)$, *and*, *or*, *zip*, as well as the list instances of $(==)$ and *compare* as minimally strict as it checks whether a function is too strict without allowing features like concurrency. To restrict the considered functions we employ the concept of sequentiality by Vuillemin (1974). In the following section we introduce sequentiality and the concept of minimally strict functions, which is based on sequentiality.

4.2 Sequential and Demanded Positions

In order to restrict the considered first-order functional language to functions that do not use features like concurrency, we employ the definition of sequentiality by Vuillemin (1974). The definition of sequentiality employs contexts, which we only informally introduce here. Let Σ be the set of constructor and function symbols. A context is a term over Σ and the additional symbol $[\cdot]$, called hole. Let C be a context with n holes, that is, n occurrences of the symbol $[\cdot]$, and $e_1, \ldots, e_n$ be expressions. Then, $C[e_1, \ldots, e_n]$ denotes the context C where the i-th hole is filled in with expression e_i. Vuillemin (1974) has given the following definition of a sequential language.

Definition 4.2.1 (Sequentiality): A functional language is called sequential if it satisfies the following. Let C be a context with n holes and

$$[\![C[\mathit{undefined}_{\tau_1}, \ldots, \mathit{undefined}_{\tau_n}]]\!] = \bot.$$

Then, there exists at least one $i \in \{1, \ldots, n\}$ such that

$$[\![C[e_1, \ldots, e_{i-1}, \mathit{undefined}_{\tau_i}, e_{i+1}, \ldots, e_n]]\!] = \bot$$

for all expressions $e_1, \ldots, e_n$ of appropriate types. □

The following example illustrates the definition of sequentiality.

Example 4.2.1: By using the definition of sequentiality we can show that a Boolean conjunction that is less strict than *andL* is not sequential. We consider a Boolean conjunction *andL′* and a context C defined as $\mathit{andL}'\langle[\cdot], [\cdot]\rangle$. We have

$$\begin{aligned}
[\![C[\mathit{undefined}_{\mathit{Bool}}, \mathit{undefined}_{\mathit{Bool}}]]\!] &= [\![\mathit{andL}'\langle \mathit{undefined}_{\mathit{Bool}}, \mathit{undefined}_{\mathit{Bool}}\rangle]\!] \\
&= [\![\mathit{andL}']\!]\, [\![\langle \mathit{undefined}_{\mathit{Bool}}, \mathit{undefined}_{\mathit{Bool}}\rangle]\!] \\
&= [\![\mathit{andL}']\!]\, \langle \bot, \bot\rangle \\
&= \bot,
\end{aligned}$$

that is, we can apply Definition 4.2.1 to the context C. The last step of this equation follows from monotonicity of $[\![\mathit{andL}']\!]$ and the specification of a Boolean conjunction, more precisely, $[\![\mathit{andL}']\!]\, \langle \mathit{True}, \mathit{False}\rangle = \mathit{False}$ and $[\![\mathit{andL}']\!]\, \langle \mathit{True}, \mathit{True}\rangle = \mathit{True}$.

In a sequential language, *andL* has to be strict with respect to one of its arguments. More precisely, we have

$$[\![C[\mathit{undefined}_{\mathit{Bool}}, b]]\!] = [\![\mathit{andL}'\langle \mathit{undefined}_{\mathit{Bool}}, b\rangle]\!] = [\![\mathit{andL}']\!]\, \langle \bot, [\![b]\!]\rangle = \bot$$

for all Boolean expressions b *or* we have

$$[\![C[b, \mathit{undefined}_{\mathit{Bool}}]]\!] = [\![\mathit{andL}'\langle b, \mathit{undefined}_{\mathit{Bool}}\rangle]\!] = [\![\mathit{andL}']\!]\,\langle [\![b]\!], \bot\rangle = \bot$$

for all Boolean expressions b. Because andL' is less strict than andL, we have

$$[\![\mathit{andL}']\!]\,\langle \mathit{False}, \bot\rangle \sqsupseteq [\![\mathit{andL}]\!]\,\langle \mathit{False}, \bot\rangle = \mathit{False}.$$

This property together with sequentiality implies

$$[\![\mathit{andL}']\!]\,\langle \bot, \mathit{False}\rangle = \bot.$$

As we have observed in Example 4.1.3, a least strict Boolean conjunction is supposed to yield *False* for the argument $\langle \bot, \mathit{False}\rangle$. However, this application of sequentiality shows that there is no least strict Boolean conjunction in a sequential language. □

To adapt the definition of sequentiality to our needs we introduce the concepts of positions, projections, and substitutions. The projections presented here are a generalization of the continuous projections that come along with the direct product. We start with the definition of the set of positions.

Definition 4.2.2 (Position): A position is a sequence of natural numbers. We denote the set of positions by $\mathbb{N}^*$, that is, the free monoid over the natural numbers. The empty sequence is denoted by ϵ. If n is a natural number and p is a position, then $n.p$ is a position. Instead of $n.\epsilon$ we synonymously use n. For positions p and p' we define that $p \leq p'$ if and only if p is a prefix of p'. □

Before we define projections and substitutions, we define the set of valid positions of an element of $[\![\tau]\!]$ for some type τ. For simplicity, we define a single function that takes an element of the interpretation of an arbitrary type and yields the set of valid positions of this element. That is, we define a function $\mathit{Pos} : \bigcup_{\tau \in \mathit{Types}} [\![\tau]\!] \to 2^{\mathbb{N}^*}$ where $\mathbb{N}^*$ denotes the set of positions. As we want to define this function recursively, we define it by means of a least fixpoint. Therefore, it is advantageous if domain and range of Pos are cpos. As mentioned before, the powerset of an arbitrary set constitutes a cpo. As the domain of Pos is not a cpo, we add a least element denoted by $\bot$ to the set $\bigcup_{\tau \in \mathit{Types}} [\![\tau]\!]$. More precisely, we have $\bot \sqsubset \bot_\tau$ for all $\tau \in \mathit{Types}$. Furthermore, two elements from the same cpo $[\![\tau]\!]$ are related by $\sqsubseteq$ if they are related by $\sqsubseteq$ in $[\![\tau]\!]$, and two elements from different cpos are incomparable. This construction is sometimes called separated sum as we sum up several cpos but separate the sets in the sense that we distinguish least elements of different cpos. Figure 4.2.1 illustrates the structure of the resulting cpo. In the following we abbreviate the set $\{\bot\} \cup \bigcup_{\tau \in \mathit{Types}} [\![\tau]\!]$ to $\Sigma_{\tau \in \mathit{Types}} [\![\tau]\!]$.

Definition 4.2.3 (Valid Position): We define a function $\mathit{Pos} : \Sigma_{\tau \in \mathit{Types}} [\![\tau]\!] \to 2^{\mathbb{N}^*}$ that yields the set of valid positions for an element of the interpretation of some type τ. By *Types* we denote the set of first-order types, that is, the set *Types* contains tuples of arbitrary arity, Booleans, and list types. As the domain of Pos contains an additional least element $\bot$, we define $\mathit{Pos}\ \bot = \emptyset$.

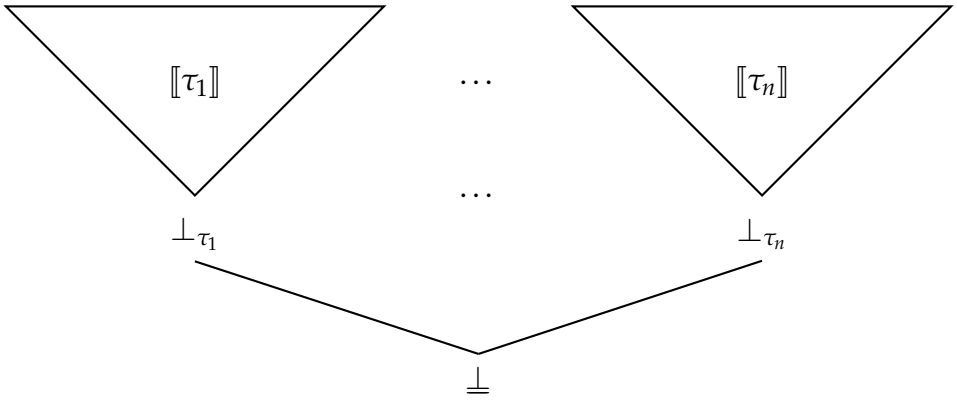


Figure 4.2.1: Separated Sum of some Cpos $[\![\tau_1]\!]$ to $[\![\tau_n]\!]$

For the elements of the interpretation of a tuple type $\tau_1 \times \cdots \times \tau_n$ we define the set of valid positions as follows.

$$Pos\ \langle v_1, \ldots, v_n \rangle = \bigcup_{i \in \{1,\ldots,n\}} \{i.p \mid p \in Pos\ v_i\}$$

In other words, we precede each valid position of one of the components of the tuple with the position of the component in the tuple.

For the elements of the interpretation of the type *Bool* we define the set of valid positions as follows.

$$Pos\ \bot_{Bool} = \{\epsilon\} \qquad Pos\ False = \{\epsilon\} \qquad Pos\ True = \{\epsilon\}$$

That is, the only valid position of any Boolean value is the root position ϵ.

Next, we define Pos for the elements of the interpretation of *List* τ. Let $v_1 \in [\![\tau]\!]$ and $v_2 \in [\![List\ \tau]\!]$. We define the set of valid positions of a list as follows.

$$Pos\ \bot_{List\ \tau} = \{\epsilon\} \qquad Pos\ Nil_\tau = \{\epsilon\}$$

$$Pos\ (Cons\ \langle v_1, v_2 \rangle) = \{\epsilon\} \cup \{1.p \mid p \in Pos\ v_1\} \cup \{2.p \mid p \in Pos\ v_2\}$$

To solve this recursive definition we use the least solution of this equation. As the set $\Sigma_{\tau \in Types}[\![\tau]\!]$ as well as $2^{\mathbb{N}^*}$ constitute cpos and the corresponding functional is monotonic, the least fixpoint of this functional exists. □

We give some examples of valid positions in the following.

Example 4.2.2: We consider the Boolean list $Cons\ \langle \bot_{Bool}, Nil_{List\ Bool} \rangle$. We have

$$Pos\ (Cons\ \langle \bot_{Bool}, Nil_{List\ Bool} \rangle) = \{\epsilon, 1, 2\},$$

that is, the root position as well as the first and the second argument of *Cons* are valid positions.

If we consider the tuple $\langle \mathit{False}, \mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \mathit{Nil}_{\mathit{List\ Bool}} \rangle \rangle$, which contains *False* in the first component and the Boolean list considered above in the second component, we have

$$\mathit{Pos}\ (\langle \mathit{False}, \mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \mathit{Nil}_{\mathit{List\ Bool}} \rangle \rangle) = \{1, 2, 2.1, 2.2\}.$$

That is, the first and the second component of the tuple are valid positions and the two arguments of *Cons* are valid positions. □

Next, we define a projection function of type $\Sigma_{\tau \in \mathit{Types}} [\![\tau]\!] \times \mathbb{N}^* \to \Sigma_{\tau \in \mathit{Types}} [\![\tau]\!]$ that takes a term e and a position p and yields the sub-term at that particular position denoted by $e|_p$. Intuitively, we want to project to valid positions only. Thus, we could simply define a partial function that is not defined if the position is not a valid position of the argument of the projection. However, consider a value like the list $\mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \bot_{\mathit{List\ Bool}} \rangle$. If we project to position 1 of this list, we want to get the first element of the list, that is, we would like to have

$$(\mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \bot_{\mathit{List\ Bool}} \rangle)|_1 = \bot_{\mathit{Bool}}.$$

For proving statements involving projections it is advantageous if the projection function is continuous. Now let us consider the supremum

$$\bot_{\mathit{List\ Bool}} \sqcup \mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \bot_{\mathit{List\ Bool}} \rangle.$$

By continuity we would get

$$\begin{aligned}(\mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \bot_{\mathit{List\ Bool}} \rangle)|_1 &= (\bot_{\mathit{List\ Bool}} \sqcup \mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \bot_{\mathit{List\ Bool}} \rangle)|_1 \\ &= \bot_{\mathit{List\ Bool}}|_1 \sqcup (\mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \bot_{\mathit{List\ Bool}} \rangle)|_1.\end{aligned}$$

If $\cdot|_\cdot$ is a partial function, then $\bot_{\mathit{List\ Bool}}|_1$ is not defined as we have $\mathit{Pos}\ \bot_{\mathit{List\ Bool}} = \{\epsilon\}$. Therefore, instead of defining a partial projection function we define a function that yields the least element $\underline{\bot}$ if we project to a non-valid position.

Definition 4.2.4 (Projection): We define a projection

$$\cdot|_\cdot : \Sigma_{\tau \in \mathit{Types}} [\![\tau]\!] \times \mathbb{N}^* \to \Sigma_{\tau \in \mathit{Types}} [\![\tau]\!]$$

that takes a term and a position and projects to the corresponding sub-term.

For all $\langle v_1, \ldots, v_n \rangle \in [\![\tau_1 \times \cdots \times \tau_n]\!]$ and all valid positions $i.p$ of $\langle v_1, \ldots v_n \rangle$ we define the projection as follows.

$$\langle v_1, \ldots, v_n \rangle|_{i.p} = v_i|_p$$

Note that we do not have to define the projection to the root position as ϵ is not a valid position of a tuple, in other words, we have $\epsilon \notin \mathit{Pos}\ \langle v_1, \ldots, v_n \rangle$.

For the elements of the interpretation of the type *Bool* we define a projection to a valid position as follows.

$$\bot_{\mathit{Bool}}|_\epsilon = \bot_{\mathit{Bool}} \qquad \mathit{False}|_\epsilon = \mathit{False} \qquad \mathit{True}|_\epsilon = \mathit{True}$$

Let $v_1 \in [\![\mathit{List}\ \tau]\!]$, $v_2 \in [\![\tau]\!]$, and $p \in \mathit{Pos}\ (\mathit{Cons}\ \langle v_1, v_2\rangle)$. We define the projection to a valid position of a list as follows.

$$\bot_{\mathit{List}\ \tau}|_\epsilon = \bot_{\mathit{List}\ \tau} \qquad\qquad \mathit{Nil}_\tau|_\epsilon = \mathit{Nil}_\tau$$

$$(\mathit{Cons}\ \langle v_1, v_2\rangle)|_p = \begin{cases} \mathit{Cons}\ \langle v_1, v_2\rangle & \text{if } p = \epsilon \\ v_1|_q & \text{if } p = 1.q \\ v_2|_q & \text{if } p = 2.q \end{cases}$$

If we project to a non-valid position, the projection yields $\lightning$. That is, for all $v \in \Sigma_{\tau \in \mathit{Types}}[\![\tau]\!]$ and $p \in \mathbb{N}^* \setminus (\mathit{Pos}\, v)$ we define $v|_p = \lightning$. We, again, use the least solution of the recursive definition of $\cdot|_\cdot$. □

We give some examples of projections in the following.

Example 4.2.3: We consider the value $\langle \mathit{False}, \mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \mathit{Nil}_{\mathit{List\ Bool}}\rangle\rangle$ and abbreviate it to v in the following. Example 4.2.2 presents the set of valid positions for this value. We have

$$v|_1 = \mathit{False}, \qquad v|_2 = \mathit{Cons}\ \langle \bot_{\mathit{Bool}}, \mathit{Nil}_{\mathit{List\ Bool}}\rangle, \qquad \text{and} \qquad v|_{2.2} = \mathit{Nil}_{\mathit{List\ Bool}}.$$

Furthermore, we have $v|_\epsilon = \lightning$, $v|_{2.3} = \lightning$, and $v|_{2.1.1} = \lightning$ because ϵ, 2.3, and 2.1.1 are not valid positions of v. □

In the same manner as we have defined the projection function we define a substitution of type $\Sigma_{\tau \in \mathit{Types}}[\![\tau]\!] \times \Sigma_{\tau \in \mathit{Types}}[\![\tau]\!] \times \mathbb{N}^* \to \Sigma_{\tau \in \mathit{Types}}[\![\tau]\!]$ that takes two terms e_1 and e_2 and a position p and substitutes a sub-term of the first term by the second term denoted by $e_1[e_2]_p$.

Definition 4.2.5 (Substitution): We define a substitution

$$\cdot[\cdot]_\cdot : \Sigma_{\tau \in \mathit{Types}}[\![\tau]\!] \times \Sigma_{\tau \in \mathit{Types}}[\![\tau]\!] \times \mathbb{N}^* \to \Sigma_{\tau \in \mathit{Types}}[\![\tau]\!].$$

Let $\langle v_1, \ldots, v_n\rangle \in [\![\tau_1 \times \cdots \times \tau_n]\!]$ and $i.p$ be a valid position of $\langle v_1, \ldots, v_n\rangle$. Because $i.p$ is a valid position, there exists τ' such that $\langle v_1, \ldots, v_n\rangle|_{i.p} \in [\![\tau']\!]$. For all $v \in [\![\tau']\!]$ we define the substitution of a valid position of a tuple as follows.

$$\langle v_1, \ldots, v_n\rangle[v]_{i.p} = v_i[v]_p$$

For all values v of $[\![\mathit{Bool}]\!]$ we define the substitution of a valid position of an element of the interpretation of *Bool* as follows.

$$\bot_{\mathit{Bool}}[v]_\epsilon = v \qquad\qquad \mathit{False}[v]_\epsilon = v \qquad\qquad \mathit{True}[v]_\epsilon = v$$

Let $v \in [\![\mathit{List}\ \tau]\!]$. We define the substitution of a list as follows.

$$\bot_{\mathit{List}\ \tau}[v]_\epsilon = v \qquad\qquad \mathit{Nil}_\tau[v]_\epsilon = v$$

Furthermore, let $v_1 \in [\![\tau]\!]$, $v_2 \in [\![List\ \tau]\!]$, and $p \in Pos\,(Cons\,\langle v_1, v_2\rangle)$. Because p is a valid position, there exists τ' such that $(Cons\,\langle v_1, v_2\rangle)|_p \in [\![\tau']\!]$. For all $v \in [\![\tau']\!]$ we define the substitution of a valid position of a list by v as follows.

$$(Cons\,\langle v_1, v_2\rangle)[v]_p = \begin{cases} v & \text{if } p = \epsilon \\ Cons\,\langle v_1[v]_q, v_2\rangle & \text{if } p = 1.q \\ Cons\,\langle v_1, v_2[v]_q\rangle & \text{if } p = 2.q \end{cases}$$

If we substitute a non-valid position or the second argument has the wrong type, the substitution yields $\bot$. More precisely, for all $v, v' \in \Sigma_{\tau \in Types}[\![\tau]\!]$, and $p \in \mathbb{N}$ we define $v[v']_p = \bot$ if $p \notin Pos\,v$. Furthermore, if $v|_p \in [\![\tau']\!]$ and $v' \notin [\![\tau']\!]$ we define $v[v']_p = \bot$. To solve the recursive definition we, again, use its least solution. □

We give some examples of substitutions in the following.

Example 4.2.4: We consider the value $\langle False, Cons\,\langle \bot_{Bool}, Nil_{List\ Bool}\rangle\rangle$ and abbreviate it to v in the following. Example 4.2.2 presents to set of valid positions for this value. We have

$$\begin{aligned} v[True]_1 &= \langle True, Cons\,\langle \bot_{Bool}, Nil_{List\ Bool}\rangle\rangle, \\ v[\bot_{List\ Bool}]_2 &= \langle False, \bot_{List\ Bool}\rangle, \end{aligned}$$

and

$$v[Cons\,\langle False, \bot_{List\ Bool}\rangle]_{2.2} = \langle False, Cons\,\langle \bot_{Bool}, Cons\,\langle False, \bot_{List\ Bool}\rangle\rangle\rangle.$$

For all $v' \in \Sigma_{\tau \in Types}[\![\tau]\!]$ we have $v[v']_\epsilon = \bot$, $v[v']_{2.3} = \bot$, and $v[v']_{2.1.1} = \bot$ because ϵ, 2.3, and 2.1.1 are not valid positions of v. Furthermore, we have $v[Nil_{Bool}]_1 = \bot$ because $v|_1 = False \in [\![Bool]\!]$ and $Nil_{Bool} \in [\![List\ Bool]\!]$. □

The following lemma states some properties about projections and substitutions that are employed in several of the following proofs.

Lemma 4.2.1: *Projections and substitutions satisfy the following properties.*

- *For all positions p the function $\cdot|_p : \Sigma_{\tau \in Types}[\![\tau]\!] \to \Sigma_{\tau \in Types}[\![\tau]\!]$ is continuous.*
- *For all positions p the function $\cdot[\cdot]_p : \Sigma_{\tau \in Types}[\![\tau]\!] \times \Sigma_{\tau \in Types}[\![\tau]\!] \to \Sigma_{\tau \in Types}[\![\tau]\!]$ is continuous.*
- *For all positions p and all $v \in [\![\tau]\!]$ the function $v[\cdot]_p : \Sigma_{\tau \in Types}[\![\tau]\!] \to \Sigma_{\tau \in Types}[\![\tau]\!]$ is strictly increasing if the results differ from $\bot$.*

Note that continuity of $\cdot[\cdot]_p$ means that for all chains $\langle v_i\rangle_{i \in I}$ and $\langle v'_i\rangle_{i \in I}$ we have

$$(\bigsqcup_{i \in I} v_i)[\bigsqcup_{i \in I} v'_i]_p = \bigsqcup_{i \in I}(v_i[v'_i]_p).$$

Furthermore, note that continuity of $\cdot|_p$ implies monotonicity of $\cdot|_p$ and continuity of $\cdot[\cdot]_p$ implies monotonicity of $\cdot[v]_p$ for a fixed $v \in [\![\tau]\!]$. Finally, note that the restriction

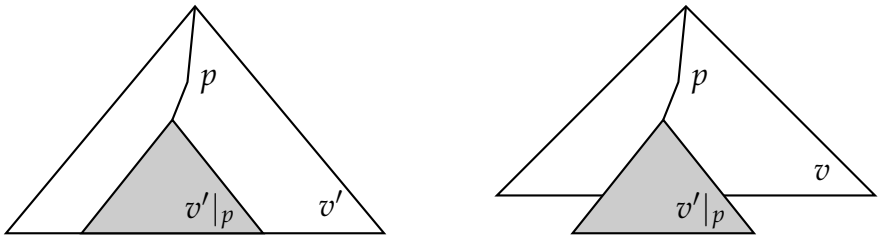


Figure 4.2.2: Graphics illustrating $v'|_p$ and $v[v'|_p]_p$

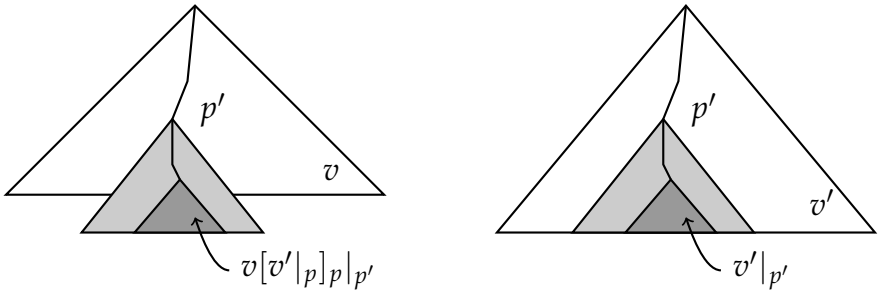


Figure 4.2.3: Graphics illustrating $(v[v'|_p]_p)|_{p'} = v'|_{p'}$ if $p' \geq p$

to non-$\bot$ results in the last property in fact restricts the property to valid arguments with respect to position and type. To extend the presented approach to other data types the corresponding projections and substitutions have to satisfy the properties presented in Lemma 4.2.1 as well.

Besides these semantics related properties, there are some properties that specify a connection between projections and substitutions. The first property states that projections and substitutions are a kind of inverse to each other in the sense that we have $v[v|_p]_p = v$ for all values of some interpretation $[\![\tau]\!]$ and valid positions p. The second property states that $p' \geq p$ implies $(v[v'|_p]_p)|_{p'} = v'|_{p'}$. We informally illustrate this equation in the following.

Consider the graphic on the left-hand side of Figure 4.2.2. The white triangle represents the term v', and the gray triangle represents the sub-term at position p, that is, $v'|_p$. In the graphic on the right-hand side the white triangle represents another term v. We replace the sub-term of v at position p by $v'|_p$. That is, at position p we insert the gray triangle into the white triangle. Now consider that we project to a position p' with $p' \geq p$ in the term $v[v'|_p]_p$. On the left-hand side of Figure 4.2.3 this sub-term is represented by the dark gray triangle. As we have $p \leq p'$ (p is a prefix of p'), we project to a term that is a sub-term of the light gray triangle. Thus, we get the same result if we project to position p' of the original term where the light gray triangle is taken from. In other words we have $(v[v'|_p]_p)|_{p'} = v'|_{p'}$.

We can state a similar result if we finally project to a position that is neither a prefix nor a suffix of the original position. That is, we consider $(v[v']_p)|_{p'}$ and we have neither $p' \geq p$ nor $p \geq p'$. Note that we replaced the innermost $v'|_p$ of the previous property by v' here as this third property holds in this more general con-

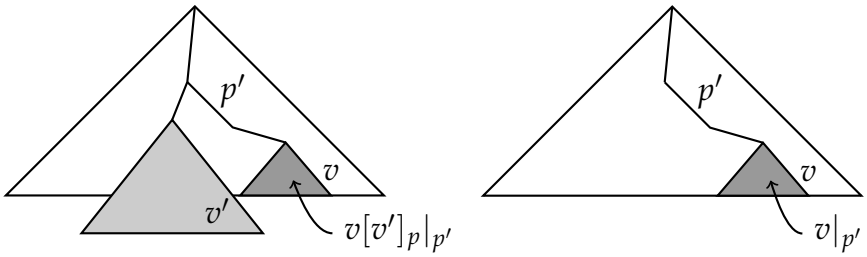


Figure 4.2.4: Graphics illustrating $(v[v']_p)|_{p'} = v|_{p'}$ if $p' \not\geq p$ and $p \not\geq p'$

text. Therefore, the light gray triangle on the left-hand side of Figure 4.2.4 represents some arbitrary term v' instead of $v'|_p$ as it was the case before. As the graphic on the lefthand side of Figure 4.2.4 illustrates, because we project to a position incomparable with p, there exists some point where we leave the path defined by p. Thus, we end up in a part of the term v that is disjunct to the light gray term. In this case it does not matter whether v contains the light gray term or not. We can project to position p' in the term v instead of projecting to p' in $v[v'|_p]_p$. The following lemma states the properties we have just illustrated.

Lemma 4.2.2: *Projections and substitutions satisfy the following properties.*

- *For all positions p and all $v \in [\![\tau]\!]$ we have $v[v|_p]_p = v$.*
- *For all positions p, p' and all $v, v' \in [\![\tau]\!]$ the following statements hold.*
  - *If $p' \geq p$, then we have $(v[v'|_p]_p)|_{p'} = v'|_{p'}$.*
  - *If $p' \not\geq p$ and $p \not\geq p'$, then we have $(v[v']_p)|_{p'} = v|_{p'}$.*

We use positions and projections to state whether a function is sequential. In the following we use $\bot$ to denote $\bot_\tau$ for some type τ. In particular this means that we have $\bot \neq \underline{\bot}$. In other words, in the following we only consider projections to valid positions, but for simplicity we do not always state this condition explicitly. The following lemma is an instance of the definition of sequentiality.

Lemma 4.2.3 (Sequential Function): *In a sequential language the following holds. Let $f \in [\![\sigma \to \tau]\!]$, pv be a partial value of type σ and $rp \in Pos\ (f\ pv)$ such that $(f\ pv)|_{rp} = \bot$. Then, there exists $p \in Pos\ pv$ such that $pv|_p = \bot$, and for all values pv' of type σ the following holds.*

$$pv' \sqsupseteq pv \wedge pv'|_p = \bot \Longrightarrow (f\ pv')|_{rp} = \bot$$

Note that the previous lemma considers partial values pv of type τ, that is, elements of $[\![\tau]\!] \setminus [\![\tau]\!]\uparrow$. Therefore, a position p with $pv|_p = \bot$ always exists.

Definition 4.2.6 (Sequential/Non-Sequential Position): We call position p of Lemma 4.2.3 a sequential position in pv at result position rp with respect to f.

We call a position p a non-sequential position in pv at result position rp with respect to f if $(f\ pv)|_{rp} = \bot$ and $p \in Pos\ pv$ such that $pv|_p = \bot$, and there exists a value pv' of type τ with the following property.

$$pv' \sqsupseteq pv \wedge pv'|_p = \bot \wedge (f\ pv')|_{rp} \neq \bot \qquad \square$$

Note that a position might be neither sequential nor non-sequential. However, if pv is a partial value and $(f\ pv)|_{rp} = \bot$, then all positions $p \in Pos\ pv$ with $pv|_p = \bot$ are either sequential or non-sequential.

Example 4.2.5: We consider the Boolean conjunction *andL* from Example 2.2.1 again. We have $(\llbracket andL \rrbracket\ \langle \bot, \bot \rangle)|_\epsilon = \bot$ and can apply Lemma 4.2.3. It states that there is a sequential position p in $\langle \bot, \bot \rangle$ at result position ϵ. Because we need $\langle \bot, \bot \rangle|_p = \bot$, the only sequential position candidates are position 1 and position 2, that is, the first and the second argument of *andL*. As Lemma 4.2.3 guarantees the existence of a sequential position, we have

$$\forall v \in \llbracket Bool \rrbracket.\ (\llbracket andL \rrbracket\ \langle v, \bot \rangle)|_\epsilon = \bot \quad \text{or} \quad \forall v \in \llbracket Bool \rrbracket.\ (\llbracket andL \rrbracket\ \langle \bot, v \rangle)|_\epsilon = \bot.$$

As we have $(\llbracket andL \rrbracket\ \langle False, \bot \rangle)|_\epsilon = False \neq \bot$, the first formula of the disjunction is false and, hence, the second formula of the disjunction is true. Thus, position 1 is a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ and positions 2 is a non-sequential position in $\langle \bot, \bot \rangle$ at result position ϵ. □

The first argument of *andL* is a sequential position in $\langle \bot, \bot \rangle$ because the function performs pattern matching on its first argument. In fact, there is a close relation between the argument positions a function performs pattern matching on and sequential positions. In the following we consider the position of an argument that is demanded by a function if we demand a specific result position. A value is demanded if it is evaluated to head normal form.

Example 4.2.6: Haskell provides a function *error* :: *String* $\to \alpha$ that takes a string and yields a run-time error if it is evaluated. If an application of *error* is evaluated, the execution is stopped and the corresponding string is printed. We can use this mechanism to observe, which argument of a function is demanded if we demand the result. For example, consider the evaluation of the expression *error* `"1"` && *error* `"2"` in GHCi, the interpreter of the Glasgow Haskell Compiler (GHC).

```
> error "1" && error "2"
*** Exception: 1
```

As the error message "1" is printed, we know that && demands its first argument if we demand the expression *error* `"1"` && *error* `"2"`. By employing exception handling we can even employ this information in Haskell itself.

If we replace the argument that is demanded by a non-erroneous value, the function might demand another argument. For example, the function && demands its second argument if we demand the expression *True* && *error* `"2"`.

```
> True && error "2"
*** Exception: 2
```

On the other hand, if the demanded argument is an error the function still evaluates this error independent of the other arguments. For example, if we demand the expression *error* `"1"` && *True* the function && still demands its first argument.

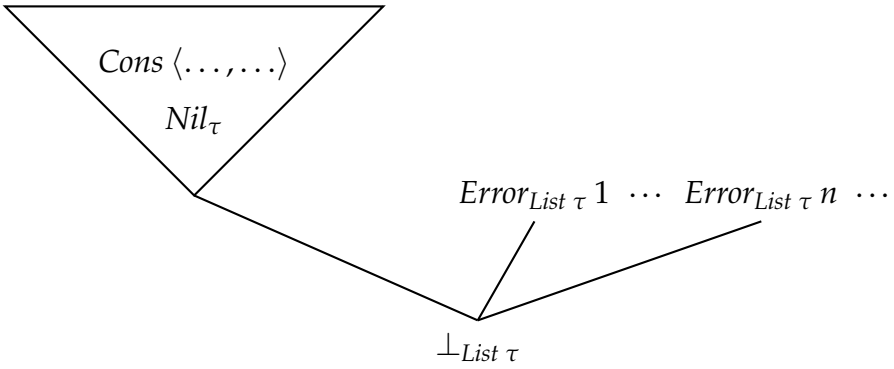


Figure 4.2.5: The Structure of $[\![List\ \tau]\!]^{Err}$

```
> error "1" && True
*** Exception: 1
```

Note that this behavior is closely connected to the existence of a sequential position. With respect to demanded positions, errors take over the role of least elements in the context of sequential positions.

In the previous examples the demand is caused by pattern matching on the corresponding argument position. A function can also demand an argument position by projecting to it. For example, consider the Haskell function $const :: \alpha \to \beta \to \alpha$, which takes two arguments and projects to its first argument. If we evaluate the expression $const\ (error\ \texttt{"1"})\ (error\ \texttt{"2"})$ in GHCi, we can observe that *const* demands its first argument if we demand its result.

```
> const (error "1") (error "2")
*** Exception: 1
```

In this case the first argument is demanded because we demand the result and *const* projects to its first argument. □

In the following, we refer to an argument position at which a value is demanded if we demand a value at a certain result position as demanded position. To give a more precise definition of demanded position and to show a connection between demanded positions and sequential positions, we sketch how the semantics, presented in Figure 2.2.4, can be extended with exceptions in the following. We denote the type and term semantics with exceptions by $[\![\cdot]\!]^{Err}$.

For simplicity we do not use strings for error messages but natural numbers. Furthermore, we do neither add a primitive to raise an exception like the Haskell function $error :: String \to \alpha$ nor a primitive to catch such exceptions as we do not need these in the following. However, adding these functions as primitives is straightforward.

We add an additional constructor $Error_\tau$ that takes a natural number as argument to the interpretation of the Boolean type and the list types. In most cases we omit the subscript type of $Error_\tau$ as it is determined by the context. Figure 4.2.5 illustrates

$$[\![\textbf{case } e \textbf{ of } \{\mathit{False} \to e_1; \mathit{True} \to e_2\}]\!]^{Err}_a = \begin{cases} \bot & \text{if } [\![e]\!]^{Err}_a = \bot \\ \mathit{Error}\ n & \text{if } [\![e]\!]^{Err}_a = \mathit{Error}\ n \\ [\![e_1]\!]^{Err}_a & \text{if } [\![e]\!]^{Err}_a = \mathit{False} \\ [\![e_2]\!]^{Err}_a & \text{if } [\![e]\!]^{Err}_a = \mathit{True} \end{cases}$$

$$[\![\textbf{case } e \textbf{ of } \{\mathit{Nil}_\tau \to e_1; \mathit{Cons}\langle x_1, x_2\rangle \to e_2\}]\!]^{Err}_a = \begin{cases} \bot & \text{if } [\![e]\!]^{Err}_a = \bot \\ \mathit{Error}\ n & \text{if } [\![e]\!]^{Err}_a = \mathit{Error}\ n \\ [\![e_1]\!]^{Err}_a & \text{if } [\![e]\!]^{Err}_a = \mathit{Nil}_\tau \\ [\![e_2]\!]^{Err}_{a[x_1 \mapsto v_1, x_2 \mapsto v_2]} & \text{if } [\![e]\!]^{Err}_a = \mathit{Cons}\ \langle v_1, v_2\rangle \end{cases}$$

Figure 4.2.6: Term Semantics for Case Expressions in a Language with Exceptions

the structure of the resulting interpretation of a list type. All exceptions are more defined than the least element $\bot_{\mathit{List}\ \tau}$, but exceptions are incomparable to each other and incomparable to the other elements of the domain. Note that v_1 as well as v_2 might be exceptions if we consider a value of the form $\mathit{Cons}\ \langle v_1, v_2\rangle$.

Basically, the exception semantics handles exceptions in the same way as it handles the least element. That is, if we perform pattern matching on an exception we yield a corresponding exception as result. However, in contrast to $\bot$, we propagate the number that is stored in an exception. Figure 4.2.6 presents the semantics for case expressions. We have to add rules to the interpretation of case expressions that apply if the scrutinee is an exception. In this case we wrap the number that indicates the kind of exception in an *Error* constructor of appropriate type. For all other expressions we take the definition of $[\![\cdot]\!]$ over but replace all occurrences of $[\![\cdot]\!]$ by $[\![\cdot]\!]^{Err}$.

The following example illustrates the semantics of the functions *andL* and a constant function with respect to the exception semantics. Furthermore, we exemplarily show that the exception semantics models the behavior as we have observed for Haskell functions in Example 4.2.6.

Example 4.2.7: Consider the function *andL* from Example 2.2.1. In the exception semantics, presented in Figure 4.2.6, we get the following semantics for *andL*.

$$[\![\mathit{andL}]\!]^{Err}_a = \lambda\langle v_1, v_2\rangle.\begin{cases} \bot & \text{if } v_1 = \bot \\ \mathit{Error}\ n & \text{if } v_1 = \mathit{Error}\ n \\ \mathit{False} & \text{if } v_1 = \mathit{False} \\ v_2 & \text{if } v_1 = \mathit{True} \end{cases}$$

If we apply $[\![\mathit{andL}]\!]^{Err}$ to $\langle \mathit{Error}\ 1, \mathit{Error}\ 2\rangle$, we get $\mathit{Error}\ 1$, and, if we apply $[\![\mathit{andL}]\!]^{Err}$ to $\langle \mathit{True}, \mathit{Error}\ 2\rangle$, we get $\mathit{Error}\ 2$. That is, the semantics of *andL* shows the same behavior we have observed for the Haskell counterpart (&&) in Example 4.2.6.

Let us consider the counterpart of the Haskell function *const* in our simple functional language. As the simple language does not provide polymorphism, we consider the following monomorphic Boolean instance of *const*.

$$const :: Bool \times Bool \to Bool$$
$$const\langle x, y\rangle = x$$

In the exception semantics we get the following semantics for *const*.

$$[\![const]\!]_a^{Err} = \lambda\langle v_1, v_2\rangle.v_1$$

Thus, we have $[\![const]\!]^{Err}\ \langle Error\ 1, Error\ 2\rangle = Error\ 1$, that is, *const* demands its first argument if we demand its result. □

Definition 4.2.8 uses the exception semantics, sketched in Figure 4.2.6, to define demanded positions. Analogous to Example 4.2.6 we apply a function to a value where the number stored in an exception uniquely identifies its position within the value. If we apply a function to a value of this kind and get an exception at a certain result position, then the position in the argument where the exception is coming from is called a demanded position with respect to the corresponding result position.

Because we want to show a correspondence between demanded and sequential positions, we establish a relation between demanded positions and the original semantics $[\![\cdot]\!]$. Therefore, we define a logical relation that relates elements of $[\![\tau]\!]^{Err}$ with elements of $[\![\tau]\!]$, that is, with elements that do not contain any exceptions. The idea is basically that the semantics $[\![\cdot]\!]$ and $[\![\cdot]\!]^{Err}$ behave equally except for exceptions *Error n* as these are not present in the domains of $[\![\cdot]\!]$. Furthermore, the semantic function $[\![\cdot]\!]^{Err}$ treats exceptions *Error n* in the same way as $[\![\cdot]\!]$ treats the least element $\bot$.

A logical relation (Girard 1972; Mitchell and Meyer 1985; Honsell and Sannella 1999) is a family of relations indexed by types, that is, $(R_\tau)_{\tau \in Types}$. For each type τ the relation R_τ relates two elements of the interpretation of τ. The family of relations is defined inductively over the type structure. Furthermore, if $(R_\tau)_{\tau \in Types}$ is a logical relation, then we have $(f, g) \in R_{\sigma\to\tau}$ if and only if for all $(x, y) \in R_\sigma$ we have $(f\ x, g\ y) \in R_\tau$. In other words, two functions are related by a logical relation if and only if the functions yield related results for all related arguments.

Intuitively, the following logical relation formalizes that the semantics $[\![\cdot]\!]^{Err}$ treats exceptions in the same way as the semantics $[\![\cdot]\!]$ treats the least element $\bot$. The logical relation err_τ^m where m is a natural number relates an element v_1 of $[\![\tau]\!]^{Err}$ with an element v_2 of $[\![\tau]\!]$ if the value v_2 contains $\bot$ at all positions that contain the exception *Error m* in v_1. Furthermore, the value v_2 contains an arbitrary value of type τ at all positions that contain an exception *Error n* with $m \neq n$ in v_1. That is, the relation err_m^τ is a kind of $\sqsubseteq$ that treats exceptions like the least element $\bot$. However, all exceptions with a specific label, namely, with label m, have to become $\bot$.

Definition 4.2.7 (Logical Relation err_τ^m): We define a family of logical relations indexed by a natural number and a type. For $m \in \mathbb{N}$ and $\tau \in Types$ we define

$err^m_\tau \subseteq [\![\tau]\!]^{Err} \times [\![\tau]\!]$ as follows.

$$\begin{aligned}
err^m_{Bool} &= \{(\bot, b) \mid b \in [\![Bool]\!]\} \\
&\quad \cup \{(Error\ n, b) \mid b \in [\![Bool]\!], m \neq n\} \\
&\quad \cup \{(Error\ m, \bot)\} \\
&\quad \cup \{(False, False), (True, True)\} \\
err^m_{List\ \tau} &= \{(\bot, l) \mid l \in [\![List\ \tau]\!]\} \\
&\quad \cup \{(Error\ n, l) \mid l \in [\![List\ \tau]\!], m \neq n\} \\
&\quad \cup \{(Error\ m, \bot)\} \\
&\quad \cup \{(Nil, Nil)\} \\
&\quad \cup \{(Cons\ \langle x, xs\rangle, Cons\ \langle y, ys\rangle) \mid (x, y) \in err^m_\tau, (xs, ys) \in err^m_{List\ \tau}\}
\end{aligned}$$

We use the least solution of the recursive equation in the definition of $err^m_{List\ \tau}$.

Two tuples are related by $err^m_{\tau_1 \times \cdots \times \tau_n}$ if all their components are pointwise related by relations of corresponding types, that is, we define $err^m_{\tau_1 \times \cdots \times \tau_n}$ as follows.

$$err^m_{\tau_1 \times \cdots \times \tau_n} = \{(\langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_n\rangle) \mid \forall i \in \{1, \ldots, n\}.\, (x_i, y_i) \in err^m_{\tau_i}\}$$

Finally, two functions are related if the images of related arguments are related, that is, we define $err^m_{\sigma \to \tau}$ as follows.

$$err^m_{\sigma \to \tau} = \{(f, g) \mid \forall x \in [\![\sigma]\!]^{Err}, y \in [\![\sigma]\!].\, (x, y) \in err^m_\sigma \Longrightarrow (f\ x, g\ y) \in err^m_\tau\}$$

Note that the definition of $err^m_{\sigma \to \tau}$ is determined by the demands on a logical relation. Furthermore, note that $err^m_{\sigma \to \tau}$ relates a function that handles exceptions, that is, an element of $[\![\sigma \to \tau]\!]^{Err}$, with a function that does not handle exceptions, that is, an element of $[\![\sigma \to \tau]\!]$. □

In the following we define when we call a position a demanded position by means of the exception semantics. We use the logical relation err^n_σ to define demanded positions with respect to the original semantics $[\![\cdot]\!]$. The following definition considers some global program P that is well-typed with respect to the typing rules from Section 2.2.

Definition 4.2.8 (Demanded Position): Let $f :: \sigma \to \tau \in P$ and $pv \in [\![\sigma]\!]$. If there exists $ev \in [\![\sigma]\!]^{Err}$ and $n \in \mathbb{N}$ such that $(ev, pv) \in err^n_\sigma$ and

$$\exists!\ p \in Pos\ ev.\ ev|_p = Error\ n$$

as well as

$$([\![f]\!]^{Err}\ ev)|_{rp} = Error\ n,$$

we call p a demanded position in $pv \in [\![\sigma]\!]$ at result position rp with respect to f. □

In other words, we consider a partial value ev that contains an exception $Error\ n$ at a unique position p. If $[\![f]\!]^{Err}$ projects the exception $Error\ n$ to the result position rp,

then we call p a demanded position at result position rp. The following example presents demanded positions of *andL*.

Example 4.2.8: Consider the exception semantics of *andL* from Example 4.2.7 again. We have

$$[\![andL]\!]^{Err}\ \langle Error\ 1, Error\ 2\rangle = Error\ 1$$

and

$$(\langle Error\ 1, Error\ 2\rangle, \langle \bot, \bot\rangle) \in err^1_{Bool\times Bool}$$

and, thus, position 1 is a demanded position in $\langle \bot, \bot\rangle$ at result position ϵ with respect to *andL*. As the logical relation $err^1_{Bool\times Bool}$ allows us to replace occurrences of exceptions *Error n* with $n \neq 1$ by arbitrary values of the appropriate type, we have

$$(\langle Error\ 1, Error\ 2\rangle, \langle \bot, True\rangle) \in err^1_{Bool\times Bool}.$$

Thus, position 1 is also a demanded position in $\langle \bot, True\rangle$ at result position ϵ with respect to *andL*. Furthermore, we have

$$[\![andL]\!]^{Err}\ \langle True, Error\ 2\rangle = Error\ 2$$

and

$$\langle True, Error\ 2\rangle, \langle True, \bot\rangle) \in err^2_{Bool\times Bool}$$

and, thus, position 2 is a demanded position in $\langle True, \bot\rangle$ at result position ϵ with respect to *andL*. □

To prove a connection between demanded and sequential positions we use two utilities. The first utility is called the Basic Lemma or Fundamental Theorem in the context of logical relations. Mostly, logical relations are defined for a simply typed lambda calculus with an explicit fixpoint combinator. In this setting the following lemma simply holds for all logical relations. As we do not consider a lambda calculus but a language with data structures and explicit recursion, we prove the following lemma in Appendix A.

We consider an expression e of some type τ and the interpretation of e with respect to environments a_1 and a_2, that is, $[\![e]\!]^{Err}_{a_1}$ and $[\![e]\!]_{a_2}$. The lemma states that these interpretations are related by err^m_τ if the images of a_1 and a_2 are related by $err^m_{\tau'}$ for all variables. Here the type τ' is the type of the corresponding variable. We use the following lemma to prove that every demanded position is also a sequential position.

Lemma 4.2.4: *Assume $\Gamma \vdash e :: \tau$ and let a_1 and a_2 be environments such that for all $x :: \tau'$ in Γ we have $(a_1(x), a_2(x)) \in err^m_{\tau'}$. In this case we have $([\![e]\!]^{Err}_{a_1}, [\![e]\!]_{a_2}) \in err^m_\tau$.*

The previous lemma in particular states that for a function $f :: \sigma \to \tau$ we have $([\![f]\!]^{Err}_a, [\![f]\!]_a) \in err^m_{\sigma\to\tau}$ for all $m \in \mathbb{N}$. By means of this statement we get a connection between the interpretation of a function with respect to the two semantics. In addition, we employ a connection between err^m_τ and the projection defined in Definition 4.2.4. Therefore, we extend projections from $[\![\tau]\!]$ to $[\![\tau]\!]^{Err}$ by handling values

of the form *Error n* in the same way as we handle the least element $\bot$. By means of this extension of the projections we can characterize values $ev \in [\![\tau]\!]^{Err}$ and $pv \in [\![\tau]\!]$ with $(ev, pv) \in err_\tau^m$ as follows. For all positions p we have

$$ev|_p = Error\ m \Longrightarrow pv|_p = \bot$$

as well as

$$m \neq n \wedge ev|_p = Error\ n \implies pv|_p \sqsupseteq \bot.$$

These implications provide a kind of formalisation of the intuitive interpretation of the logical relation err_τ^m. In other words, while we may replace occurrences of the exception *Error m* in *ev* by $\bot$, we may replace occurrences of other exceptions *Error n* with $m \neq n$ by arbitrary values.

On the basis of these observations we can show that every demanded position in *pv* at result position rp with respect to f is a sequential position in *pv* at result position rp with respect to $[\![f]\!]$. Note that we have defined demanded positions with respect to a syntactic function $f :: \sigma \to \tau$ while we have defined sequential positions with respect to a semantic function $f \in [\![\sigma \to \tau]\!]$.

Lemma 4.2.5: *Let $f :: \sigma \to \tau \in P$. If p is a demanded position in pv at result position rp with respect to f, then p is a sequential position in pv at result position rp with respect to $[\![f]\!]$.*

The previous lemma shows that the set of demanded positions is a subset of the set of sequential positions. However, there are sequential positions that are not demanded positions. While with respect to some value at every result position there is always only a single demanded position, there might be several sequential positions at the same result position.

Example 4.2.9: Consider the following definition of a projection to the second argument that is strict in its first argument.

$$\begin{array}{l} strictSnd :: Bool \times Bool \to Bool \\ strictSnd\langle x, y\rangle = \\ \quad \textbf{case}\ x\ \textbf{of} \\ \qquad\quad False \to y \\ \qquad\quad True\ \to y \end{array}$$

We get the following semantics for *strictSnd* with respect to the original semantics. Note that we have combined the cases $v_1 = False$ and $v_2 = True$ to a single case.

$$[\![strictSnd]\!] = \lambda\langle v_1, v_2\rangle. \begin{cases} \bot & \text{if } v_1 = \bot \\ v_2 & \text{otherwise} \end{cases}$$

With respect to this semantics we have

$$([\![strictSnd]\!]\ \langle \bot, b\rangle)|_\epsilon = \bot$$

for all values b of type *Bool*. Likewise, we have

$$(\llbracket strictSnd \rrbracket\ \langle b, \bot \rangle)|_\epsilon = \bot$$

for all values b of type *Bool*. Thus, position 1 as well as position 2 are sequential positions in $\langle \bot, \bot \rangle$ at position ϵ with respect to *strictSnd*.

Now, let us consider the exception semantics of *strictSnd*.

$$\llbracket strictSnd \rrbracket^{Err} = \lambda\langle v_1, v_2 \rangle. \begin{cases} \bot & \text{if } v_1 = \bot \\ Error\ n & \text{if } v_1 = Error\ n \\ v_2 & \text{otherwise} \end{cases}$$

With respect to this semantics we have

$$\llbracket strictSnd \rrbracket^{Err}\ \langle Error\ 1, Error\ 2 \rangle = Error\ 2$$

and

$$(\langle Error\ 1, Error\ 2 \rangle, \langle \bot, \bot \rangle) \in err^2_{\langle Bool, Bool \rangle}.$$

Therefore, position 2 is a demanded position in $\langle \bot, \bot \rangle$ at position ϵ with respect to *strictSnd*. □

The exception semantics presented here does not match the semantics of Haskell exceptions (Peyton Jones et al. 1999). The exceptions employed by Haskell are called imprecise exceptions. The semantics of these exceptions is less restrictive to guarantee that certain transformations are semantic-preserving. The basic idea behind imprecise exceptions is to use sets of exceptions. More precisely, instead of a single natural number an exception contains a set of natural numbers where the set denotes the set of possible exceptions that might be raised by the evaluation of an expression. For example, consider the Haskell function *head*, which is defined as follows.

```
head :: [α] → α
head []     = error "empty list"
head (x: _) = x
```

If we apply *head* to an exception like *error* `"argument"`, in the imprecise exceptions semantics we get a set containing the exception `"argument"` as well as the exception `"empty list"`. If we catch the exception by employing the Haskell function *catch*, it choses one of the possible exceptions non-deterministically. Therefore, *catch* is in the *IO* monad as it can be considered to ask some kind of oracle, which exception to choose.

As long as the compiler does not apply any optimizations, in practice, Haskell exceptions behave in the way specified by the exception semantics $\llbracket \cdot \rrbracket^{Err}$. In the same manner as we assume a simplified exception semantics, the behavior of the black-box testing tool Lazy Smallcheck by Runciman et al. (2008) is based on this simplified exception model. Lazy Smallcheck employs exception handling mecha-

nisms to implement a kind of needed narrowing (Antoy et al. 1994) to systematically generate test cases for property-based testing.

We will reconsider demanded positions when we present the implementation of Sloth. There, we are interested in demanded positions because they are an easy way to identify sequential positions of a function. For now, we switch to the semantics $[\![\cdot]\!]$ again and deal with the question when a function is minimally strict.

4.3 Minimally Strict Functions

In this section we employ the definition of sequentiality, presented in the previous section, to define a criterion to check whether a sequential function is minimally strict. When we only consider sequential functions, we are looking for minimally strict functions and not for a least strict function.

We start by providing a formal definition of "minimally strict".

Definition 4.3.1: A sequential function $f \in [\![\sigma \to \tau]\!]$ is minimally strict if for all sequential functions $g \in [\![\sigma \to \tau]\!]$ the relation $g \preceq f$ implies $g = f$. □

In order to provide a criterion that states whether a function is minimally strict we start with some preliminaries. By Σ_C we denote the set of semantic constructor symbols. As we only consider Booleans and lists as algebraic data types, we have $\Sigma_C = \{\mathit{False}, \mathit{True}, \mathit{Nil}, \mathit{Cons}\}$. The refinement of a partial value is the replacement of a subterm that is $\bot$ by a partial value of the form $C\ \bot$ where C is a constructor from Σ_C. We consider two kinds of refinements: a refinement at a sequential position and a refinement at a non-sequential position.

Definition 4.3.2 (Refinement Relations): For all values v of type τ and all positions $p \in \mathit{Pos}\, v$ with $v|_p = \bot$ and $v|_p \in [\![\tau']\!]$ we define the refinement of v as follows. We have $v \lessdot_p v'$ if there exists $C \in \Sigma_C$ such that $C \in [\![\sigma \to \tau']\!]$ and $v' = v[C\ \bot]_p$. Note that, if C takes multiple arguments, then $\sigma = \tau_1 \times \cdots \times \tau_n$ and $\bot = \langle \bot_{\tau_1}, \ldots, \bot_{\tau_n} \rangle$.

On basis of this definition and the notion of sequential and non-sequential positions we define two refinement relations. We relate values v and v' of type τ and a result position rp for a given function f by

$$v \rightarrowtail^{f}_{rp} v'$$

if there exists $p \in \mathit{Pos}\, v$ such that $v \lessdot_p v'$ and p is a *sequential* position in v at result position rp with respect to f. We also refer to this as a sequential step. Furthermore we relate values v and v' of type τ and a result position rp for a given function f by

$$v \rightsquigarrow^{f}_{rp} v'$$

if there exists $p \in \mathit{Pos}\, v$ such that $v \lessdot_p v'$ and p is a *non-sequential* position in v at result position rp with respect to f. We also refer to this as a non-sequential step.

Whenever f is determined by the context, we omit it. Furthermore, although f is supposed to be a semantic function for simplicity we annotate the refinement relations with the corresponding syntactic function. □

In the following, if we have $v \lessdot_p v'$ but disregard the refinement position, we omit the position and write $v \lessdot v'$. In the more general setting of an arbitrary partially ordered set $(P, \sqsubseteq)$ for elements $x \in P$ and $y \in P$ the relation $\lessdot$ is called covering relation and $x \lessdot y$ is defined as $x \sqsubset y$ and there exists no $z \in P$ with $x \sqsubset z \sqsubset y$.

Let us consider some values that are related by one of the refinement relations.

Example 4.3.1: Consider *andL* from Example 2.2.1 again. We have

$$\langle \bot, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle False, \bot \rangle$$

as well as

$$\langle \bot, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, \bot \rangle$$

because position 1 is a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ with respect to *andL*. Furthermore, we have

$$\langle True, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, False \rangle$$

as well as

$$\langle True, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, True \rangle$$

because position 2 is a sequential position in $\langle True, \bot \rangle$ at position ϵ with respect to *andL*. On the other hand we have

$$\langle \bot, \bot \rangle \rightsquigarrow_{\epsilon}^{andL} \langle \bot, False \rangle$$

as well as

$$\langle \bot, \bot \rangle \rightsquigarrow_{\epsilon}^{andL} \langle \bot, True \rangle$$

because position 2 is a non-sequential position in $\langle \bot, \bot \rangle$ at result position ϵ with respect to *andL*. Note that we neither have

$$\langle False, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle False, False \rangle$$

nor

$$\langle False, \bot \rangle \rightsquigarrow_{\epsilon}^{andL} \langle False, False \rangle$$

because $(\llbracket andL \rrbracket \langle False, \bot \rangle)|_{\epsilon} = False \neq \bot$, and, therefore, position 2 is neither a sequential nor a non-sequential position in $\langle False, \bot \rangle$ at result position ϵ with respect to *andL*.

If we consider the function *and* from Example 4.1.2, we have

$$t_1 \rightarrowtail_{\epsilon}^{and} t_2$$

for all $t_1, t_2 \in \llbracket Bool \times Bool \rrbracket$ with $t_1 \lessdot t_2$ because $\llbracket and \rrbracket$ yields $\bot$ as long as any of its arguments is still undefined. In other words, all refinement steps with respect to $\llbracket and \rrbracket$ are sequential steps. □

On basis of the definition of refinement steps we define the characteristic set of a function. The elements of the characteristic set are test cases used to check whether a function is minimally strict. The characteristic set of a function $f \in \llbracket \sigma \to \tau \rrbracket$ is a subset of the set of all pairs of argument values and valid result positions such that f applied to the argument yields $\bot$ at the result position. That is, the characteristic set

is a subset of the set $\{(v, rp) \mid v \in [\![\sigma]\!], rp \in Pos\,(f\,v), (f\,v)|_{rp} = \bot\}$. We omit pairs from the characteristic set for which the behavior of f is determined by sequentiality.

Definition 4.3.3 (Characteristic Set): The characteristic set C_f of a sequential function $f \in [\![\sigma \to \tau]\!]$ is inductively defined as follows.

1. For all result positions rp such that $(f\,\bot)|_{rp} = \bot$ we have $(\bot, rp) \in C_f$.
2. For all pairs $(v, rp) \in C_f$ and all values v' such that $v \rightarrowtail^f_{rp} v'$ and all result positions rp' such that $(f\,v')|_{rp'} = \bot$ we have $(v', rp') \in C_f$. □

Obviously, we have $C_f \subseteq \{(v, rp) \mid v \in [\![\sigma]\!], rp \in Pos\,(f\,v), (f\,v)|_{rp} = \bot\}$. Furthermore, if we have $(v, rp) \in C_f$, then there exists a sequence of sequential steps $\bot \rightarrowtail^* v$ that is monotonic with respect to the considered result positions. Thus, if we have

$$\bot \rightarrowtail^* v_1 \rightarrowtail_{rp} v_2 \rightarrowtail^* v_3 \rightarrowtail_{rp'} v_4 \rightarrowtail^* v,$$

then rp is a prefix of rp', denoted by $rp \leq rp'$.

Example 4.3.2: Let us consider the Boolean conjunctions *andL* from Example 2.2.1 as well as *and* from Example 4.1.2, again. What do the characteristic sets for $[\![andL]\!]$ and $[\![and]\!]$ look like?

In the following, for simplicity, we say that "a value v is not in C_f" or "v is not an element of C_f" if we have $(v, rp) \notin C_f$ for *all* result positions rp.

First, we consider the function $[\![andL]\!]$. We have $([\![andL]\!]\,\langle\bot, \bot\rangle)|_\epsilon = \bot$, and, by definition of the characteristic set, we have $(\langle\bot, \bot\rangle, \epsilon) \in C_{[\![andL]\!]}$. Note that $\langle\bot, \bot\rangle$ is the least element of $[\![Bool \times Bool]\!]$, which is denoted by $\bot$ in Definition 4.3.3. Position ϵ is the only valid result position of $[\![andL]\!]\,\langle\bot, \bot\rangle$, which yields $\bot$, when evaluated. Thus, the value $\langle\bot, \bot\rangle$ is not in $C_{[\![andL]\!]}$.

As we have observed in Example 4.3.1, we have

$$\langle\bot, \bot\rangle \rightarrowtail^{andL}_\epsilon \langle False, \bot\rangle$$

as well as

$$\langle\bot, \bot\rangle \rightarrowtail^{andL}_\epsilon \langle True, \bot\rangle.$$

Nevertheless, as $[\![andL]\!]\,\langle False, \bot\rangle$ yields a total result, the value $\langle False, \bot\rangle$ is not in $C_{[\![andL]\!]}$ because there exists no result position rp such that $([\![andL]\!]\,\langle False, \bot\rangle)|_{rp} = \bot$. That is, some values v are not in the characteristic set because the considered function yields a total result if it is applied to v. In contrast, the pair $(\langle True, \bot\rangle, \epsilon)$ is in the characteristic set as we have $([\![andL]\!]\,\langle True, \bot\rangle)|_\epsilon = \bot$.

In Example 4.3.1 we have observed that

$$\langle\bot, \bot\rangle \rightsquigarrow^{andL}_\epsilon \langle\bot, False\rangle$$

as well as

$$\langle\bot, \bot\rangle \rightsquigarrow^{andL}_\epsilon \langle\bot, True\rangle$$

holds and, thus, $\langle\bot, False\rangle$ and $\langle\bot, True\rangle$ are not in the characteristic set. These values are not in the characteristic set of $[\![andL]\!]$ because the behavior for these arguments

is determined by sequentiality. More precisely, as position 1 is a sequential position in $\langle \bot, \bot \rangle$ we have $[\![andL]\!]\ \langle \bot, False \rangle = \bot$ and $[\![andL]\!]\ \langle \bot, True \rangle = \bot$ by sequentiality.

The value $\langle False, False \rangle$ is not in the characteristic set because neither $\langle False, \bot \rangle$ nor $\langle \bot, False \rangle$ are in the characteristic set. We do not have to consider $\langle False, False \rangle$ because it is more defined than $\langle False, \bot \rangle$ and *andL* already yields a total result for $\langle False, \bot \rangle$. Therefore, all values more defined than $\langle False, \bot \rangle$ are not in the characteristic set.

As we have observed in Example 4.3.1, we have the sequential steps

$$\langle True, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, False \rangle$$

and

$$\langle True, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, True \rangle.$$

However, as the results of $[\![andL]\!]$ for $\langle True, False \rangle$ as well as for $\langle True, True \rangle$ are total we have neither $\langle True, False \rangle \in C_{[\![andL]\!]}$ nor $\langle True, True \rangle \in C_{[\![andL]\!]}$.

In summary, we get the following characteristic set for $[\![andL]\!]$.

$$C_{[\![andL]\!]} = \{(\langle \bot, \bot \rangle, \epsilon), (\langle True, \bot \rangle, \epsilon)\}$$

The value $\langle False, \bot \rangle$ is missing in $C_{[\![andL]\!]}$ because *andL* yields a total value, namely, *False*, for this argument. The arguments $\langle \bot, False \rangle$ and $\langle \bot, True \rangle$ are missing in $C_{[\![andL]\!]}$ because there is no sequence of sequential steps from $(\langle \bot, \bot \rangle, \epsilon)$ to these values.

In the case of $[\![and]\!]$ all steps are sequential with respect to the result position ϵ.

$$C_{[\![and]\!]} = \left\{ \begin{array}{c} (\langle \bot, \bot \rangle, \epsilon), (\langle False, \bot \rangle, \epsilon), (\langle True, \bot \rangle, \epsilon), \\ (\langle \bot, False \rangle, \epsilon), (\langle \bot, True \rangle, \epsilon). \end{array} \right\}.$$

Therefore, the characteristic set consists of all pairs (v, ϵ) where v is a value of type $Bool \times Bool$ for which $[\![and]\!]$ yields a non-total result. □

By means of the characteristic set we define a criterion to check whether a function is minimally strict. The criterion states that a sequential function $f \in [\![\sigma \to \tau]\!]$ is not minimally strict if and only if there exists $(v, rp) \in C_f$ with $(\inf_f\ v)|_{rp} \sqsupset \bot$. In the remainder of this chapter we prove the following theorem, where $\inf_f\ v$ is defined by $\inf_f\ v = \bigsqcap\{f\ tv \mid tv \in [\![\sigma]\!]\uparrow, v \sqsubseteq tv\}$ (see Definition 4.1.3).

Theorem 4.3.1: *A sequential function $f \in [\![\sigma \to \tau]\!]$ is minimally strict if and only if we have $(\inf_f\ v)|_{rp} = \bot$ for all $(v, rp) \in C_f$.*

In the following two sections (Section 4.3.1 and Section 4.3.2) we assume that all considered functions are sequential.

4.3.1 Sufficiency of the Criterion

In this section we show that $(\inf_f\ v)|_{rp} = \bot$ for all $(v, rp) \in C_f$ is a sufficient condition for f being minimally strict. In other words, we show that $(\inf_f\ v)|_{rp} = \bot$ for all $(v, rp) \in C_f$ implies that f is minimally strict.

To prove this statement we show that the characteristic set of a function is in a certain sense characteristic. We consider two functions f and g such that $f \preceq g$. If we have $(f\ v)|_{rp} = \bot$ for all values $v \in [\![\sigma]\!]$ and valid result positions rp such that $(g\ v)|_{rp} = \bot$, then we have $f = g$. In the following we show that we do not have to check all values v and results positions rp such that $(g\ v)|_{rp} = \bot$. We already have $f = g$ if $(f\ v)|_{rp} = \bot$ for all $(v, rp) \in C_g$.

The characteristic set contains only finite values. Therefore, $(f\ v)|_{rp} = \bot$ for all $(v, rp) \in C_g$ merely implies that f and g agree for all finite values v. Nevertheless, as we use algebraic cpos the behavior of continuous functions for non-finite values is determined by their behavior for finite values. Consider two functions $f, g \in [\![\sigma \to \tau]\!]$ and a non-finite value $v \in [\![\sigma]\!]$. If f and g agree for all finite values, they also agree for all non-finite values as the following equational reasoning shows.

$$
\begin{aligned}
f\ v &= f \bigsqcup \{v' \mid v' \in [\![\sigma]\!], v' \sqsubseteq v, v' \text{ finite}\} && \text{algebraicity} \\
&= \bigsqcup \{f\ v' \mid v' \in [\![\sigma]\!], v' \sqsubseteq v, v' \text{ finite}\} && \text{continuity of } f \\
&= \bigsqcup \{g\ v' \mid v' \in [\![\sigma]\!], v' \sqsubseteq v, v' \text{ finite}\} && f \text{ and } g \text{ agree for finite values} \\
&= g \bigsqcup \{v' \mid v' \in [\![\sigma]\!], v' \sqsubseteq v, v' \text{ finite}\} && \text{continuity of } g \\
&= g\ v && \text{algebraicity}
\end{aligned}
$$

As a first step towards the sufficiency of the criterion, we show that for all pairs (v, rp) with $(f\ v)|_{rp} = \bot$ that are not in the characteristic set there exists a pair (v', rp) that is in the characteristic set. Furthermore, there exists a sequence of non-sequential steps at position rp from v' to v, that is, $v'(\rightsquigarrow_{rp})^* v$. Figure 4.3.1 illustrates the statement. The white rectangle represents the set of all pairs (v, rp) such that $(f\ v)|_{rp} = \bot$ while the gray rectangle represents the characteristic set C_f. For every pair (v, rp) that is not in the characteristic set there exists a pair (v', rp) in the characteristic set and a non-sequential path from v' to v. Considering this sequence, we then employ the following lemma to show that we have $(f\ v)|_{rp} = \bot$. In the following we assume that $\cdot[v]_p$ has a higher precedence than function application, that is, $f\ v[v']_p$ stands for $f\ (v[v']_p)$.

Lemma 4.3.1: *Let p be a non-sequential position of f in v at result position rp. Then for all values v' of appropriate type we have $(f\ v[v']_p)|_{rp} = \bot$.*

Note that the previous lemma implies that we have $(f\ v_2)|_{rp} = \bot$ if $v_1 \rightsquigarrow_{rp} v_2$ because we have $v_2 = v_1[C\ \bot]_p$ for some constructor C and a non-sequential position p. Thus, if there exists a non-sequential sequence $v'(\rightsquigarrow_{rp})^* v$ we have $(f\ v)|_{rp} = \bot$.

To prove that there always exists a non-sequential sequence from a pair that is in the characteristic set to one that is not, we consider an arbitrary pair $(v, rp) \notin C_f$ with $(f\ v)|_{rp} = \bot$. The following lemma shows that $v \neq \bot$ implies that there exists a predecessor v' in the sense that we have $v' \rightsquigarrow_{rp} v$ or we have $v' \rightarrowtail_{rp'} v$ and $rp' \leq rp$.

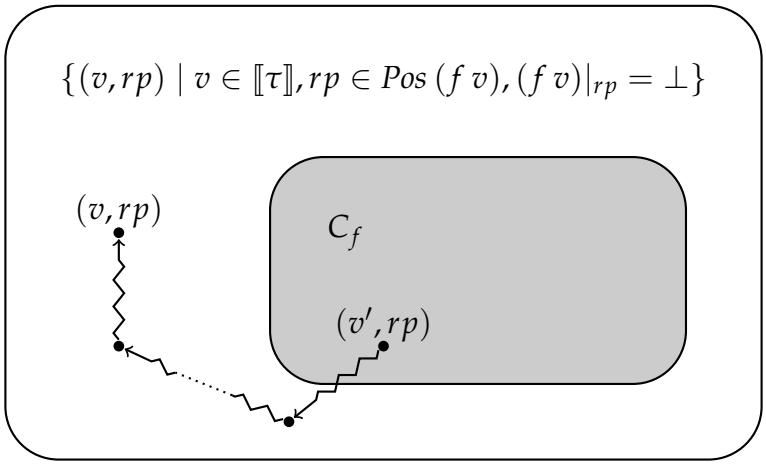


Figure 4.3.1: Illustration of Lemma 4.3.5

Lemma 4.3.2: *For all values $v \in [\![\tau]\!] \setminus \{\bot\}$ and $rp \in Pos\,(f\ v)$ with $(f\ v)|_{rp} = \bot$ there exists $v' \in [\![\tau]\!]$ and $rp' \in Pos\,(f\ v')$ such that $v' \rightsquigarrow_{rp} v$ or such that $v' \rightarrowtail_{rp'} v$ and $rp' \leq rp$.*

Let us get back to the proof of the statement illustrated by Figure 4.3.1. We only consider finite values v and prove the statement by induction over the number of constructors in v. As mentioned before, we consider a pair $(v, rp) \notin C_f$ with $(f\ v)|_{rp} = \bot$. By the previous lemma there exists a value v' such that we have either $v' \rightsquigarrow_{rp} v$ or we have $v' \rightarrowtail_{rp'} v$ and $rp' \leq rp$. First we look at the former case. If we have $(v', rp) \in C_f$, we are finished as we have a non-sequential sequence from v' to v. Therefore, we consider the case that $(v', rp) \notin C_f$. By using the induction hypothesis we get a non-sequential sequence that arrives at v'. In other words, there exists $(v'', rp) \in C_f$ such that $v''(\rightsquigarrow_{rp})^* v'$. Thus, we use $v' \rightsquigarrow_{rp} v$ to construct a new non-sequential sequence that arrives at v.

Let us look at the remaining case, namely, $v' \rightarrowtail_{rp'} v$ with $rp' \leq rp$. As we have $(v, rp) \notin C_f$, we get $(v', rp') \notin C_f$ by definition of the characteristic set. By induction hypothesis we conclude that there exists $(v'', rp') \in C_f$ such that $v''(\rightsquigarrow_{rp'})^* v' \rightarrowtail_{rp'} v$. As we are looking for a purely non-sequential sequence, in the following, we show that we can rearrange this sequence such that all sequential steps are "performed" before any non-sequential steps. Before we prove the statement for arbitrary sequences, we consider sequences of length two, that is, we have a sequence of the form $v_1 \rightsquigarrow_{rp} v_2 \rightarrowtail_{rp} v_3$.

Example 4.3.3: Consider *andL* from Example 2.2.1 again and the following sequence of a non-sequential and a sequential step.

$$\langle \bot, \bot \rangle \rightsquigarrow_{\epsilon}^{andL} \langle \bot, True \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, True \rangle.$$

If we interchange the steps, we get the following sequence of two sequential steps.

$$\langle \bot, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle True, True \rangle$$

Now consider the following sequence.

$$\langle \bot, \bot \rangle \rightsquigarrow_{\epsilon}^{andL} \langle \bot, False \rangle \rightarrowtail_{\epsilon}^{andL} \langle False, False \rangle.$$

If we first perform the sequential step, the step remains sequential.

$$\langle \bot, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle False, \bot \rangle$$

However, we cannot perform a second step as we have $(\llbracket andL \rrbracket \langle False, \bot \rangle)|_{\epsilon} = False$. Thus, there exists neither a sequential nor a non-sequential position in $\langle False, \bot \rangle$. In other words, we have neither

$$\langle False, \bot \rangle \rightarrowtail_{\epsilon}^{andL} \langle False, False \rangle$$

nor

$$\langle False, \bot \rangle \rightsquigarrow_{\epsilon}^{andL} \langle False, False \rangle.$$

We have observed that a non-sequential step might become sequential or might vanish if we interchange it with a sequential step. Finally, we show that the non-sequential step might even remain non-sequential. We define the following generalization of *andL* to three Boolean arguments, called *andL3*.

$$\begin{array}{l} andL3 :: Bool \times Bool \times Bool \to Bool \\ andL3\langle x, y, z \rangle = \\ \quad \textbf{case } x \textbf{ of} \\ \qquad False \to False \\ \qquad True \ \to \textbf{case } y \textbf{ of} \\ \qquad\qquad\qquad False \to False \\ \qquad\qquad\qquad True \ \to z \end{array}$$

Employing the semantics of Figure 2.2.4 we get the following semantics for *andL3*.

$$\llbracket andL3 \rrbracket = \lambda\langle v_1, v_2, v_3 \rangle. \begin{cases} \bot & \text{if } v_1 = \bot \\ False & \text{if } v_1 = False \\ \bot & \text{if } v_1 = True \wedge v_2 = \bot \\ False & \text{if } v_1 = True \wedge v_2 = False \\ v_3 & \text{otherwise} \end{cases}$$

Now consider the following sequence of a non-sequential and then a sequential step.

$$\langle \bot, \bot, \bot \rangle \rightsquigarrow_{\epsilon}^{andL3} \langle \bot, \bot, True \rangle \rightarrowtail_{\epsilon}^{andL3} \langle True, \bot, True \rangle$$

If we interchange these steps, we get a sequential and then a non-sequential step.

$$\langle \bot, \bot, \bot \rangle \rightarrowtail_{\epsilon}^{andL3} \langle True, \bot, \bot \rangle \rightsquigarrow_{\epsilon}^{andL3} \langle True, \bot, True \rangle$$

To show that the second step is non-sequential we have to provide a value v with $v \sqsupseteq \langle \mathit{True}, \bot, \bot \rangle$ and $v|_3 = \bot$ such that $(\llbracket \mathit{andL3} \rrbracket\, v)|_\epsilon \neq \bot$. The value $\langle \mathit{True}, \mathit{False}, \bot \rangle$ satisfies these conditions because we have $(\llbracket \mathit{andL3} \rrbracket\, \langle \mathit{True}, \mathit{False}, \bot \rangle)|_\epsilon = \mathit{False}$. □

As a next step, we prove that, if there exists a sequence of a non-sequential and a final sequential step, then there exists a sequence of a sequential and an arbitrary step that arrives at the same value as the original sequence. Let us consider a sequence of a non-sequential and a sequential step, namely, a sequence of the following form.

$$v_1 \rightsquigarrow_{rp_1} v_2 \rightarrowtail_{rp_2} v_3$$

In fact, we do not consider arbitrary sequences of this form, but only sequences that are monotonic with respect to result positions, that is, we have $rp_1 \leq rp_2$. In this case by Lemma 4.3.1 we have $(f\, v_2)|_{rp_1} = \bot$ and, therefore, $rp_1 = rp_2$. Thus, we can even only consider a particular form of these sequences, namely, the following.

$$v_1 \rightsquigarrow_{rp_1} v_2 \rightarrowtail_{rp_1} v_3$$

As Example 4.3.3 shows, we do not always arrive at the result value of the original sequence if we consider sequences of this form. That is, if we first perform the sequential step we might not be able to arrive at the final value by neither a sequential nor a non-sequential step. Therefore, we only consider sequences such that $(f\, v_3)|_{rp_3} = \bot$ for some result position rp_3 with $rp_1 \leq rp_3$. In this case we have $(f\, v)|_{rp_2} = \bot$ for all values v with $v_1 \sqsubseteq v \sqsubseteq v_3$ and some result position rp_2 with $rp_1 \leq rp_2 \leq rp_3$. This follows from the definition of the relation $\sqsubseteq$ of the considered domains as well as monotonicity of f. As we have $(f\, v)|_{rp_2} = \bot$, there either exists a sequential or a non-sequential step that arrives at v_3.

Now that we know that we always arrive at the final value, let us consider a sequence of the following form.

$$v_1 \rightsquigarrow_{rp_1} v_2 \rightarrowtail_{rp_1} v_3$$

If we first perform the sequential step of this sequence, we have to distinguish two cases. In the first case we get a sequential step from v_1 to some value v and a non-sequential step from v to v_2. As the latter step is non-sequential and by precondition we have $(f\, v_2)|_{rp_3} = \bot$, we get $(f\, v)|_{rp_3} = \bot$ because non-sequential steps do not produce more defined function results (Lemma 4.3.1). In the other case the resulting steps are both sequential. In this case the second step might be at a result position rp_2 that is between rp_1 and rp_3. In other words, we have $(f\, v_2)|_{rp_2} = \bot$ and $rp_1 \leq rp_2 \leq rp_3$. The following lemma summarizes these observations.

Lemma 4.3.3: *For all $v_1, v_2, v_3 \in \llbracket \tau \rrbracket$ if*

$$v_1 \rightsquigarrow_{rp_1} v_2 \rightarrowtail_{rp_1} v_3$$

and $(f\, v_3)|_{rp_3} = \bot$ with $rp_1 \leq rp_3$, then there exists $v \in \llbracket \tau \rrbracket$ such that either

$$v_1 \rightarrowtail_{rp_1} v \rightsquigarrow_{rp_3} v_3 \qquad \textit{or} \qquad v_1 \rightarrowtail_{rp_1} v \rightarrowtail_{rp_2} v_3$$

for some rp_2 *with* $rp_1 \leq rp_2 \leq rp_3$.

We generalize the previous lemma to sequences of arbitrary length that are terminated by a sequential step. By iterated application of the previous lemma we can move this sequential step to the front of the sequence. Moving this step to the front might result in new non-sequential steps that can be moved to the front again. Finally, we get a sequence that first performs only sequential steps before it then performs only sequential steps. Furthermore, the resulting sequence is monotonic with respect to result positions. This monotonicity allows us to conclude that the pair (v', rp') is in the characteristic set if the pair (v_1, rp) is in the set as well.

Lemma 4.3.4: *For all* $v_1, v_2, v_3 \in [\![\tau]\!]$ *if*

$$v_1 (\rightsquigarrow_{rp})^* v_2 \rightarrowtail_{rp} v_3$$

and $(f\ v_3)|_{rp'} = \bot$ *with* $rp \leq rp'$, *then there exist* $v, v' \in [\![\tau]\!]$ *such that*

$$v_1 \rightarrowtail_{rp} v \rightarrowtail^* v' (\rightsquigarrow_{rp'})^* v_3.$$

Furthermore, this sequence is montone with respect to result positions.

We use the previous lemma to show that for every pair of argument value and result position that is not in the characteristic set of a function there exists a non-sequential sequence that arrives at this pair and starts from a pair in the characteristic set.

Lemma 4.3.5: *For all finite values* $v \in [\![\tau]\!]$ *with* $(f\ v)|_{rp} = \bot$ *the following holds.*

$$(v, rp) \notin C_f \Longrightarrow \exists (v', rp) \in C_f : v' (\rightsquigarrow_{rp})^* v$$

The main theorem of this section shows that the characteristic set is indeed in a certain sense characteristic for a function. Functions f and g with $f \sqsupseteq g$ are equal if and only if $(f\ v)|_{rp} = \bot$ for all pairs (v, rp) of the characteristic set of g. Before we prove this main theorem, we prove the following lemma, which shows a connection between non-sequential refinement steps of functions that are related by $\sqsupseteq$. Note that we consider functions f and g with $f \sqsupseteq g$ and not $f \preceq g$. We do not need the additional constraint that the functions agree for total arguments to proof the following statements. Nevertheless, obviously these statements also hold for functions f and g with $f \preceq g$.

Lemma 4.3.6: *Let* $f, g \in [\![\sigma \to \tau]\!]$ *such that* $f \sqsupseteq g$. *For all* $v, v' \in [\![\sigma]\!]$ *if* $v \rightsquigarrow^g_{rp} v'$ *and* $(f\ v)|_{rp} = \bot$, *then* $v \rightsquigarrow^f_{rp} v'$.

Proof: Let $v, v' \in [\![\tau]\!]$ such that $v \rightsquigarrow^g_{rp} v'$. We have $(f\ v)|_{rp} = \bot$ and $(g\ v)|_{rp} = \bot$. Let p be a non-sequential position in v at result position rp with respect to g. By definition of non-sequential steps there exists a position p such that $v \lessdot_p v'$. Furthermore, as p is a non-sequential position in v at result position rp with respect to g there exists a value $v'' \in [\![\tau]\!]$ such that $v'' \sqsupset v$ and $v''|_p = \bot$ and $(g\ v'')|_p \neq \bot$. Because

$f \preceq g$, we have $f\,v'' \sqsupseteq g\,v''$. Therefore, we have $(f\,v'')|_p \neq \bot$ by monotonicity of $\cdot|_p$. That is, p is also a non-sequential position in v at result position rp with respect to f. □

The following theorem finally shows that functions f and g with $f \sqsupseteq g$ are equal if we have $(f\,v)|_{rp} = \bot$ for all $(v, rp) \in C_g$.

Theorem 4.3.2: *Let $f, g \in [\![\tau \to \tau']\!]$ such that $f \sqsupseteq g$. The following holds.*

$$f = g \iff \forall (v, rp) \in C_g.\ (f\,v)|_{rp} = \bot$$

Proof: We prove both implications separately.

$\Longrightarrow$: Because we have $(g\,v)|_{rp} = \bot$ for all $(v, rp) \in C_g$ and $f = g$, we get $(f\,v)|_{rp} = \bot$ for all $(v, rp) \in C_g$.

$\Longleftarrow$: As we have observed in the introduction of this section, in the setting we are considering, if two functions agree for all finite values, they also agree for non-finite values. Therefore, we only have to consider finite values in the following.

Because $f \sqsupseteq g$, we have $f\,v \sqsupseteq g\,v$ for all $v' \in [\![\tau]\!]$. We assume that there exists a finite value $v' \in [\![\tau]\!]$ such that $f\,v' \sqsupset g\,v'$. Then, by definition of $\sqsupseteq$ there exists a result position rp such that $(f\,v')|_{rp} \neq \bot$ but $(g\,v')|_{rp} = \bot$. Thus, we only have to show that $(g\,v)|_{rp} = \bot$ implies $(f\,v)|_{rp} = \bot$ for all finite values v.

Let $v \in [\![\tau]\!]$ be a finite value and $rp \in Pos\,(g\,v)$ such that $(g\,v)|_{rp} = \bot$. We have $rp \in Pos\,(f\,v)$ because $f \sqsupseteq g$. If $(v, rp) \in C_g$, then by precondition we have $(f\,v)|_{rp} = \bot$. If $(v, rp) \notin C_g$, by Lemma 4.3.5 there exists $(v', rp) \in C_g$ such that $v'(\rightsquigarrow^g_{rp})^* v$. As $(v', rp) \in C_g$ and $(g\,v')|_{rp} = \bot$, we have $(f\,v')|_{rp} = \bot$ as well. By multiple applications of Lemma 4.3.6 and Lemma 4.3.1 we get $v'(\rightsquigarrow^f_{rp})^* v$ and, thus, $(f\,v)|_{rp} = \bot$. □

A reformulation of one implication of the previous theorem reveals a connection between Theorem 4.3.2 and the sufficiency of the criterion to check whether a function is minimally strict. More precisely, we observe that a less strict function is always more defined with respect to a pair from the characteristic set of the unnecessarily strict function.

Corollary 4.3.1: *Let $f, g \in [\![\sigma \to \tau]\!]$. Then the following holds.*

$$f \sqsupset g \Longrightarrow \exists (v, rp) \in C_g.\ (f\,v)|_{rp} \sqsupset \bot$$

Proof: Let $f, g \in [\![\sigma \to \tau]\!]$ such that $f \sqsupset g$, that is, $f \sqsupseteq g$ and $f \neq g$. By Theorem 4.3.2 there exists $(v, rp) \in C_g$ such that $(f\,v)|_{rp} \neq \bot$. Because we have $f \sqsupset g$, we get $(f\,v)|_{rp} \sqsupset \bot$. □

Finally, we show the sufficiency of the criterion, which gives us one half of the proof of Theorem 4.3.1.

Lemma 4.3.7: *Let $g \in [\![\sigma \to \tau]\!]$ be sequential. If there exists a sequential function $f \in [\![\sigma \to \tau]\!]$ with $f \sqsupset g$, then there exists $(v, rp) \in C_g$ with $(\inf_g\ v)|_{rp} \sqsupset \bot$.*

Proof: If there exists a sequential function f with $f \sqsupset g$, by Corollary 4.3.1 there exists $(v, rp) \in C_g$ such that $(f\ v)|_{rp} \sqsupset \bot$. Furthermore, for all total values tv of type τ we have $f\ tv = g\ tv$, and, therefore, for all values v of type τ we have $\inf_f\ v = \inf_g\ v$. As we have already observed (just before Definition 4.1.3), we have $\inf_f\ v \sqsupseteq f\ v$. Finally, we get $(\inf_g\ v)|_{rp} = (\inf_f\ v)|_{rp} \sqsupseteq (f\ v)|_{rp} \sqsupset \bot$. □

The previous lemma also holds if we consider functions f and g such that $f \prec g$.

4.3.2 Necessity of the Criterion

In this section we show that $(\inf_f\ v)|_{rp} = \bot$ for all $(v, rp) \in C_f$ is a necessary condition for f being minimally strict. That is, we show that, if f is minimally strict, then we have $(\inf_f\ v)|_{rp} = \bot$ for all $(v, rp) \in C_f$. Actually, we prove the contraposition of this statement: if there exists $(v, rp) \in C_f$ with $(\inf_f\ v)|_{rp} \neq \bot$, then f is not minimally strict. To prove this statement, for a function $f \in [\![\sigma \to \tau]\!]$ with $(v, rp) \in C_f$ and $(\inf_f\ v)|_{rp} \neq \bot$, we define a function $\overline{f}_{(v,rp)}$ that is less strict than f. Note that the less strict function is indexed by the counter-example. That is, for every counter-example we can define a function that is less strict than f. Here and in the following, by *counter-example* we refer to a pair $(v, rp) \in C_f$ with $(\inf_f\ v)|_{rp} \neq \bot$. Furthermore, note that $(\inf_f\ v)|_{rp} \neq \bot$ implies $(\inf_f\ v)|_{rp} \sqsupset \bot$ because $\inf_f\ v \sqsupseteq f\ v$ and $(f\ v)|_{rp} = \bot$.

If there exists $(v, rp) \in C_f$ with $(\inf_f\ v)|_{rp} \sqsupset \bot$, we are looking for a function $\overline{f}_{(v,rp)}$ that agrees with $\inf_f\ v$ at result position rp. We are looking for a function $\overline{f}_{(v,rp)}$ with

$$(\overline{f}_{(v,rp)}\ v)|_{rp} = (\inf_f\ v)|_{rp}.$$

As a first approach for all other arguments v' and result positions rp', the function $\overline{f}_{(v,rp)}$ may agree with f, that is,

$$(\overline{f}_{(v,rp)}\ v')|_{rp'} = (f\ v')|_{rp'}.$$

However, the following example demonstrates that we cannot use this simple definition as the resulting function is not continuous.

Example 4.3.4: Consider a function that takes a list and yields a singleton list.

```
singleton :: [Bool] → [Bool]
singleton xs =
  case xs of
    Nil_Bool       → Cons⟨False, Nil_Bool⟩
    Cons⟨y, ys⟩ → case y of
                    False → Cons⟨y, Nil_Bool⟩
                    True  → Cons⟨y, Nil_Bool⟩
```

This function is unnecessarily strict for two reasons. First, it checks whether the argument is the empty list or not although it yields a list with one element in all cases. Furthermore, it performs pattern matching on the first element of the argument list although it yields a singleton list in both cases. We get the following semantics for *singleton*.

$$[\![singleton]\!] = \lambda v. \begin{cases} \bot & \text{if } v = \bot \\ Cons\ \langle False, Nil_{Bool} \rangle & \text{if } v = Nil_{Bool} \\ \bot & \text{if } v = Cons\ \langle \bot, v_2 \rangle \\ Cons\ \langle v_1, Nil_{Bool} \rangle & \text{if } v = Cons\ \langle v_1, v_2 \rangle \wedge v_1 \neq \bot \end{cases}$$

As we have $(\bot, \epsilon) \in C_{[\![singleton]\!]}$ as well as

$$(\inf_{[\![singleton]\!]}\ \bot)|_\epsilon = Cons\ \langle \bot, Nil_{Bool} \rangle \sqsupset \bot = [\![singleton]\!]\ \bot,$$

Theorem 4.3.1 would state that *singleton* is not minimally strict. Therefore, we are looking for a less strict function and consider the following definition of $\overline{[\![singleton]\!]}$.

$$\overline{[\![singleton]\!]}\ v = \begin{cases} Cons\ \langle \bot, Nil_{Bool} \rangle & \text{if } v = \bot \\ [\![singleton]\!]\ v & \text{otherwise} \end{cases}$$

Obviously, $\overline{[\![singleton]\!]}$ is less strict than $[\![singleton]\!]$. However, this function is not monotonic as we have $\bot \sqsubseteq Cons\ \langle \bot, \bot \rangle$, but

$$\overline{[\![singleton]\!]}\ \bot = Cons\ \langle \bot, Nil_{Bool} \rangle$$

and

$$\overline{[\![singleton]\!]}\ (Cons\ \langle \bot, \bot \rangle) = \bot.$$

This example demonstrates that we may not change the definition of a function for a single argument, in this case $\bot$, as the resulting function is non-continuous. Thus, we might consider the following definition, which does not change the behavior for a single argument but for all arguments that are more defined than $\bot$.

$$\overline{[\![singleton]\!]}\ v = \begin{cases} Cons\ \langle \bot, Nil_{Bool} \rangle & \text{if } v \sqsupseteq \bot \\ [\![singleton]\!]\ v & \text{otherwise} \end{cases}$$

However, in contrast to the previous definition, this time $\overline{[\![singleton]\!]}$ is not less strict than $[\![singleton]\!]$ as the following example demonstrates. We have

$$\overline{[\![singleton]\!]}\ (Cons\ \langle False, \bot \rangle) = Cons\ \langle \bot, Nil_{Bool} \rangle$$

and

$$[\![singleton]\!]\ (Cons\ \langle False, \bot \rangle) = Cons\ \langle False, Nil_{Bool} \rangle.$$

This example shows that we have to take over the definition of $[\![singleton]\!]$ if it yields a more defined result than $\overline{[\![singleton]\!]}$.

In order to define a function that shows this behavior we use the least upper bound of the original result and the less strict result. Here and in the following, we assume that function application by juxtaposition binds stronger than the binary supremum operator $\sqcup$.

$$\overline{[\![singleton]\!]}\ v = \begin{cases} [\![singleton]\!]\ v \sqcup Cons\ \langle \bot, Nil_{Bool} \rangle & \text{if } v \sqsupseteq \bot \\ [\![singleton]\!]\ v & \text{otherwise} \end{cases}$$

In this case the resulting function is continuous and less strict than the original function. In the general case of an arbitrary pair $(v, rp) \in C_f$ with $\inf_f\ v \sqsupset \bot$ we have to spend some extra care. We have to guarantee that $\overline{f}$ is only more defined at the considered result position rp and not at other result positions. Otherwise the function might become non-sequential. In the example at hand this task was easy because we considered the result position ϵ. □

The following definition presents a function $\overline{f}$ that is continuous, sequential, and less strict than f. In contrast to the previous example we use a substitution to only refine the function with respect to result position rp. More precisely, we replace the subterm of the result of f that is undefined but should be more defined by the subterm of the supremum at the corresponding position.

Definition 4.3.4 (Less Strict Function): Let $f \in [\![\sigma \to \tau]\!]$ and $(v, rp) \in C_f$ such that $(\inf_f\ v)|_{rp} \sqsupset \bot$. For all $v' \in [\![\tau]\!]$ we define a function $\overline{f}_{(v,rp)} : [\![\sigma]\!] \to [\![\tau]\!]$ as follows.

$$\overline{f}_{(v,rp)}\ v' := \begin{cases} (f\ v')[(f\ v' \sqcup \inf_f\ v)|_{rp}]_{rp} & \text{if } v' \sqsupseteq v \\ f\ v' & \text{otherwise} \end{cases}$$

In the following we mostly use $\overline{f}$ instead of $\overline{f}_{(v,rp)}$. □

By employing the statements of Lemma 4.2.1 and Lemma 4.2.2 we can prove that the first line is, in fact, equivalent to $f\ v' \sqcup (f\ v')[(\inf_f\ v)|_{rp}]_{rp}$.

To show that $\overline{f}$ is well-defined we have to show that the supremum $f\ v' \sqcup \inf_f\ v$ always exists. Right after Definition 4.1.3 we have shown that for every value v there exists a total value that is at least as defined as v. Therefore, for every value $v' \in [\![\sigma]\!]$ there exists a total value $tv \in [\![\sigma]\!]\uparrow$ with $tv \sqsupseteq v'$. Furthermore, we have $tv \sqsupseteq v$ because $v' \sqsupseteq v$. By monotonicity we have $f\ tv \sqsupseteq f\ v'$. By definition of $\inf_f$ we get $f\ tv \sqsupseteq \inf_f\ v$ because $\inf_f$ is an infimum of a set that, among other applications, contains the application $f\ tv$. In a directed cpo every directed set has a least upper bound. As $\{f\ v', \inf_f\ v\}$ is a directed set — $f\ tv$ is an upper bound of this set — $f\ v'$ and $\inf_f\ v$ have a least upper bound.

Example 4.3.5: Let us consider the functions *andL* from Example 2.2.1 and *and* from Example 4.1.2 and their characteristic sets $C_{[\![andL]\!]}$ and $C_{[\![and]\!]}$ from Example 4.3.2. We

have

$$\begin{aligned}(\inf_{[\![andL]\!]}\langle\bot,\bot\rangle)|_\epsilon &= \bigsqcap\left\{\begin{array}{c}[\![andL]\!]\langle False, False\rangle, [\![andL]\!]\langle False, True\rangle\\ [\![andL]\!]\langle True, False\rangle, [\![andL]\!]\langle True, True\rangle\end{array}\right\}\\ &= \bigsqcap\{False, False, False, True\}\\ &= \bot\end{aligned}$$

and

$$\begin{aligned}(\inf_{[\![andL]\!]}\langle True,\bot\rangle)|_\epsilon &= \bigsqcap\{[\![andL]\!]\langle True, False\rangle, [\![andL]\!]\langle True, True\rangle\}\\ &= \bigsqcap\{False, True\}\\ &= \bot.\end{aligned}$$

By Lemma 4.3.7, which is one implication of Theorem 4.3.1, this implies that all sequential functions g are not less strict than $[\![andL]\!]$. Thus, $[\![andL]\!]$ is minimally strict. On the other hand we have

$$(\langle False,\bot\rangle,\epsilon) \in C_{[\![and]\!]}$$

and

$$\begin{aligned}(\inf_{[\![and]\!]}\langle False,\bot\rangle)|_\epsilon &= \bigsqcap\{[\![andL]\!]\langle False, False\rangle, [\![andL]\!]\langle False, True\rangle\}\\ &= \bigsqcap\{False, False\}\\ &= False\\ &\sqsupset \bot\end{aligned}$$

as well as

$$(\langle\bot, False\rangle,\epsilon) \in C_{[\![and]\!]}$$

and

$$\begin{aligned}(\inf_{[\![and]\!]}\langle\bot, False\rangle)|_\epsilon &= \bigsqcap\{[\![andL]\!]\langle False, False\rangle, [\![andL]\!]\langle True, False\rangle\}\\ &= \bigsqcap\{False, False\}\\ &= False\\ &\sqsupset \bot.\end{aligned}$$

The unproven implication of Theorem 4.3.1 will show that this implies that *and* is not minimally strict. We will prove this missing implication by showing that the existence of $(v, rp) \in C_{[\![and]\!]}$ with $(\inf_{[\![and]\!]} v)|_{rp} \sqsupset \bot$ implies that $\overline{[\![and]\!]}_{(v,rp)}$ is less strict than $[\![and]\!]$. Furthermore, the value of $(\inf_{[\![and]\!]} v)|_{rp}$ indicates how we can improve *and* with respect to strictness. The implementation would be less strict if it would yield *False* instead of $\bot$ for the argument $\langle False,\bot\rangle$ or for the argument $\langle\bot, False\rangle$. Note that there is no sequential function that satisfies both counter-examples.

As the pair $(\langle False,\bot\rangle,\epsilon)$ as well as $(\langle\bot, False\rangle,\epsilon)$ are counter-examples, there are two possibilities to define functions that are less strict than *and*, namely,

$$\overline{[\![and]\!]}_{(\langle False,\bot\rangle,\epsilon)}$$

as well as

$$\overline{[\![and]\!]}_{(\langle\bot,False\rangle,\epsilon)}.$$

As an example, the following reasoning shows that the former function yields a more defined result than $[\![and]\!]$ for the argument $\langle False, \bot\rangle$.

$$\begin{aligned}
&\overline{[\![and]\!]}_{(\langle False,\bot\rangle,\epsilon)}\ \langle False, \bot\rangle \\
&\quad = ([\![and]\!]\ \langle False, \bot\rangle)[([\![and]\!]\ \langle False, \bot\rangle \sqcup \inf_{[\![and]\!]}\ \langle False, \bot\rangle)|_\epsilon]_\epsilon \\
&\quad = \bot[(\bot \sqcup False)|_\epsilon]_\epsilon \\
&\quad = \bot[False]_\epsilon \\
&\quad = False
\end{aligned}$$

Furthermore, it yields a result that is at least as defined as the result of $[\![and]\!]$ if we consider an argument that is more defined than $\langle False, \bot\rangle$. In the case of $[\![and]\!]$ the less strict definition agrees with the infimum, for example, we have

$$\begin{aligned}
&\overline{[\![and]\!]}_{(\langle False,\bot\rangle,\epsilon)}\ \langle False, False\rangle \\
&\quad = ([\![and]\!]\ \langle False, False\rangle)[([\![and]\!]\ \langle False, False\rangle \sqcup \inf_{[\![and]\!]}\ \langle False, \bot\rangle)|_\epsilon]_\epsilon \\
&\quad = False[(False \sqcup False)|_\epsilon]_\epsilon \\
&\quad = False[False]_\epsilon \\
&\quad = False.
\end{aligned}$$

Finally, if we consider a value that is less defined than or incomparable with the value $\langle False, \bot\rangle$, the improved function yields the same result as the original function.

$$\overline{[\![and]\!]}_{(\langle False,\bot\rangle,\epsilon)}\ \langle \bot, False\rangle = [\![and]\!]\ \langle \bot, False\rangle = \bot$$

In summary the function $\overline{[\![and]\!]}_{(\langle False,\bot\rangle,\epsilon)}$ is semantically equivalent to $[\![andL]\!]$ while the function $\overline{[\![and]\!]}_{(\langle\bot,False\rangle,\epsilon)}$ is semantically equivalent to $[\![andR]\!]$. □

As a first step towards a proof that $\overline{f}$ is well-defined, by case distinction we show that $\overline{f}$ is indeed less strict than f. In particular, a function $\overline{f}_{(v,rp)}$ yields a more defined result than f if we consider the argument v.

Lemma 4.3.8: *For all $f \in [\![\sigma \to \tau]\!]$ if there exists $v \in [\![\sigma]\!]$ and $rp \in Pos\ (f\ v)$ such that $(\inf_f\ v)|_{rp} \sqsupset \bot$, then $\overline{f}_{(v,rp)}$ is less strict than f.*

The characteristic set of a function does not contain non-finite elements. For example, if we consider a function $f \in [\![[Bool] \to [Bool]]\!]$ the characteristic set C_f does not contain infinite lists.

The following lemma states that the characteristic set only contains finite elements where *finite* is defined in Definition 2.2.1. We prove this lemma by induction over the structure of the characteristic set and by employing Lemma A.3.1.

Lemma 4.3.9: *For all $(v, rp) \in C_f$ the value v is finite.*

To show that $\overline{f}$ is well-defined we have to prove that $\overline{f}$ is continuous, in other words, $\overline{f} \in [\![\sigma \to \tau]\!]$, and that $\overline{f}$ is sequential. First we show that $\overline{f}$ is a monotonic and continuous function. We prove that $\overline{f}$ is monotonic by case distinction.

Lemma 4.3.10: *Let $f \in [\![\sigma \to \tau]\!]$. If there exists $(v, rp) \in C_f$ such that $(\inf_f v)|_{rp} \sqsupset \bot$, then $\overline{f}_{(v,rp)}$ is monotonic.*

The following lemma shows that $\overline{f}$ is a continuous function. Note that we need the monotonicity of $\overline{f}$ to show that $\langle \overline{f}\, v_i \rangle_{i \in I}$ is a chain if we consider an arbitrary chain $\langle v_i \rangle_{i \in I}$. Because we consider chain-complete partial orders, this implies that the supremum

$$\bigsqcup_{i \in I} \left(\overline{f}_{(v,rp)}\, v_i \right)$$

exists. We prove that $\overline{f}$ is continuous by case distinction and the fact that the characteristic set contains only finite values.

Lemma 4.3.11: *Let $f \in [\![\sigma \to \tau]\!]$. If there exists $(v, rp) \in C_f$ such that $(\inf_f v)|_{rp} \sqsupset \bot$, then $\overline{f}_{(v,rp)}$ is continuous.*

To prove that $\overline{f}$ is sequential we have to prove a property about the characteristic set C_f. If we consider an arbitrary pair (v, rp), which is not an element of the characteristic set, for the definition of $\overline{f}_{(v,rp)}$ the resulting function might not be sequential. Therefore, the sequentiality of $\overline{f}$ has to rely on some property of the characteristic set. The following example provides a function $\overline{f}_{(v,rp)}$ that is not sequential if we use an arbitrary pair (v, rp) that is not an element of the characteristic set of f.

Example 4.3.6: For example, consider the function *andL* again. We have

$$(\inf_{[\![andL]\!]} \langle \bot, False \rangle)|_\epsilon = False \sqsupset \bot.$$

Although the pair $(\langle \bot, False \rangle, \epsilon)$ is not in the characteristic set of *andL*, we consider the function $\overline{[\![andL]\!]}_{(\langle \bot, False \rangle, \epsilon)}$ in the following. We have

$$\overline{[\![andL]\!]}_{(\langle \bot, False \rangle, \epsilon)} \langle False, \bot \rangle = [\![andL]\!] \langle \bot, False \rangle = False$$

as well as

$$\begin{aligned}
&\overline{[\![andL]\!]}_{(\langle \bot, False \rangle, \epsilon)} \langle \bot, False \rangle \\
&\quad = ([\![andL]\!] \langle \bot, False \rangle)[([\![andL]\!] \langle \bot, False \rangle \sqcup \inf_{[\![andL]\!]} \langle \bot, False \rangle)|_\epsilon]_\epsilon \\
&\quad = \bot[(\bot \sqcup False)|_\epsilon]_\epsilon \\
&\quad = False.
\end{aligned}$$

Therefore, the function considered above is not sequential as there exists no sequential position in $\langle \bot, \bot \rangle$ at position ϵ. This example demonstrates that we may not

consider arbitrary pairs of values and result positions for the definition of $\overline{f}$ but only pairs that belong to C_f. □

Let us assume that we want to prove that $\overline{f}$ is sequential. That is, for every partial value pv we have to provide a sequential position in pv. More precisely, we have to provide some position p such that for all pv' with $pv' \sqsupseteq pv$ and $pv'|_p = \bot$ we have $(\overline{f}\ pv')|_{rp} = \bot$. Now consider that there exists some partial value pv' with $pv' \sqsupseteq pv$ and $pv'|_p = \bot$ such that $pv' \sqsupseteq v$ while we have $pv \not\sqsupseteq v$ where $(v, rp) \in C_f$ is the counter-example used to define $\overline{f}$. By definition of $\overline{f}$ we have

$$\overline{f}\ pv = f\ pv$$

but

$$\overline{f}\ pv' = (f\ pv')[(f\ pv' \sqcup \inf_f\ v)|_{rp}]_{rp}.$$

Furthermore, we have

$$\begin{aligned}
(\overline{f}\ pv')|_{rp} &= ((f\ pv')[(f\ pv' \sqcup \inf_f\ v)|_{rp}]_{rp})|_{rp} \\
&= (f\ pv' \sqcup \inf_f\ v)|_{rp} && \text{Lemma 4.2.2} \\
&= (f\ pv')|_{rp} \sqcup (\inf_f\ v)|_{rp} && \text{continuity of } \cdot|_{rp}
\end{aligned}$$

As (v, rp) is a counter-example, we have $(\inf_f\ v)|_{rp} \neq \bot$, and, therefore, $(\overline{f}\ pv')|_{rp} \neq \bot$. That is, although there exists a sequential position in pv with respect to f this position is not sequential with respect to $\overline{f}$. Thus, in this case we have to prove that there exists another sequential position. More precisely, we need a sequential position such that for all pv' with $pv' \sqsupseteq pv$ and $pv'|_p = \bot$ we have $pv' \not\sqsupseteq v$ to avoid the problem described above.

In Example 4.3.6 the role of pv is taken over by $\langle \bot, \bot \rangle$ and the role of pv' is taken over by $\langle \bot, \mathit{False} \rangle$. Furthermore, we have $v = \langle \bot, \mathit{False} \rangle$, and, therefore, $pv \not\sqsupseteq v$ but $pv' \sqsupseteq v$. The definition of $\overline{[\![\mathit{andL}]\!]}$ "destroys" the sequential position in $\langle \bot, \bot \rangle$ with respect to *andL*, namely, position 1. But, as we did consider a pair (v, rp) that is not in the characteristic there exists no other sequential position in $\langle \bot, \bot \rangle$, and, therefore, $\overline{[\![\mathit{andL}]\!]}$ is not sequential. In the following we illustrate why there always exists a sequential position with the desired property if we only consider counter-examples from the characteristic set.

The following lemma proves the property of the elements of the characteristic set of a function we have just illustrated. In Lemma 4.3.13 we employ this property to prove that $\overline{f}$ is a sequential function if f is a sequential function. We prove the following lemma by employing Lemma A.3.1.

Lemma 4.3.12: *Let p be a sequential position of f in pv at position rp and $(v, rp) \in C_f$ with $v \not\sqsubseteq pv$. Then there exists a sequential position p' of f in pv at position rp such that for all $pv' \in [\![\tau]\!]$ with $pv' \sqsupseteq pv$ and $pv'|_{p'} = \bot$ we have $pv' \not\sqsupseteq v$.*

With the help of the previous lemma we are ready to prove that $\overline{f}_{(v,rp)}$ is sequential if there exists $(v, rp) \in C_f$ such that $(\inf_f\ v)|_{rp} \sqsupset \bot$.

Lemma 4.3.13: *Let $f \in [\![\sigma \to \tau]\!]$. If there exists $(v, rp) \in C_f$ such that $(\inf_f\ v)|_{rp} \sqsupset \bot$, then $\overline{f}_{(v,rp)}$ is a sequential function.* □

The following lemma finally proves that the presented criterion is necessary for the existence of a less strict, sequential function and constitutes the missing halve to the proof of Theorem 4.3.1.

Lemma 4.3.14: *Let $f \in [\![\sigma \to \tau]\!]$ be a sequential function. If there exists $(v, rp) \in C_f$ such that $(\inf_f\ v)|_{rp} \sqsupset \bot$, then there exists a sequential function $g \in [\![\sigma \to \tau]\!]$ with $g \prec f$.*

Proof: As there exists $(v, rp) \in C_f$ with $(\inf_f\ v)|_{rp} \sqsupset \bot$, we can define $\overline{f}_{(v,rp)}$ and we have $\overline{f} \prec f$ by Lemma 4.3.8. By Lemma 4.3.10 and Lemma 4.3.11 we have $\overline{f} \in [\![\sigma \to \tau]\!]$ and by Lemma 4.3.13 $\overline{f}$ is sequential. □

5 Implementation of Sloth

In this chapter we present the implementation of Sloth. More precisely, we do not present the implementation by means of source code but the basic approach in an abstract manner and by means of examples.

Sloth can be used to check whether a function is unnecessarily strict. For example, consider the Haskell implementation of the function *and* from Example 4.1.2.

$$
\begin{aligned}
&\mathit{and} :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}\\
&\mathit{and}\ \mathit{False}\ \mathit{False} = \mathit{False}\\
&\mathit{and}\ \mathit{False}\ \mathit{True}\ \ = \mathit{False}\\
&\mathit{and}\ \mathit{True}\ \ \mathit{False} = \mathit{False}\\
&\mathit{and}\ \mathit{True}\ \ \mathit{True}\ \ = \mathit{True}
\end{aligned}
$$

Sloth provides a function that is called *strictCheck* to check whether a function is minimally strict for inputs up to a specific size. The size of a term is the number of constructors in the term. For example, the following application checks whether *and* is minimally strict for all pairs of Boolean values up to size five.

```
> strictCheck and 5
2: \False ⊥ -> False
Finished 7 tests.
```

Sloth reports one counter-example that states that the evaluation of the application *and False* $\bot$ yields $\bot$ while there exists a less strict implementation that yields *False* instead. The highlighting on the right-hand side of -> denotes that there exists an implementation that yields the highlighted value while the current implementation yields $\bot$ instead of the highlighted value.

The remainder of this chapter shows how Sloth enumerates test cases and checks whether a test case is a counter-example. Here, a test case is a value of the argument type of the function. A counter-example is a test case together with the current result of the function for this argument and a proposed, more defined, result. In contrast to Chapter 4, Sloth does not use an uncurried notation but the Haskell style curried notation for test cases. In the following, we use both notations synonymously. For example, the counter-example for *and* consists of the test case $\langle \mathit{False}, \bot \rangle$, the current result $\bot$, and the proposed result *False*. Furthermore, in this chapter we present functions in Haskell syntax and abstain from presenting their counterparts in our simple functional language. Besides, we use Haskell-style list constructors $[\,]$ and $(:)$ instead of *Nil* and *Cons* $\langle \cdot, \cdot \rangle$ to emphasize the connection between the output of Sloth and our considerations on the level of semantics.

We refer to the symbol $\bot$ as *error* because this symbol, in fact, denotes a run-time error in this context. Nevertheless, it could also denote a non-terminating computation as the considered functions behave equally for run-time errors and non-terminating computations. This observation only holds if we disregard certain kinds

of exception handling mechanisms as a function otherwise could behave differently for a run-time error than for a non-terminating computation.

As Theorem 4.3.1 proves, to check whether a function f is minimally strict, we have to check whether we have $(\inf_f\ v)|_{rp} = \bot$ for all pairs (v, rp) of the characteristic set of f. A pair (v, rp) consists of a value v of the argument type of f and a position rp that references a sub-value of the value $f\ v$. The vertical bar followed by the position rp denotes the projection to position rp. For now we consider $\inf_f\ v$ as some abstract value and explain its definition in more detail later.

Intuitively, a pair (v, rp) is an element of the characteristic set of a function f, if the function may yield a more defined result for the argument v at result position rp without contradicting sequentiality. That is, if we consider a function f' that is equivalent to f but yields a more defined result for the argument v at result position rp, then f' is still sequential. Besides, for all elements (v, rp) of the characteristic set the current implementation of f yields an error for the argument v at result position rp. Thus, yielding a more defined result for an element (v, rp) means yielding a result that is not an error. In summary, the elements of the characteristic set are candidates for making a function less strict. In Section 5.1 we illustrate how Sloth enumerates the elements of the characteristic set.

To check whether there exists a sequential function that is less strict than a function f, we check whether we have $(\inf_f\ v)|_{rp} = \bot$ for all pairs (v, rp) of the characteristic set. Intuitively, the value $(\inf_f\ v)|_{rp}$ is not an error if f yields similar results at result position rp for arguments that are more defined than v. The value $(\inf_f\ v)|_{rp}$ represents this similarity. As the function behaves similar for all more defined inputs, we can define a function that yields this similarity instead of an error for the argument v at result position rp. In other words, there exists a function that yields more parts of the result of f by inspecting the same part of its argument. In Section 5.2 we show how Sloth checks whether we have $(\inf_f\ v)|_{rp} = \bot$.

5.1 Enumerating Test Cases

So, how can we enumerate the elements of the characteristic set of a function f? If a pair (v, rp) is an element of the characteristic set, by Definition 4.3.3 the value v is either the least element of the argument type of f or there exists another pair (v', rp') in the characteristic set such that $v' \lessdot_p v$ for some position p. For now we ignore the additional requirements on rp and rp' that are set up by the definition of the characteristic set.

Two values v and v' are related by $v \lessdot_p v'$ if we get v' from v by replacing an occurrence of an error in v at position p by a constructor applied to errors only. Let us consider the relation $\lessdot_p$ for values of type $Bool \times Bool$. We can represent this relation $\lessdot_p$ by a tree as shown in Figure 5.1.1. The root note of the tree is the least element of the corresponding type. Each value is followed by an argument position that contains an error, in other words, if a value v is followed by a position p, we have $v|_p = \bot$. Furthermore, if two subsequent values v and v' have an intermediate argument position p, we have $v \lessdot_p v'$. For example, in Figure 5.1.1 between the

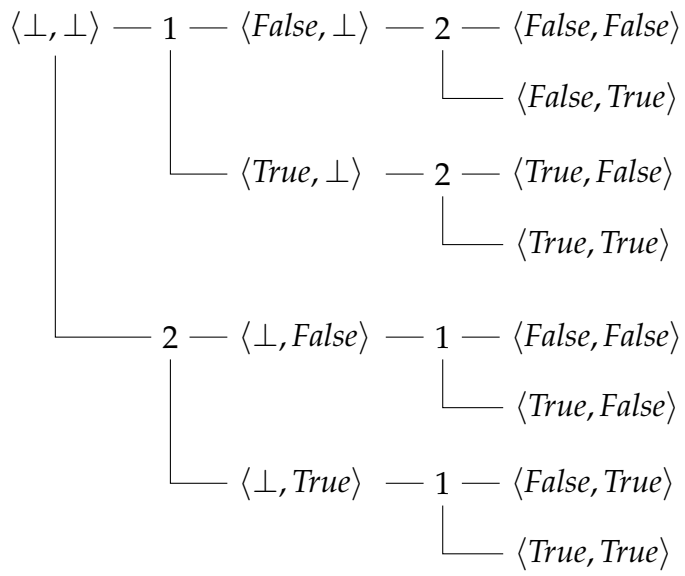


Figure 5.1.1: Tree representing the Relation $\lessdot_p$ for Pairs of Boolean Values.

values $\langle True, \bot \rangle$ and $\langle True, True \rangle$ there is a node labeled with 2 because we have $\langle True, \bot \rangle \lessdot_2 \langle True, True \rangle$.

If we omit the position nodes, the tree in Figure 5.1.1 resembles the Hasse diagram for the ordering $\sqsubseteq$ for values of type $Bool \times Bool$. The position nodes are actually redundant as we can always deduce the position from the values of the preceding and the subsequent value. Nonetheless, we keep these redundant nodes as it makes this information explicit. Additionally, while the corresponding Hasse diagram is a directed acyclic graph, for simplicity, here as well as in the implementation of Sloth we use trees. As future work, we might check whether the performance of the implementation improves when we switch from trees to directed acyclic graphs. However, in most cases Sloth only generates small parts of these trees because of lazy evaluation.

If we consider a type that has infinitely many values, the corresponding tree might have infinitely many nodes as well. Therefore, we cut the tree at the depth that corresponds to the number of constructors passed as second argument to *strictCheck*. If a value v precedes a value v' in the tree that represents $\lessdot$, then v has exactly one constructor less than v'.

To calculate the characteristic set — as we will later see — and to calculate the value $(\inf_f v)|_{rp}$, we need the results of applications of f to values that are more defined than some value v. In Figure 5.1.1 the subtree that is rooted at some value v contains all values that are more defined than v. Therefore, we add the results of applying the considered function to each node of the tree that represents the relation $\lessdot$.

As an example let us consider the function *andL*, which is the Haskell implementation of the function with the same name from Example 2.2.1.

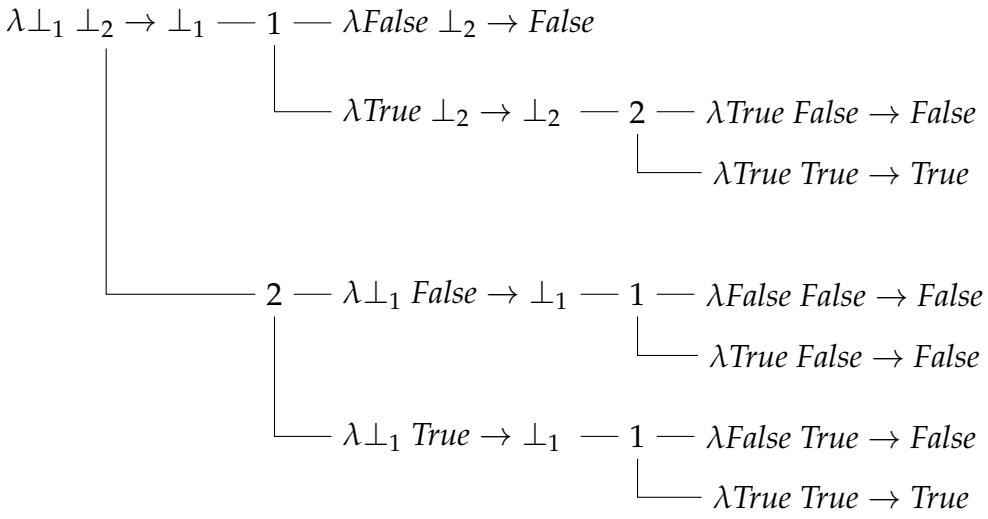


Figure 5.1.2: Tree representing the Relation $\lessdot_p$ for Pairs of Boolean Values with Results of Applying *andL* to these Values.

$$
\begin{aligned}
&\mathit{andL} :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}\\
&\mathit{andL}\ \mathit{False}\ \_ = \mathit{False}\\
&\mathit{andL}\ \mathit{True}\ \ x = x
\end{aligned}
$$

Instead of a node $\langle \mathit{False}, \bot \rangle$, with respect to the Boolean conjunction *andL* we consider a node of the form $\lambda \mathit{False}\ \bot \to \mathit{False}$. The value on the left-hand side of $\to$ is the argument and the value on the right-hand side is the result of applying the considered function to this argument. We refer to this kind of node as *mapping node*.

Figure 5.1.2 shows the result of applying *andL* to every node of the tree in Figure 5.1.1. To gain additional information we label errors in the argument value with their position in the argument by using positions as presented in Definition 4.2.2. Intuitively, an error like $\bot_1$ is represented by a run-time error that carries the position 1 as error message. By employing exception handling mechanisms, we regain these error messages from the result of an application as it is modeled by the exception semantics, presented in Figure 4.2.6. For example, as *andL* performs pattern matching on its first argument, the resulting error has the same label as the one in the first argument, namely, position 1. Using this mechanism, for every result position that contains an error we can determine the corresponding demanded position. For example, as we have $[\![\mathit{andL}]\!]\ \langle \bot_1, \bot_2 \rangle = \bot_1$, by Definition 4.2.8 position 1 is the demanded position in $\langle \bot, \bot \rangle$ at result position ϵ with respect to *andL*. The demanded position is the position in the argument that is evaluated to head normal form if we evaluate the value at a certain result position to head normal form. Note that the definition of demanded positions requires that every label of an error is unique. This requirement is satisfied as each error is labeled with its position in the argument value.

The tree in Figure 5.1.2 does not contain the nodes $\lambda \mathit{False}\ \mathit{False} \to \mathit{False}$ as well as $\lambda \mathit{False}\ \mathit{True} \to \mathit{False}$. We omit successors of mapping nodes whose result value is total. By monotonicity of the considered functions we know that all successors of a

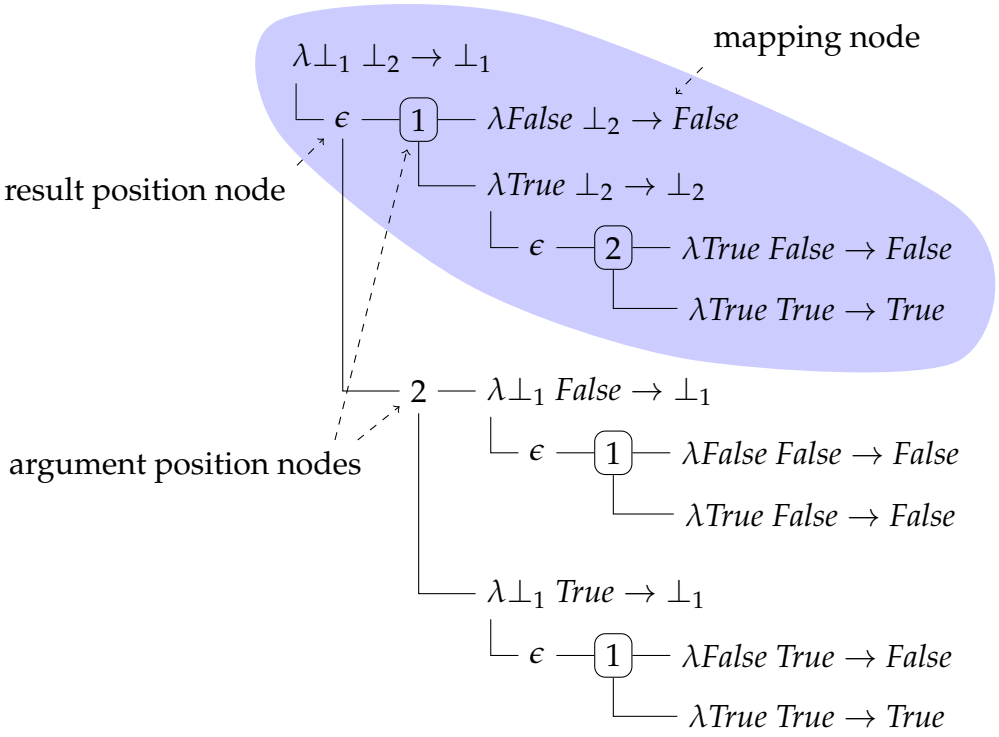


Figure 5.1.3: The Refinement Tree for *andL*, the Highlighted Part represents the Characteristic Set for *andL*.

node like this will yield the same result. For example, as we have $[\![andL]\!]\ \langle False, \bot\rangle = False$ we get $[\![andL]\!]\ \langle False, False\rangle = False$ as well as $[\![andL]\!]\ \langle False, True\rangle = False$.

Our final goal is to calculate the characteristic set of *andL*. As an element of the characteristic set is a pair consisting of a value and a result position, we have to add result positions to the tree in Figure 5.1.2. Intuitively, to calculate the characteristic set, for every value v and every result position rp such that $(f\ v)|_{rp} = \bot$, we have to identify all corresponding sequential positions. Therefore, we define a tree in which every value v is followed by all eligible result positions. Each result position is itself followed by all possible sequential positions with respect to the preceding result position and the preceding value. This way we define a tree that represents a superset of the characteristic set. We refer to this tree as *refinement tree*. To calculate the characteristic set we prune the refinement tree. Figure 5.1.3 presents the refinement tree for *andL*. In the following, we explain the three kinds of nodes of a refinement tree in detail.

mapping node A mapping node is labeled with a value of the argument type and the result of the function for this argument. For example, the root node in Figure 5.1.3, labeled with $\lambda\bot_1\ \bot_2 \to \bot_1$, is a mapping node. To gain additional information we label errors in the argument value with their position in the argument. Using these labels, at each result position we can easily identify the corresponding demanded position.

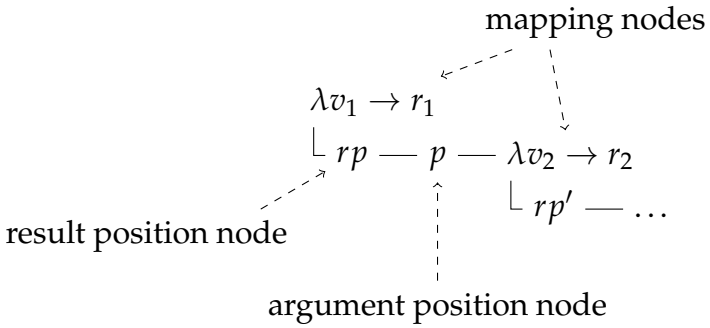


Figure 5.1.4: A Path of a Refinement Tree.

result position node For every position of an error in the result value of a mapping node, the node has a successor that is labeled with this position. We refer to this kind of nodes as result position nodes. For example, the mapping node labeled with $\lambda\bot_1\ \bot_2 \to \bot_1$ has only one child, labeled with ϵ, because ϵ is the only position of an error in the result value of this mapping node, namely, $\bot_1$. Result position nodes of a refinement tree are ordered by the position ordering $\leq$. If a result position node labeled with rp is an ancestor of a result position node labeled with rp', then rp is a prefix of rp'. We may omit other result positions by the definition of the characteristic set as we will observe soon.

argument position node For every position of an error in the argument value of a mapping node, the corresponding result position nodes have a successor that is labeled with the position of the error in the argument value. We refer to this kind of nodes as argument position nodes. For example, the successors of the topmost result position node, labeled with ϵ, are labeled with 1 and 2 because 1 and 2 are the positions of errors in the argument value of the preceding mapping node, namely, $\lambda\bot_1\ \bot_2 \to \bot_1$. In Figure 5.1.3 we additionally mark argument position nodes that correspond to demanded positions with respect to the preceding result position node.

In summary, on a path between two mapping nodes there is always a result position node and an argument position node as illustrated in Figure 5.1.4. Here v_1 and v_2 are argument values and r_1 and r_2 are result values. Furthermore, rp is a result position such that $r_1|_{rp} = \bot$, rp' is a result position such that $r_2|_{rp'} = \bot$, and p is an argument position such that $v_1|_p = \bot$.

By definition of the characteristic set, the root node of a refinement tree is an element of the characteristic set. More precisely, all pairs that contain the argument of the root mapping node and the result position of a subsequent result position node are elements of the characteristic set. In the case of the refinement tree for *andL* the pair $(\langle\bot, \bot\rangle, \epsilon)$ is an element of the characteristic set, because ϵ is the result position of a result position node that succeeds the mapping node $\lambda\bot_1\ \bot_2 \to \bot_1$.

When we consider a path as shown in Figure 5.1.4, if the pair (v_1, rp) is in the characteristic set, then the pair (v_2, rp') is in the characteristic set as well if the step from v_1 to v_2 is sequential. Furthermore, rp' has to be a prefix of rp and we need

$(f\ v_2)|_{rp'} = \bot$. The last two requirements are satisfied by the definition of the refinement tree. Therefore, if the pair (v_1, rp) is an element of the characteristic set, we only have to check whether the step from v_1 to v_2 is sequential to observe that (v_2, rp') is also an element of the characteristic set. By definition of the refinement tree we have $v_1 \lessdot_p v_2$ and p is either a sequential or a non-sequential position. That is, all argument position nodes either correspond to a sequential or a non-sequential position by the definition of the refinement tree. In the rest of this section we illustrate how we identify whether a position is sequential or non-sequential.

To determine additional elements of the characteristic set in Figure 5.1.3 besides the root node, we have to identify sequential positions in $\langle \bot, \bot \rangle$ at result position ϵ. Position 1 and position 2 are possible sequential positions in $\langle \bot, \bot \rangle$ at result position ϵ. Lemma 4.2.5 states that all demanded positions are sequential positions. Thus, we can identify one sequential position easily. Position 1 is a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ because it is a demanded position in $\langle \bot, \bot \rangle$ at result position ϵ. This way we can quite efficiently identify one sequential position for every mapping node and every subsequent result position node.

As we have observed in Example 4.2.9, there are sequential positions that are not demanded positions. Therefore, we have to check whether any of the other possible sequential positions is indeed a sequential position. For example, position 2 might be a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ. To eliminate non-sequential positions, Sloth searches for witnesses that prove that an argument position is a non-sequential position. For example, let us consider the possible sequential positions in $\langle \bot, \bot \rangle$ at result position ϵ, namely, position 2. The mapping node labeled with $\lambda \mathit{False}\ \bot_2 \to \mathit{False}$ proves that position 2 is not a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ as we have already observed in Example 4.2.5. To show that p is a non-sequential position in v at result position rp, by definition of non-sequential positions we have to find a value v' such that $v' \sqsupseteq v$ and $v'|_p = \bot$ but $(f\ v')|_{rp} \neq \bot$. In the particular case at hand we have $\langle \mathit{False}, \bot \rangle \sqsupseteq \langle \bot, \bot \rangle$ and

$$\langle \mathit{False}, \bot \rangle|_2 = \bot \qquad \text{but} \qquad (\llbracket \mathit{andL} \rrbracket\ \langle \mathit{False}, \bot \rangle)|_\epsilon \neq \bot.$$

Therefore, position 2 is not a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ with respect to *andL*.

To check whether a position p is a non-sequential position in a value v at position rp, Sloth searches the subtree for mapping nodes of the form $\lambda v' \to r'$ such that

$$v'|_p = \bot \qquad \text{and} \qquad r'|_{rp} \neq \bot.$$

A mapping node with this property shows that position p is not a sequential position in v at result position rp. Note that we have $v' \sqsupseteq v$ as v' is an element of the subtree that is rooted at the mapping node, whose argument is v. Furthermore, note that a single witness might prove that a couple of argument positions are non-sequential.

We do not have to randomly search the tree for witnesses. Instead we can employ the information that is gained by identifying demanded positions. Because position 1 is the demanded position and, therefore, a sequential position in $\langle \bot, \bot \rangle$, at result position ϵ we have $\llbracket \mathit{andL} \rrbracket\ \langle \bot, v \rangle = \bot$ for all values v of type *Bool*. Therefore, we

will not discover a witness (arguments v_1 and v_2 such that $[\![andL]\!]\ \langle v_1, v_2\rangle \neq \bot$) as long as the first argument is $\bot$. When we search a tree for witnesses, at result position nodes we only consider subtrees with respect to argument position nodes whose positions are demanded positions. For example, we consider the mapping node $\lambda\bot_1\ \bot_2 \to \bot_1$. Because the argument position node labeled 1 is marked as demanded position, we only search the subtree that is rooted at this argument position node for witnesses.

As position 2 is not a sequential position in $\langle\bot, \bot\rangle$ at result position ϵ, we only keep the highlighted part of the refinement tree in Figure 5.1.3. In this case the resulting tree is already the characteristic set. For each result position node the tree contains only a single argument position node, which is the demanded position in each case.

If we consider a function whose argument type has infinitely many values, checking whether a position is non-sequential cannot be done this way. We might simply miss a witness because we do not consider arguments of sufficiently large sizes. For now, we ignore this case but consider it in Section 5.4.

5.2 Checking Test Cases

After we have identified the tree that represents the characteristic set, we have to check whether we have $(\inf_f\ v)|_{rp} = \bot$ for all argument values v of mapping nodes and all subsequent result positions rp. All pairs (v, rp) with $(\inf_f\ v)|_{rp} \neq \bot$ are counter-examples. As we have

$$\inf_f\ v = \bigsqcap\{f\ tv \mid tv \in [\![\tau]\!]\uparrow, v \sqsubseteq tv\},$$

for every value v in the tree we have to compute the set $\{f\ tv \mid tv \in [\![\tau]\!]\uparrow, v \sqsubseteq tv\}$. For example, to check whether mapping node $\lambda\bot_1\ \bot_2 \to \bot_1$ together with result position ϵ is a counter-example we search the corresponding subtree for mapping nodes whose arguments are total values. In this particular case, we apply *andL* to $\langle False, \bot\rangle$, $\langle True, False\rangle$, and $\langle False, False\rangle$ and collect the values *False*, *True*, and *False*. This way we observe that we have

$$(\inf_{[\![andL]\!]}\ \langle\bot, \bot\rangle)|_\epsilon = (\bigsqcap\{False, False, True\})|_\epsilon = \bot|_\epsilon = \bot,$$

and, hence, the partial value $\langle\bot, \bot\rangle$ together with result position ϵ is not a counter-example. When we search for total values, we, again, only consider demanded positions. Note that the first argument, namely, $\langle False, \bot\rangle$, is not a total value. A mapping node whose result value is a total value does not have any successors.

In the same way as we have observed that $(\langle\bot, \bot\rangle, \epsilon)$ is not a counter-example, we observe that all elements of the characteristic set of *andL* are no counter-examples, thus, by Theorem 4.3.1 *andL* is minimally strict.

The argument type of *andL* has only finitely many values. If we consider functions whose argument type has infinitely many values, we are faced with additional problems. In the following we present the approach that is used to address these

problems. We illustrate that Sloth distinguishes the following kinds of counter-examples. Sloth uses the same highlighting as the words "potential" and "definite" to indicate the corresponding kind of counter-example.

definite **counter-example** A definite counter-example stays a counter-example no matter what size we use for the test cases.

potential **counter-example** A potential counter-example might be no counter-example if we consider inputs of larger sizes. That is, there might be a size greater than the current size such that Sloth does not report the potential counter-example anymore.

In the following we explain the difference between the two kinds of counter-examples in detail. If the argument type τ of a function f has infinitely many elements, the set

$$\mathit{Total}\ v := \{f\ tv \mid tv \in [\![\tau]\!]\uparrow, v \sqsubseteq tv\}$$

might be infinite for some value v of type τ. To check whether an element of the characteristic set is a counter-example, we have to calculate the greatest lower bound of the set $\mathit{Total}\ v$. If we approximate this set, that is, we calculate a set T' that is a proper subset of $\mathit{Total}\ v$, we get $\bigsqcap T' \sqsupseteq \bigsqcap \mathit{Total}\ v$ because the operator $\bigsqcap$ is monotonically decreasing. Let us consider that we have discovered a counter-example, namely, a pair (v, rp) with $(\bigsqcap T')|_{rp} \sqsupset \bot$. If we consider an approximation T'' that is more precise than the approximation T', we might have

$$\bigsqcap T' \sqsupset \bigsqcap T'' \qquad \text{and} \qquad (\bigsqcap T')|_{rp} = \bot.$$

Thus, with respect to the more precise approximation the pair (v, rp) is not a counter-example anymore.

The tool presented by Chitil (2006) classifies all arguments v with $\bigsqcap T' \sqsupset f\ v$ as unnecessarily strict. In contrast, we try to identify counter-examples that stay counter-examples even if we consider an arbitrarily precise approximation. Consider the following definition.

$$\begin{aligned}
&\mathit{potential} :: [\mathit{Bool}] \to \mathit{Bool}\\
&\mathit{potential}\ [\,] \quad\ = \mathit{True}\\
&\mathit{potential}\ (_ : xs) = \mathit{potential}\ xs
\end{aligned}$$

Obviously, this function is unnecessarily strict. It performs pattern matching although it yields *True* for all total inputs. Sloth reports the following counter-examples if we check *potential* for Boolean lists up to size three.

```
> strictCheck potential 3
1: \⊥ -> True
3: \(⊥:⊥) -> True
Finished 7 tests.
```

In contrast to the counter-examples for *intersperse* from Section 3.2, these counter-examples are potential counter-examples. That is, there might be a size greater than

three such that Sloth does not report some of the counter-examples it reports for size three. One way to confirm a potential counter-example is to increase the size. However, Sloth still reports only potential counter-examples for the function *potential* if we consider arguments up to size four. The counter-examples even stay potential no matter what size we use.

If we take a closer look at the test cases, we might wonder why the largest counter-example has only one constructor although we have checked *potential* for lists up to size three, that is, lists with at most three constructors. When we check *potential* for lists up to size three, using *verboseCheck*, we get the following result. In contrast to *strictCheck*, the function *verboseCheck* additionally reports successful test cases.

```
> verboseCheck potential 3
1: \⊥ -> True
2: \[] -> True
3: \(⊥:⊥) -> True
4: \(⊥:[]) -> True
5: \(⊥:⊥:⊥) -> ?
6: \(⊥:⊥:[]) -> True
7: \(⊥:⊥:⊥:⊥) -> ?
Finished 7 tests.
```

As the output illustrates that current Sloth implementation checks lists up to size three but does show all results when we use *strictCheck*.

If the size of the approximation T' of the set *Total* v is very small, the result is quite likely to be wrong. For example, if we consider a set T' with a single element to approximate the set *Total* v, then we most certainly have $(\bigsqcap T')|_{rp} \sqsupset \bot$. Therefore, Sloth only considers approximations of the set *Total* v with at least three elements. The question marks on the right-hand sides of -> state that the number of elements of the approximation is too small in these cases. By using a configuration we can specify the minimum number of elements for the approximation of the set *Total* v. Although there are probably more useful heuristics than demanding a fixed number of elements, we have not explored other heuristics.

Let us go back to the problem of identifying definite counter-examples. Consider the following variation of the function *potential*.

$$\begin{aligned} &\mathit{potential}' :: [\mathit{Bool}] \to \mathit{Bool} \\ &\mathit{potential}'\ (\_:\_:\_) = \mathit{False} \\ &\mathit{potential}'\ \_\quad\quad\ = \mathit{True} \end{aligned}$$

This function is semantically equivalent to *potential* with respect to arguments up to size two. Therefore, if we only consider total values up to size two, $\mathit{potential}'$ is unnecessarily strict. The function is supposed to yield the value *True* without inspecting its argument as $\mathit{potential}'$ yields *True* for all total arguments up to size two. However, if we consider the total value $[\mathit{False}, \mathit{False}]$, then we observe that $\mathit{potential}'$ is not unnecessarily strict as we have $[\![\mathit{potential}']\!]\,[\mathit{False}, \mathit{False}] = \mathit{False}$. That is, obviously, in general, we cannot decide whether a test case is a counter-example by only considering a finite number of total values.

We can manually verify a potential counter-example. A potential counter-example is definitely a counter-example if all more defined total inputs lead to results that are at least as defined as the recommended result. For example, consider the first counter-example for *potential*. For all total inputs that are more defined than $\bot$ the function *potential* yields *True*, which is as defined as the recommended result *True*. Therefore, the first counter-example is definitely a counter-example. Note that in most cases it is easy to verify a potential counter-example as we only have to consider the behavior of a function with respect to total values.

Although some counter-examples are only potential counter-examples, there are cases in which a finite approximation is sufficient to identify a definite counter-example. For example, consider the following function.

$$\begin{aligned} &\mathit{definite} :: [\mathit{Bool}] \to [\mathit{Bool}] \\ &\mathit{definite}\ [\,] \quad\ = [\mathit{True}] \\ &\mathit{definite}\ (\_ : \mathit{xs}) = \mathit{True} : \mathit{definite}\ \mathit{xs} \end{aligned}$$

Both counter-examples for *definite* are definite counter-examples as it is indicated by the highlighting.

```
> strictCheck definite 2
1: \⊥ -> True:⊥
3: \(⊥:⊥) -> True:True:⊥
Finished 5 tests.
```

Sloth employs monotonicity to identify definite counter-examples. Let us consider the first counter-example. Sloth observes that we have

$$[\![\mathit{definite}]\!]\ [\,] = \mathit{True} : [\,]$$

as well as

$$[\![\mathit{definite}]\!]\ (\bot : \bot) = \mathit{True} : \bot.$$

Informally, these equations state that, no matter whether the argument of *definite* is an empty or a non-empty list, the result is a list with at least one element whose first element is *True*. We can formally justify this statement as follows. As *definite* is monotonic, we have

$$[\![\mathit{definite}]\!]\ v \sqsupseteq \mathit{True} : \bot$$

for all Boolean lists v such that $v \sqsupseteq \bot : \bot$. To check whether *definite* is unnecessarily strict for the argument $\bot$ at result position ϵ we have to show that the following inequality holds.

$$(\inf_{[\![\mathit{definite}]\!]}\ \bot)|_\epsilon \sqsupset \bot.$$

We can show this strict inequality by the following estimate.

$$\begin{aligned} (\inf_{[\![\mathit{definite}]\!]}\ \bot)|_\epsilon &= \bigsqcap \{[\![\mathit{definite}]\!]\ tv \mid tv \in [\![[\mathit{Bool}]]\!]\uparrow, \bot \sqsubseteq tv\} \\ &= [\![\mathit{definite}]\!]\ [\,] \sqcap \bigsqcap \{[\![\mathit{definite}]\!]\ tv \mid tv \in [\![[\mathit{Bool}]]\!]\uparrow, \bot : \bot \sqsubseteq tv\} \end{aligned}$$

$$
\begin{aligned}
&= \mathit{True} : [\,] \sqcap \bigsqcap \{ [\![\mathit{definite}]\!]\ \mathit{tv} \mid \mathit{tv} \in [\![[\mathit{Bool}]]\!]\uparrow, \bot : \bot \sqsubseteq \mathit{tv} \} \\
&\sqsupseteq \mathit{True} : [\,] \sqcap [\![\mathit{definite}]\!]\ (\bot : \bot) \\
&= \mathit{True} : [\,] \sqcap \mathit{True} : \bot \\
&= \mathit{True} : \bot
\end{aligned}
$$

Hence, we can observe that *definite* is unnecessarily strict for the input $\bot$ by only considering two applications. As a side effect this approach allows us to evaluate only a small part of the refinement tree to check whether an application is unnecessarily strict. More precisely, to observe that the application $[\![\mathit{definite}]\!]\ \bot$ is unnecessarily strict we even do not have to search the tree for applications of *definite* to total values. Instead, we only have to evaluate a single level of the tree as the applications $[\![\mathit{definite}]\!]\ [\,]$ and $[\![\mathit{definite}]\!]\ (\bot : \bot)$ already provide sufficient information.

In the case of *definite* the estimate is actually an equality. However, there are cases in which we get a strict inequality instead. As a consequence, the counter-examples reported by Sloth may not agree with the behavior of a minimally strict implementation. For example, consider the following definition, which is a variation of *definite*.

$$
\begin{aligned}
&\mathit{definite}' :: [\mathit{Bool}] \to [\mathit{Bool}] \\
&\mathit{definite}'\ [\,] \qquad = [\mathit{True}, \mathit{True}] \\
&\mathit{definite}'\ (\_ : \mathit{xs}) = \\
&\quad \mathit{True} : \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
&\qquad\qquad [\,] \quad \to [\mathit{True}] \\
&\qquad\qquad \_ : \mathit{ys} \to \mathit{True} : \mathit{definite}'\ \mathit{ys}
\end{aligned}
$$

If we check this function, Sloth reports the following counter-examples.

```
> strictCheck definite' 2
1: \⊥ -> True:⊥
3: \(⊥:⊥) -> True:True:⊥
Finished 5 tests.
```

That is, although a minimally strict implementation obviously yields $\mathit{True} : \mathit{True} : \bot$ for the argument $\bot$, Sloth proposes the result $\mathit{True} : \bot$ as the following strict inequality illustrates.

$$
\begin{aligned}
(\inf_{[\![\mathit{definite}']\!]}\ \bot)|_\epsilon &= \bigsqcap \{ [\![\mathit{definite}']\!]\ \mathit{tv} \mid \mathit{tv} \in [\![[\mathit{Bool}]]\!]\uparrow, \bot \sqsubseteq \mathit{tv} \} \\
&= [\![\mathit{definite}']\!]\ [\,] \sqcap \bigsqcap \{ [\![\mathit{definite}']\!]\ \mathit{tv} \mid \mathit{tv} \in [\![[\mathit{Bool}]]\!]\uparrow, \bot : \bot \sqsubseteq \mathit{tv} \} \\
&= \mathit{True} : \mathit{True} : [\,] \sqcap \bigsqcap \{ [\![\mathit{definite}']\!]\ \mathit{tv} \mid \mathit{tv} \in [\![[\mathit{Bool}]]\!]\uparrow, \bot : \bot \sqsubseteq \mathit{tv} \} \\
&\sqsupset \mathit{True} : \mathit{True} : [\,] \sqcap \mathit{True} : \bot \\
&= \mathit{True} : \bot
\end{aligned}
$$

This is the price that we have to pay for identifying counter-examples as definite and for evaluating smaller parts of the tree.

5.3 Presenting Counter-Examples

As we have observed in Chapter 4, if we consider only sequential functions, we are looking for minimal elements with respect to the less-strict ordering. Therefore, there might be several less strict implementations for one function that are contradictory to each other.

If we check *and* using *verboseCheck*, Sloth does not present all elements of the characteristic set of *and* (see Example 4.3.2 for the characteristic set of *and*). Instead, Sloth reports the following test cases.

```
> verboseCheck and 4
1: \⊥ ⊥ -> ⊥
2: \False ⊥ -> False
3: \True ⊥ -> ⊥
4: \False False -> False
5: \False True -> False
6: \True False -> False
7: \True True -> True
Finished 7 tests.
```

Figure 5.3.1 shows the tree that is generated by Sloth to check whether *and* is minimally strict. In this case the tree does not contain any witnesses that a position is non-sequential. When we consider a function whose argument type has only finitely many elements, like *and*, the absence of a witness implies that all remaining positions are sequential. We will later consider the case of a function whose argument type has infinitely many values.

The highlighted pairs of mapping node and result position node in Figure 5.3.1 are counter-examples. As we have observed in Example 4.3.5, there exists no sequential Boolean conjunction that satisfies both counter-examples. Instead of presenting all, potentially contradictory, counter-examples, Sloth presents one set of test cases that contains a counter-example and is completely satisfiable by a single sequential implementation.

By employing the structure of a refinement tree we can quite easily identify sets that are satisfiable by a single sequential function. For every result position node we may only consider a single argument position node. In other words, for every result position node we only keep a single successor and drop the other successors. The argument positions that remain after this process resemble the demanded positions of the proposed less strict implementation. Furthermore, the resulting tree resembles a kind of **case** cascade that implements the pattern matching of the less strict implementation up to a given size.

In general, we may not keep more than one argument position node per result position node as we otherwise might propose an implementation that is not sequential. For example, let us assume that we choose argument position 1 at the topmost result position node of the tree in Figure 5.3.1. In this case we are supposed to implement a function and' such that $[\![and']\!]\,\langle False, \bot\rangle \neq \bot$. By sequentiality we have $[\![and']\!]\,\langle \bot, v\rangle = \bot$ for all Boolean values v. Therefore, we have to drop all counter-examples that occur in the tree rooted at argument position 2 and do not occur in the

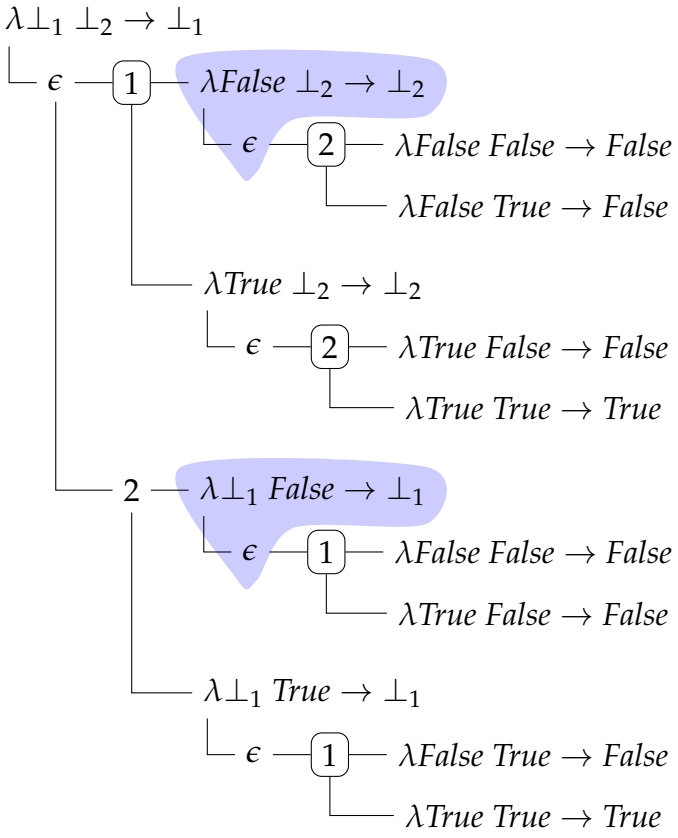


Figure 5.3.1: The Refinement Tree for *and* and two Counter-Examples.

tree rooted at argument position 1 as these are exactly the applications of the form $[\![and']\!]\ \langle \bot, v \rangle$ for $v \neq \bot$. We can apply the same argument to show that we may only choose a single argument position if we consider an arbitrary result position node of an arbitrary refinement tree.

Thus, in Figure 5.3.1, with respect to the outermost result position node, we have to select either the subtree that is rooted at argument position 1 or the subtree that is rooted at argument position 2. The subtree that is rooted at argument position 1 resembles *andL* while the subtree that is rooted at argument position 2 resembles *andR*. If there are multiple subtrees that contain counter-examples, we apply the following approach. First we select the subtree that contains the smallest counter-example with respect to the number of constructors. In other words, if there are two subtrees that contain counter-examples we choose the tree whose counter-example occurs in the smaller depth. In the case of *and* both counter-examples occur in depth one.

If one of the argument positions in question is a demanded position, we consider the corresponding subtree. That is, in the case of *and* we choose argument position 1 and not position 2 because position 1 is the demanded position of $\langle \bot, \bot \rangle$ at result position ϵ. Note that *and* first performs pattern matching on its first argument because of the left-to-right pattern matching order used by Haskell. By preferring demanded positions we keep the changes with respect to the pattern matching order minimal that have to be applied to get a less strict implementation.

If none of the argument positions in question is a demanded position, we choose the leftmost argument position. More precisely, we choose the smallest argument position with respect to a lexicographical order on positions. We think that this decision is the most natural one as it reflects the default left-to-right pattern matching order of Haskell.

5.4 Identifying Sequential Positions

As we have observed before, if we consider a function whose argument type has infinitely many values, the absence of a witness does not imply that a position is sequential. We might simply have to consider arguments of larger sizes to find a witness and, hence, observe that a position is non-sequential.

In this section we introduce a third highlighting, which highlights a whole test case. In contrast to previous examples, sometimes we are not able to decide whether a test case is an element of the characteristic set. Therefore, we introduce the following concept of potential test cases. Sloth uses the same highlighting as the word "potential" to indicate that a test case is a potential test case.

potential test case A potential test case might be no test case anymore if we consider inputs of larger sizes.

As an example, let us consider the following function that takes two Boolean lists and checks whether its first argument is at least as long as its second argument.

$$
\begin{aligned}
&\mathit{greaterEqual} :: [\mathit{Bool}] \to [\mathit{Bool}] \to \mathit{Bool}\\
&\mathit{greaterEqual}\ [\,]\ \ \ \ [\,]\ \ \ \ = \mathit{True}\\
&\mathit{greaterEqual}\ (\_ : \_)\ \ [\,]\ \ \ \ = \mathit{True}\\
&\mathit{greaterEqual}\ (\_ : \mathit{xs})\ (\_ : \mathit{ys}) = \mathit{greaterEqual}\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

If we check *greaterEqual*, Sloth yields the following counter-example that is marked as potential test case.

```
> strictCheck greaterEqual 3
2: \⊥ [] -> True
Finished 7 tests.
```

Thus, the counter-example might be no test case anymore if we consider inputs of a larger size. Therefore, the reported counter-example might be no counter-example anymore if we consider inputs of larger sizes. More precisely, there is no witness that proves that position 2 is a non-sequential position in $\langle \bot, \bot \rangle$ at result position ϵ if we consider arguments up to size three. Hence, position 2 might also be a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ. If it is a sequential position, then the pair $(\langle \bot, [\,] \rangle, \epsilon)$ is an element of the characteristic set and, thus, the presented counter-example is indeed a counter-example. If, on the other hand, position 2 is a non-sequential position in $\langle \bot, \bot \rangle$ at result position ϵ, then the pair $(\langle \bot, [\,] \rangle, \epsilon)$ is not an element of the characteristic set and, thus, the counter-example is not a counter-example.

Nevertheless, for every non-sequential position there exists a finite witness that shows that this position is non-sequential. If a test case is not an element of the characteristic set, by checking the function for arguments up to a larger size, Sloth will eventually discover this witness and discard the test case. However, Sloth might as well never discard the test case, no matter which size we use, as the corresponding position is indeed a sequential position.

In fact, Sloth does not identify the test case for *greaterEqual* as a potential test case because it applies an additional criterion to identify sequential positions. The only way to identify a sequential position is to observe that it is a demanded position. For example, we have $[\![\mathit{greaterEqual}]\!]\ \langle \bot_1, \bot_2 \rangle = \bot_1$, and, hence, position 1 is a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ. So, why is position 2 a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ as well? After *greaterEqual* has performed pattern matching on its first argument, it performs pattern matching on its second argument. In other words, we have

$$[\![\mathit{greaterEqual}]\!]\ \langle [\,], \bot_2 \rangle = \bot_2$$

and

$$[\![\mathit{greaterEqual}]\!]\ \langle (\bot_{1.1} : \bot_{1.2}), \bot_2 \rangle = \bot_2.$$

That is, if we refine the first argument, in both cases position 2 becomes a demanded position. As every demanded position is also a sequential position, we have

$$[\![\mathit{greaterEqual}]\!]\ \langle [\,], \bot \rangle = \bot$$

and

$$[\![\mathit{greaterEqual}]\!]\ \langle v, \bot \rangle = \bot$$

for all lists v with $v \sqsupseteq \bot : \bot$. As we, furthermore, have $[\![\mathit{greaterEqual}]\!]\ \langle \bot, \bot \rangle = \bot$, we get $[\![\mathit{greaterEqual}]\!]\ \bot v \bot = \bot$ for all lists v. Thus, position 2 is also a sequential position in $\langle \bot, \bot \rangle$. This way, Sloth observes that the test case $(\langle \bot, [\,] \rangle, \epsilon)$ is definitely an element of the characteristic set of *greaterEqual* as position 2 is a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ.

However, even with this additional approach to identifying non-sequential positions, in some cases Sloth only reports potential test cases. For example, consider the following constant function.

$$\begin{aligned}
&\mathit{constBool} :: [\mathit{Bool}] \to \mathit{Bool} \to \mathit{Bool} \\
&\mathit{constBool}\ [\,] \qquad b = b \\
&\mathit{constBool}\ (_ : \mathit{bs})\ b = \mathit{constBool}\ \mathit{bs}\ b
\end{aligned}$$

All test cases for *constBool* are potential elements of the characteristic set.

```
> strictCheck constBool 5
2: \⊥ False -> False
3: \⊥ True -> True
5: \(⊥:⊥) False -> False
7: \(⊥:⊥) True -> True
Finished 15 tests.
```

Position 1 is a sequential position in $\langle \bot, \bot \rangle$ because it is a demanded position in $\langle \bot, \bot \rangle$, in other words, we have $[\![constBool]\!]\ \langle \bot_1, \bot_2 \rangle = \bot_1$. However, in contrast to *greaterEqual*, the function *constBool* does not demand its second argument after it has performed pattern matching on its first argument. We have

$$[\![constBool]\!]\ \langle [\,], \bot_2 \rangle = \bot_2$$

but

$$[\![constBool]\!]\ \langle (\bot_{1.1} : \bot_{1.2}), \bot_2 \rangle = \bot_{1.2}.$$

More generally speaking, *constBool* projects to its second argument if it is applied to a list that is terminated by an empty list while it projects to the according error if the list is terminated by an error. In other words, we have

$$[\![constBool]\!]\ \langle (\bot_{1.1} : \ldots : [\,]), \bot_2 \rangle = \bot_2$$

and

$$[\![constBool]\!]\ \langle (\bot_{1.1} : \ldots : \bot_{1.n}), \bot_2 \rangle = \bot_{1.n}.$$

Therefore, Sloth is not able to observe that we have $[\![constBool]\!]\ \langle v, \bot \rangle = \bot$ for all Boolean lists v by considering finitely many test cases only. There might as exist an argument such that *constBool* does not demand its second argument. For example, the following definition is semantically equivalent with *constBool* if we consider lists up to size 2.

$$\begin{aligned} &constBool' :: [Bool] \to Bool \to Bool \\ &constBool'\ (\_ : \_ : \_)\ \_ = True \\ &constBool'\ \_\qquad\quad\ b = b \end{aligned}$$

However, position 2 is not a sequential position in $\langle \bot, \bot \rangle$ at result position ϵ with respect to $constBool'$ because we have $[\![constBool']\!]\ \langle (\bot : \bot : \bot), \bot \rangle = True$. In other words, as position 2 is a non-sequential position in $\langle \bot, \bot \rangle$ at result position ϵ, there exists a finite witness that proves this fact, in this particular case the smallest witness is the argument $\langle (\bot : \bot : \bot), \bot \rangle$.

6 Minimally Strict Polymorphic Functions

In Chapter 4 we have only considered monomorphic functions, that is, functions f of type $\tau_1 \to \tau_2$ where τ_1 and τ_2 are monomorphic types. In this section we show how we can check whether a polymorphic function is minimally strict by checking whether a specific monomorphic instance is minimally strict. More precisely, we show that we only have to check a subset of all possible inputs for the integer instance of a polymorphic function. This approach leads to a surprisingly small number of test cases. For example, when we consider functions of type $[\alpha] \to [\alpha]$ we only have to check a linear number of test cases in the number of elements in the list while there is an exponential number of test cases in the number of elements in the list for all naive monomorphic instances.

More precisely, we show that we can check whether two polymorphic functions are related by the less-strict relation by checking the functions for all inputs where we replace the polymorphic components by integers that encode the position of the component in the data structure. This approach is similar to an approach by Bernardy et al. (2010), but, in contrast, we consider non-termination and *seq*[1]. Furthermore, we show that we can use a similar approach to check whether a polymorphic function is minimally strict. In fact, this is our actual goal as we want to use the results of this chapter the improve the efficiency of testing polymorphic functions using Sloth. On the way to this efficient test of polymorphic functions we obtain several results about polymorphic functions in Haskell with/without a strict evaluation primitive. For example, we characterize a polymorphic data structure like a list by its shape and its content and prove that a polymorphic function is characterized by its behavior on shapes. This approach is similar to the container view by Bundy and Richardson (1999) and Prince et al. (2008) but, in contrast, we consider non-termination and the strict evaluation primitive *seq*.

In contrast to Chapter 4, in this chapter we do not strictly separate syntax and semantics because otherwise the proofs presented here become very hard to conceive. This approach is often used in the context of equational reasoning for functional programs (Gibbons 1999; Danielsson and Jansson 2004; Fernandes et al. 2007) because the semantics of a functional program is quite similar to the program itself. Note that we, in contrast to some works that use equational reasoning, still consider the influence of $\bot$, that is, the influence of run-time errors and non-terminating expressions.

In the following by $\equiv$ we denote semantic equivalence. Furthermore, we relate syntactic objects by the cpo ordering $\sqsubseteq$ and denote that their semantics are related. When we have to consider several monomorphic instances of a polymorphic func-

[1]The function $seq :: \alpha \to \beta \to \beta$ satisfies the laws $seq\ \bot\ y \equiv \bot$ and $seq\ x\ y \equiv y$ if $x \not\equiv \bot$.

tion, we use a subscript type to indicate the specific instance. All proofs that are omitted in this section are found in the Appendix B.

6.1 Introduction

Consider the following polymorphic function called *inits*, which is defined in the standard Haskell library *Data.List*. This function takes a list and yields a list of all initial segments of this list, shortest first.

$$\begin{aligned} &\mathit{inits} :: [\alpha] \to [[\alpha]] \\ &\mathit{inits}\ [\,] \quad\ = [[\,]] \\ &\mathit{inits}\ (x : xs) = [[\,]] +\!\!+ \mathit{map}\ (x:)\ (\mathit{inits}\ xs) \end{aligned}$$

To check this function, Sloth generates test cases, namely, elements of the argument type of *inits*. Therefore, we have to choose a monomorphic instance of the polymorphic function *inits*. Obviously, we want to choose the monomorphic instance of the polymorphic function that results in the smallest number of test cases. A candidate that might come into mind is the monomorphic instance for the unit type $()$, which has only a single element denoted by $()$. Though, unfortunately, that a monomorphic unit instance of a polymorphic function is unnecessarily strict does not imply that the corresponding polymorphic function is unnecessarily strict. For example, consider the identity function $\mathit{id}\ x = x$. The function $\mathit{id} :: () \to ()$, that is, the monomorphic unit instance of the polymorphic identity function, is unnecessarily strict. Let us consider the unary function that emerges from applying *const*[2] to $()$. The function $\mathit{const}\ () :: () \to ()$ is less strict than *id* because we have $\mathit{id}\ \bot \equiv \bot$ but $\mathit{const}\ ()\ \bot \equiv ()$ and $\mathit{id}\ () \equiv ()$ as well as $\mathit{const}\ ()\ () \equiv ()$. That is, for the argument $\bot$ the result of $\mathit{const}\ ()$ is more defined than the result of *id*. Though, apparently, there exists no polymorphic function of type $\alpha \to \alpha$ whose unit instance behaves like $\mathit{const}\ ()$.

As we have already mentioned in Section 3.2, to check whether a polymorphic function is as little strict as possible we instantiate all occurrences of type variables by an opaque type, called *A*, which is provided by Sloth. If we check this instance of *inits* for lists up to size two, Sloth reports the following counter-examples.

```
> strictCheck (inits :: [A] -> [[A]]) 2
1: \⊥ -> []:⊥
3: \(a:⊥) -> []:(a:[]):⊥
Finished 5 tests.
```

The first counter-example states that $\mathit{inits}\ \bot$, yields $\bot$ while there exists a less strict implementation that yields $[\,] : \bot$ instead. The second counter-example states that $\mathit{inits}\ (a : \bot)$, where a is an arbitrary value, yields $[\,] : \bot$ while there exists a less strict implementation that yields $[\,] : (a : [\,]) : \bot$ instead.

If we reconsider the implementation of *inits*, we observe that it checks whether its argument is the empty list although the first element of the result list is always the

[2] The function $\mathit{const} :: \alpha \to \beta \to \alpha$ is defined by $\mathit{const}\ x\ \_ = x$.

empty list. If we consider the following equivalent definition of *inits*, this circumstance becomes even more obvious.

$$
\begin{array}{l}
\mathit{inits} :: [\alpha] \to [[\alpha]] \\
\mathit{inits}\ \mathit{xs} = \\
\quad \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad [\,] \quad\ \to [\,] : [\,] \\
\qquad y : \mathit{ys} \to [\,] : \mathit{map}\ (y:)\ (\mathit{inits}\ \mathit{ys})
\end{array}
$$

Thus, we can apply **case** deferment to derive the following less strict implementation. That is, we apply the same transformation as we have used to derive a less strict implementation of *intersperse* in Section 3.2. However, note that we first have to apply another transformation to *inits* to observe that both right-hand sides of the **case** expression have the same context. In Section 7.1 we will consider **case** deferment in more detail.

$$
\begin{array}{l}
\mathit{inits}' :: [\alpha] \to [[\alpha]] \\
\mathit{inits}'\ \mathit{xs} = \\
\quad [\,] : \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad [\,] \quad\ \to [\,] \\
\qquad\qquad y : \mathit{ys} \to \mathit{map}\ (y:)\ (\mathit{inits}'\ \mathit{ys})
\end{array}
$$

If we reconsider the test cases generated by Sloth, we observe that Sloth has only generated five test cases to check *inits* for lists up to size two. Furthermore, if we check *inits* for lists up to size four, Sloth generates only nine test cases. In contrast, Sloth checks exponentially many test cases in the length of the list if we test any monomorphic instance of *inits*. For example, if we check the Boolean instance of *inits* for lists up to size four, Sloth generates 21 test cases. As Sloth manages the same test with linearly many test cases, it obviously treats the data type A in a special way. In this chapter we prove that the approach for A-instances, which is used by Sloth, is correct.

In Section 4 two functions are only related by the less-strict relation if they agree for total arguments. For simplicity, we omit this requirement here, but we can extend all results to the original definition by considering total arguments as a special case. In other words, in this chapter we employ the following definition of the less-strict relation for monomorphic functions.

Definition 6.1.1 (Less-Strict Relation, Monomorphic Functions): Let $f, g :: \tau_1 \to \tau_2$ be monomorphic functions. The functions f and g are related by $\preceq$ if the following holds.

$$f \preceq g :\Longleftrightarrow \forall x :: \tau_1 . f\ x \sqsupseteq g\ x$$

The right-hand side is equivalent to $f \sqsupseteq g$ by the semantic ordering of functions. □

As this definition of the less-strict relation, as well as the original Definition 4.1.2, only consider monomorphic functions, we have to provide a generalization to polymorphic functions. Two polymorphic functions are related by the less-strict relation if and only if all their monomorphic instances are related by the less-strict relation.

Here, f_τ denotes the instantiation of the polymorphic function f to the monomorphic type τ.

Definition 6.1.2 (Less-Strict Relation, Polymorphic Functions): Let f and g be polymorphic functions of equal type. The functions f and g are related by $\preceq$ if the following holds.

$$f \preceq g :\Longleftrightarrow \forall \tau :: *. f_\tau \preceq g_\tau$$

By $\tau :: *$ we denote that τ is a type. □

Here and in the following, we only consider non-empty types. That is, every type τ has an inhabitant besides the least element $\bot$. Finally, note that most of the results, presented in this chapter, are not restricted to considerations about the less-strict relation as we can define $f \equiv g$ by $f \preceq g \wedge g \preceq f$. For example, we can employ our results to perform the tests by Coutts et al. (2007) more efficiently. They check whether list functions that are implemented by means of their stream fusion library are equivalent to their original definition by checking whether the two implementations agree for all partial inputs. We cannot only apply the results to the area of testing but also to any form of verification that considers the influence of errors and non-terminating expressions. For example, Abel et al. (2005) have presented a transformation of Haskell programs into a monadic representation of the program, to reason about Haskell programs using the proof assistent Agda. Employing the results of the chapter at hand we can improve the efficiency of proving statements about polymorphic functions using their approach.

6.2 Free Theorems

The proofs in this chapter make heavy use of free theorems (Wadler 1989). Free theorems are semantic statements about functions that are derived from type information only. That is, without considering the implementation of a function we can derive semantic statements about a function. Free theorems are a mighty proof tool, which have been applied successfully in a variety of contexts (Gill et al. 1993; Johann 2002; Voigtländer 2008a,b; Oliveira et al. 2010).

Consider the following definition of the standard Haskell function *reverse*, which reverses a list.

$$\begin{aligned}
&\mathit{reverse} :: [\alpha] \to [\alpha] \\
&\mathit{reverse} = \mathit{rev}\ [\,] \\
&\quad \textbf{where} \\
&\qquad \mathit{rev}\ \mathit{xs}\ [\,] \quad\ = \mathit{xs} \\
&\qquad \mathit{rev}\ \mathit{xs}\ (y : \mathit{ys}) = \mathit{rev}\ (y : \mathit{xs})\ \mathit{ys}
\end{aligned}$$

For all functions $g :: \tau_1 \to \tau_2$ and all lists $xs :: [\tau_1]$ the following equation holds. By the subscript types we indicate that the two applications of *reverse* are, in fact, specific monomorphic instances.

$$\mathit{map}\ g\ (\mathit{reverse}_{\tau_1}\ \mathit{xs}) \equiv \mathit{reverse}_{\tau_2}\ (\mathit{map}\ g\ \mathit{xs})$$

Intuitively, this equality states that it does not matter, whether we first reverse a list and apply a function to all elements of the result list afterwards or first apply a function to all elements of a list and reverse the result afterwards.

We can justify this property as follows. The function *reverse* takes a list and reverses the order of the elements of the list. For example, consider a list

$$xs \equiv [x_1, \ldots, x_n].$$

We have

$$reverse_{\tau_1}\ xs \equiv [x_n, \ldots, x_1].$$

The behavior of *reverse* does not depend on the elements of the argument list, it only depends on the shape of the argument list. The behavior of *reverse* cannot depend on the elements of the list because it is polymorphic, and, therefore, it cannot inspect the elements of the argument list. Furthermore, *map* preserves the shape of a list. That is, *reverse* permutes *xs* in the same manner as it permutes *map g xs*. More precisely, we have

$$map\ g\ xs \equiv [g\ x_1, \ldots, g\ x_n]$$

and

$$reverse_{\tau_2}\ (map\ g\ xs) \equiv [g\ x_n, \ldots, g\ x_1].$$

Therefore, we get

$$map\ g\ (reverse_{\tau_1}\ xs) \equiv reverse_{\tau_2}\ (map\ g\ xs).$$

Now let us consider an arbitrary function f of type $[\alpha] \to [\alpha]$ instead of *reverse*. As f is polymorphic with respect to the element type of the list, the behavior of f can only depend on the shape of the list. For example, consider a list

$$xs \equiv [x_1, \ldots, x_n].$$

A polymorphic function like f can only use the elements of the argument list as elements of the result list. But, in contrast to *reverse*, an arbitrary function f does not have to use every element of the argument for the result, and, furthermore, it may also use elements more than once. Thus, there exists a function i and a natural number m such that *index* maps the indicies $1, \ldots, m$ of the result list to indicies $1, \ldots, n$ of the argument list and we have

$$f_{\tau_1}\ xs \equiv [x_{index\ 1}, \ldots, x_{index\ m}].$$

Furthermore, as *map g xs* has the same shape as *xs* the function f chooses the same elements from the list *map g xs* and we get

$$f_{\tau_2}\ (map\ g\ xs) \equiv [g\ x_{index\ 1}, \ldots, g\ x_{index\ m}].$$

Hence, even in the more general case of an arbitrary function $f :: [\alpha] \to [\alpha]$ we get

$$map\ g\ (f_{\tau_1}\ xs) \equiv f_{\tau_2}\ (map\ g\ xs).$$

The statement that for all functions $f :: [\alpha] \to [\alpha]$, for all functions $g :: \tau_1 \to \tau_2$ and all lists $xs :: [\tau_1]$ we have

$$map\ g\ (f_{\tau_1}\ xs) \equiv f_{\tau_2}\ (map\ g\ xs).$$

is called the free theorem for the type $[\alpha] \to [\alpha]$. In the following we will also use the term "free theorem for g", where g is a function. In this case we mean the free theorem for the type of g where we replace monomorphic instances of the generic function $f :: [\alpha] \to [\alpha]$ by monomorphic instances of g. For example, the free theorem for *reverse* is an instance of the free theorem for the type $[\alpha] \to [\alpha]$ and states

$$map\ g\ (reverse_{\tau_1}\ xs) \equiv reverse_{\tau_2}\ (map\ g\ xs).$$

That is, we replace monomorphic instances of the generic function f in the free theorem for the type $[\alpha] \to [\alpha]$ by instances of *reverse*.

We can derive free theorems for all polymorphic function types. For example, consider the function

$$filter :: (\alpha \to Bool) \to [\alpha] \to [\alpha],$$

which filters the elements of a list by a given predicate. The free theorem for *filter* states that for all functions $f :: \tau_1 \to \tau_2$ and predicates $p :: \tau_1 \to Bool$, $q :: \tau_2 \to Bool$ if we have

$$p\ x \equiv q\ (f\ x)$$

for all $x :: \tau_1$, then we have

$$map\ f\ (filter_{\tau_1}\ p\ xs) \equiv filter_{\tau_2}\ q\ (map\ f\ xs)$$

for all $xs :: [\tau_1]$.

Throughout this chapter we employ various free theorems without stating them explicitly. The web application *Automatic generation of free theorems* can be used to generate these theorems from the corresponding type (Böhme 2007; Seidel and Voigtländer 2009). If the step of a proof is labeled "free theorem for f", we can generate the free theorem by inserting the type of f into the web application. For example, the tool generates the following output for the type $[\alpha] \to [\alpha]$.

The theorem generated for functions of the type

$$f :: \forall\ a.\ [a] \to [a]$$

in the sublanguage of Haskell with no bottoms, is:

$$\forall\ t_1, t_2\ \in\ TYPES, R\ \in\ REL\ (t_1, t_2).$$
$$\forall\ (x, y)\ \in\ lift\{[\,]\}(R).\ (f_{t_1}\ x, f_{t_2}\ y)\ \in\ lift\{[\,]\}(R)$$

$$lift\{[\,]\}(R)$$
$$= \{([\,], [\,])\}\ \cup\ \{(x : xs, y : ys)\ |\ ((x, y)\ \in\ R) \wedge ((xs, ys)\ \in\ lift\{[\,]\}(R))\}$$

Reducing all permissible relation variables to functions yields:

$$\forall\, t_1, t_2 \;\in\; \mathit{TYPES}, g :: t_1 \to t_2.$$
$$\quad \forall\, x :: [t_1]\,.\; \mathit{map}_{t_1 t_2}\; g\; (f_{t_1}\; x) = f_{t_2}\; (\mathit{map}_{t_1 t_2}\; g\; x)$$

The most general form of a free theorem considers relations instead of functions. The first part of the output presents the free theorem in relational form. The relation lift$\{[\,]\}(R)$ relates two lists x and y if and only if the lists have the same shape and their elements are point-wise related by the relation R. Thus, the relational free theorem states that, if the elements of two lists x and y are point-wise related by a relation R, then the f-images of x and y are also point-wise related by R.

In contrast to this most general form of a free theorem, the most common form of a free theorem is the instantiation of all relations with functions. That is, instead of an arbitrary relation R we consider only functions. In this case the relation lift$\{[\,]\}(R)$ used in the relational form becomes an application of the list mapping *map f* for a function f that corresponds to the relation R. The functional form is more common as it is closer to the functional program, and the power of relational free theorems is only rarely necessary. Later in this chapter we will see an example where we need the power of relational free theorems to prove a statement.

As the output of the automatic generator states, the presented theorem only holds in the "sublanguage of Haskell with no bottoms", that is, in a language without run-time errors and non-termination. For example, consider the following instantiation of the free theorem for the type $[\alpha] \to [\alpha]$.

$$f :: [\alpha] \to [\alpha]$$
$$f\ \_ = [\bot]$$

$$g :: \mathit{Bool} \to \mathit{Bool}$$
$$g\ \_ = \mathit{True}$$

If we consider the argument $[\,]$, we have

$$\mathit{map}\ g\ (f\ [\,]) \equiv \mathit{map}\ g\ [\bot] \equiv [\mathit{True}]$$

while on the other hand we have

$$f\ (\mathit{map}\ g\ [\,]) \equiv f\ [\,] \equiv [\bot].$$

Hence, $\mathit{map}\ g\ (f\ xs) \equiv f\ (\mathit{map}\ g\ xs)$ does not hold for all functions $g :: \tau_1 \to \tau_2$ and all lists $xs :: [\tau_1]$. The intuitive explanation of the free theorem fails in the presence of $\bot$ because the polymorphic function cannot only use the elements of the argument list for the result list, but it can also introduce $\bot$. Wadler (1989) has already observed that g has to be strict[3] if we consider run-time errors and non-termination. The automatic generation of free theorems generates the side conditions that are necessary to fix the free theorem in the presence of these features if we state that we want to consider "general recursion".

[3] A function f is strict if $f\ \bot \equiv \bot$.

An extension of free theorems by Johann and Voigtländer (2004) even considers the influence of the strict evaluation primitive *seq*, which satisfies the laws $seq\ \bot\ y \equiv \bot$ and $seq\ x\ y \equiv y$ if $x \not\equiv \bot$. Free theorems break in the presence of the standard Haskell function *seq* even if we only consider strict functions. For example, consider the following instantiation of the free theorem for the type $[\alpha] \to [\alpha]$.

$$f :: [\alpha] \to [\alpha]$$
$$f\ (x : y : \_) = [seq\ x\ y]$$

$$g :: Bool \to Bool$$
$$g\ True = True$$

If we consider the argument $[False, True]$, we have

$$map\ g\ (f\ [False, True]) \equiv map\ g\ [seq\ False\ True] \equiv map\ g\ [True] \equiv [True]$$

while on the other hand we have

$$f\ (map\ g\ [False, True]) \equiv f\ [\bot, True] \equiv [seq\ \bot\ True] \equiv [\bot].$$

To fix free theorems in the presence of *seq* we have to assure that certain functions are total[4]. In the case of the free theorem above g has to be total. In Section 6.3 we consider statements in a language without *seq*, and in Section 6.4 we consider a language with *seq*.

6.3 Less Strict Functions on Lists

In this section we consider functions of type $[\alpha] \to [\alpha]$ that do not use *seq*. We show that, if any monomorphic instance of two polymorphic functions is related by the less-strict relation, then all other monomorphic instances are related as well. This result implies that two polymorphic functions are already related if one of their monomorphic instances is related. Note that this does not imply that a polymorphic function is unnecessarily strict if a monomorphic instance is unnecessarily strict. As we have observed in the introduction of this chapter, there may exist a monomorphic, less strict function that does not have a polymorphic generalization.

As a first step towards the result concerning the less-strict relation, we prove for polymorphic functions $f :: [\alpha] \to [\alpha]$ and $g :: [\alpha] \to [\alpha]$ and an arbitrary type τ that $f_\tau \preceq g_\tau$ implies $f_{()} \preceq g_{()}$.

Lemma 6.3.1: *Let $f, g :: [\alpha] \to [\alpha]$. For all types τ the following holds.*

$$f_\tau \preceq g_\tau \Longrightarrow f_{()} \preceq g_{()}$$

Intuitively, this statement is proved by employing a free theorem and relating all elements x of type τ such that $x \not\equiv \bot$ with $()$ and $\bot_\tau$ with $\bot_{()}$. Note that the proof employs that the type τ has an inhabitant.

[4] A function f is total if for all x with $x \not\equiv \bot$ we have $f\ x \not\equiv \bot$.

Proving the converse of Lemma 6.3.1 is considerably more complex than proving Lemma 6.3.1. We start with the definition of a function $shape :: [\alpha] \to [Int]$ that replaces the elements of a list by their index in the list, starting with index 0.

Definition 6.3.1 (Shape): We define the following function that replaces all elements of a list by increasing natural numbers, starting with 0.

$$\begin{aligned} &shape :: [\alpha] \to [Int] \\ &shape = zipWith\ (\lambda n\ \_ \to n)\ [0\,..] \end{aligned}$$

The expression $[0\,..]$ generates an infinite list of increasing integers starting with 0. The definition of the function *zipWith* can be found in Section 2.1.5. □

For example, we get the results

$$\begin{aligned} shape\ [False, True, True] &\equiv [0, 1, 2], \\ shape\ (\texttt{'z'} : \texttt{'a'} : \bot) &\equiv 0 : 1 : \bot, \end{aligned}$$

and

$$shape\ [\bot, 8] \equiv [0, 1].$$

Note that $0 : 1 : \bot$ is different from $[0, 1, \bot]$ as the former one denotes a list that contains the elements 0 and 1 and is terminated by $\bot$ while the latter one denotes a list that contains the elements 0, 1, and $\bot$ and is terminated by the empty list. To be more precise we could as well write $0 : 1 : \bot_{[Int]}$ and $[0, 1, \bot_{Int}]$.

The function *shape* is similar to the function *template* defined by Voigtländer (2009). While *shape* yields only the shape of a list, *template* additionally yields a mapping from indices to the original elements of the list. By using this mapping it is possible to reconstruct the original list. Here, we define the mapping by means of the list indexing function $(!!) :: [\alpha] \to Int \to \alpha$. The function $(!!)$ takes a list and an index in the list (where the smallest index is 0) and yields the element at the corresponding position. If the index does not exist, the function yields $\bot$ instead. The definition of $(!!)$ is presented in Section 2.1.4. More importantly than the fact that we use list indexing, *shape* can handle partially defined lists, while *template* cannot. For example, we have $template\ (3 : 10 : \bot) \equiv \bot$. Note that this difference is essential as we want to use *shape* to prove statements about the less-strict relation.

To prove that we can characterize a list by its *shape* and $(!!)$ we prove the following lemma. The characterization of a list by means of *shape* and $(!!)$ is similar to the container approach by Prince et al. (2008). As Prince et al. (2008) do not consider $\bot$ or *seq*, a list is represented by its length (that is, the shape, appearing as the parameter of a dependent type) and a function from index to element.

Lemma 6.3.2: *For all $x :: \tau$ and $xs :: [\tau]$ the following holds.*

$$map\ ((x : xs)!!)\ [1\,..] \equiv map\ (xs!!)\ [0\,..]$$

Although this statement looks quite obvious, we have to use a relational free theorem to prove it. We cannot use a standard free theorem as we have to somehow incorporate that $(x : xs)!!$ on the left-hand side is never applied to the index 0.

Now we are ready to show that a list is reconstructible from its shape by employing list indexing. In the following proof we employ the free theorem for *zipWith*, which can be generated by entering "*zipWith*" into the free theorem generator. The theorem states that for all functions f, g, h, p, and q of appropriate type that satisfy

$$h\ (p\ x\ y) \equiv q\ (f\ x)\ (g\ y)$$

for all values $x :: \tau_1$ and $y :: \tau_2$, we have

$$map\ h\ (zipWith\ p\ z\ v) \equiv zipWith\ q\ (map\ f\ z)\ (map\ g\ v)$$

for all lists $z :: [\tau_1]$ and $v :: [\tau_2]$. We give a detailed proof of the following lemma as we reconsider it in the presence of *seq*.

Lemma 6.3.3: *For all lists $xs :: [\tau]$ the following holds.*

$$map\ (xs!!)\ (shape\ xs) \equiv xs$$

Proof: Proof by structural induction over *xs*.

Base Cases: If $xs \equiv \bot$ or $xs \equiv [\,]$, we reason as follows.

$$\begin{array}{ll}
& map\ (xs!!)\ (shape\ (\bot\ /\ [\,])) \\
\equiv & \{\ \text{definition of } shape\ \} \\
& map\ (xs!!)\ (zipWith\ (\lambda n\ _ \to n)\ [0\,..]\ (\bot\ /\ [\,])) \\
\equiv & \{\ \text{definition of } zipWith\ \} \\
& map\ (xs!!)\ (\bot\ /\ [\,]) \\
\equiv & \{\ \text{definition of } map\ \} \\
& \bot\ /\ [\,]
\end{array}$$

Inductive Step: If $xs \equiv y : ys$, we reason as follows.

$$\begin{array}{lll}
& map\ ((y:ys)!!)\ (shape\ (y:ys)) & \\
\equiv & \{\ \text{definition of } shape\ \} & \\
& map\ ((y:ys)!!)\ (zipWith\ (\lambda n\ _ \to n)\ [0\,..]\ (y:ys)) & \\
\equiv & \{\ \text{definition of } [0\,..]\ \} & \\
& map\ ((y:ys)!!)\ (zipWith\ (\lambda n\ _ \to n)\ (0:[1\,..])\ (y:ys)) & \\
\equiv & \{\ \text{definition of } zipWith\ \} & \\
& map\ ((y:ys)!!)\ (0 : zipWith\ (\lambda n\ _ \to n)\ [1\,..]\ ys) & \\
\equiv & \{\ \text{definition of } map\ \} & \\
& ((y:ys)\ !!\ 0) : map\ ((y:ys)!!)\ (zipWith\ (\lambda n\ _ \to n)\ [1\,..]\ ys) & \\
\equiv & \{\ \text{definition of } (!!)\ \} & \\
& y : map\ ((y:ys)!!)\ (zipWith\ (\lambda n\ _ \to n)\ [1\,..]\ ys) & \\
\equiv & \{\ \text{free theorem for } zipWith, ((y:ys)!!) \text{ strict}\ \} & (*) \\
& y : zipWith\ (\lambda n\ _ \to n)\ (map\ ((y:ys)!!)\ [1\,..])\ ys) & \\
\equiv & \{\ \text{Lemma 6.3.2}\ \} & \\
& y : zipWith\ (\lambda n\ _ \to n)\ (map\ (ys!!)\ [0\,..])\ ys) & \\
\equiv & \{\ \text{free theorem for } zipWith, (ys!!) \text{ strict}\ \} & (**)
\end{array}$$

$$
\begin{array}{ll}
& y : map\ (ys!!)\ (zipWith\ (\lambda n\ \_ \to n)\ [0\,..]\ ys) \\
\equiv & \quad \{\ \text{induction hypothesis}\ \} \\
& y : ys
\end{array}
$$

To show the use of a free theorem in detail, we exemplarily state the instantiation of the free theorem for *zipWith* to prove the step labeled with $(*)$.

$$
\begin{array}{lll}
h = ((y : ys)!!) & q = \lambda n\ \_ \to n & z = [1\,..] \\
p = \lambda n\ \_ \to n & f = ((y : ys)!!) & v = ys \\
g = id &&
\end{array}
$$

It is easy to show that we have $h\ (p\ x\ y) \equiv q\ (f\ x)\ (g\ y)$ for all appropriate x and y, which proves the step labeled with $(*)$. We return to the steps labeled with $(*)$ and $(**)$ when we consider the presence of *seq*. □

The just proved characterization of lists directly allows for a characterization of polymorphic list functions via their behavior on shapes. The *Int* instance of the polymorphic function f may be compared with a corresponding container morphism of f (Prince et al. 2008).

The following proof employs the free theorem for functions of type $[\alpha] \to [\alpha]$, which states that we have $f_{\tau_2}\ (map\ g\ xs) \equiv map\ g\ (f_{\tau_1}\ xs)$ for all strict functions $g :: \tau_1 \to \tau_2$ and all $xs :: [\tau_1]$.

Lemma 6.3.4: *For all functions $f :: [\alpha] \to [\alpha]$ and all lists $xs :: [\tau]$ the following holds.*

$$f_\tau\ xs \equiv map\ (xs!!)\ (f_{Int}\ (shape\ xs))$$

Proof: Let $xs :: [\tau]$. We reason as follows.

$$
\begin{array}{lll}
& f_\tau\ xs & \\
\equiv & \quad \{\ \text{Lemma 6.3.3}\ \} & \\
& f_\tau\ (map\ (xs!!)\ (shape\ xs)) & \\
\equiv & \quad \{\ \text{free theorem for } f, (xs!!) \text{ strict}\ \} & (\dagger) \\
& map\ (xs!!)\ (f_{Int}\ (shape\ xs)) &
\end{array}
$$

We return to the step labeled with $(*)$ when we consider the presence of *seq*. □

Lemma 6.3.4 enables us to prove a kind of standard lemma about functions of type $[\alpha] \to [\alpha]$. It is a formalization of the intuitive explanation of the validity of free theorems.

A polymorphic function of type $[\alpha] \to [\alpha]$ can only use the elements of its argument list and $\bot$ for the elements of the result list. Therefore, for every element in the result list there exists a position in the argument list, where it is taken from, or it is $\bot$. Furthermore, because the function is polymorphic, it can distinguish argument lists only by their shape. Thus, if we apply a polymorphic function to two lists with equal shape it chooses elements from the same positions of the argument list for the result list.

The following proof employs an instance of the free theorem for functions of type $[\alpha] \to \alpha$. More precisely, for a number $n :: Int$ we consider the function $(!!n) :: [\alpha] \to \alpha$. The corresponding free theorem states that for all strict functions $g :: \tau_1 \to \tau_2$ and all $xs :: [\tau_1]$ we have $(map\ g\ xs)\ !!\ n \equiv g\ (xs\ !!\ n)$.

Lemma 6.3.5: *Let $f :: [\alpha] \to [\alpha]$ and $xs :: [\tau_1]$ and $ys :: [\tau_2]$ such that $shape\ xs \equiv shape\ ys$. Then, for all $n :: Int$ there exists $k :: Int$ with*

$$(f_{\tau_1}\ xs)\ !!\ n \equiv xs\ !!\ k \quad \textit{and} \quad (f_{\tau_2}\ ys)\ !!\ n \equiv ys\ !!\ k.$$

Proof: Let $xs :: [\tau_1]$ and $n :: Int$. We define k as $(f_{Int}\ \ (shape\ \ xs))\ !!\ n$ and reason as follows.

$$\begin{array}{ll}
& (f_{\tau_1}\ xs)\ !!\ n \\
\equiv & \quad \{ \text{ Lemma 6.3.4 } \} \\
& (map\ (xs!!)\ (f_{Int}\ (shape\ xs)))\ !!\ n \\
\equiv & \quad \{ \text{ free theorem for } (!!n), (xs!!) \text{ strict } \} \\
& xs\ !!\ ((f_{Int}\ (shape\ xs))\ !!\ n) \\
\equiv & \quad \{ \text{ definition of } k \} \\
& xs\ !!\ k
\end{array}$$

Let $ys :: [\tau_2]$ such that $shape\ xs \equiv shape\ ys$. We reason as follows.

$$\begin{array}{ll}
& (f_{\tau_2}\ ys)\ !!\ n \\
\equiv & \quad \{ \text{ like above } \} \\
& ys\ !!\ ((f_{Int}\ (shape\ ys))\ !!\ n) \\
\equiv & \quad \{ shape\ xs \equiv shape\ ys, \text{ definition of } k \} \\
& ys\ !!\ k
\end{array}$$

Note that, if n is out of bounds, then $(f_{Int}\ (shape\ xs))\ !!\ n \equiv \bot$, that is, $k \equiv \bot$. □

Back on the road, the following lemma is the essential lemma to prove that $f_{()} \preceq g_{()}$ implies $f_\tau \preceq g_\tau$ for all types τ, or, more precisely, to prove its contraposition, that is, $f_\tau \not\preceq g_\tau$ implies $f_{()} \not\preceq g_{()}$. This proof relies on mapping a list $xs :: [\tau]$ such that $f_\tau\ xs \not\sqsupseteq g_\tau\ xs$ to a list $ys :: [()]$ such that $f_{()}\ ys \not\sqsupseteq g_{()}\ ys$. But, constructing a suitable map, we encounter the following problem. When we consider an arbitrary type τ, the type $[\tau]$ may contain values xs and ys such that $xs\ !!\ k \not\equiv \bot$, $ys\ !!\ k \not\equiv \bot$, and $xs\ !!\ k \sqsupset ys\ !!\ k$. This case does not occur for values of type $[()]$ as the elements of a list of type $[()]$ are either $\bot$ or $()$. By the following lemma we elude this problem. It shows that for every problematic list there exists a non-problematic list that can be considered instead.

Lemma 6.3.6: *Let $f, g :: [\alpha] \to [\alpha]$. For every $xs :: [\tau]$ and $n :: Int$ such that*

$$(f_\tau\ xs)\ !!\ n \not\equiv \bot,\ (g_\tau\ xs)\ !!\ n \not\equiv \bot,\ \textit{and}\ (f_\tau\ xs)\ !!\ n \not\equiv (g_\tau\ xs)\ !!\ n$$

there exists $ys :: [\tau]$ such that

$$(f_\tau\ ys)\ !!\ n \equiv \bot\ \textit{and}\ (g_\tau\ ys)\ !!\ n \not\equiv \bot.$$

Now we are ready to prove that $f_\tau \not\preceq g_\tau$ implies $f_{()} \not\preceq g_{()}$ for all types τ. We employ that we have $xs \not\sqsupseteq ys$ if and only if $shape\ xs \not\sqsupseteq shape\ ys$ or $xs\,!!\,n \not\sqsupseteq ys\,!!\,n$ for some $n :: Int$. Furthermore, if there exists a list $xs :: [\tau]$ such that $(f_\tau\ xs)\,!!\,n \not\equiv \bot$, $g_\tau\ xs\,!!\,n \not\equiv \bot$, and $(f_\tau\ xs)\,!!\,n \not\equiv (g_\tau\ xs)\,!!\,n$ by the previous lemma we can instead consider a list $ys :: [\tau]$ such that $(f_\tau\ ys)\,!!\,n \equiv \bot$ and $(g_\tau\ ys)\,!!\,n \not\equiv \bot$.

Lemma 6.3.7: *Let $f, g :: [\alpha] \to [\alpha]$. If there exists a type τ such that $f_\tau \not\preceq g_\tau$, then $f_{()} \not\preceq g_{()}$.*

Let us summarize the statements so far. If for any type the monomorphic instances of two polymorphic funcitons are related by the less-strict relation, then for all other types the monomorphic instances are related by the less-strict relation as well.

Theorem 6.3.1: *Let $f, g :: [\alpha] \to [\alpha]$. For all types τ_1 and τ_2 the following holds.*

$$f_{\tau_1} \preceq g_{\tau_1} \Longleftrightarrow f_{\tau_2} \preceq g_{\tau_2}$$

Proof: By Lemma 6.3.1 and Lemma 6.3.7 we have $f_\tau \preceq g_\tau \Longleftrightarrow f_{()} \preceq g_{()}$ for all types τ. We prove the statement by instantiations of τ to τ_1 and τ_2. □

Employing Theorem 6.3.1 we can check whether two polymorphic functions are related by the less-strict relation by checking whether the unit instances are related by the less-strict relation. But, the characterization of lists by means of *shape* allows for a statement that is even more powerful than Theorem 6.3.1 with respect to exhaustive testing. We prove that it suffices to check two polymorphic functions for all possible images of *shape* whether they are related by the less-strict relation. Note that, if we consider lists of length n, then there are only linearly many shapes while there are exponentially many lists of type $[\tau]$ for any non-empty type τ.

Theorem 6.3.2: *For $f, g :: [\alpha] \to [\alpha]$ the following statement holds.*

$$f_{Int} \circ shape \preceq g_{Int} \circ shape \Longleftrightarrow f \preceq g$$

Proof: $\Longrightarrow$**:** Let $xs :: [\tau]$. We reason as follows.

$$\begin{array}{ll}
& f_\tau\ xs \\
\equiv & \quad \{\ \text{Lemma 6.3.4}\ \} \\
& map\ (xs!!)\ (f_{Int}\ (shape\ xs)) \\
\sqsupseteq & \quad \{\ \text{monotonicity}, f_{Int} \circ shape \preceq g_{Int} \circ shape\ \} \\
& map\ (xs!!)\ (g_{Int}\ (shape\ xs)) \\
\equiv & \quad \{\ \text{Lemma 6.3.4}\ \} \\
& g_\tau\ xs
\end{array}$$

$\Longleftarrow$**:** We prove the contraposition of this implication. If there exists $xs :: [\tau]$ such that $f_{Int}\ (shape\ xs) \not\sqsupseteq g_{Int}\ (shape\ xs)$, then there exists a list $is :: [Int]$, namely, $is = shape\ xs$, such that $f_{Int}\ is \not\sqsupseteq g_{Int}\ is$. Thus, we have $f \not\preceq g$ by the definition of the less-strict relation for polymorphic functions (Definition 6.1.2). □

6.4 Less Strict Functions in the Presence of seq

In this section we consider a language that provides a strict evaluation primitive *seq*. Although Theorem 6.3.1, proved in the absence of *seq*, seems quite natural, the statement breaks when *seq* is available. In the presence of *seq* the property $f_{()} \preceq g_{()}$ does not imply $f_\tau \preceq g_\tau$ for all types τ. For example, consider the following functions of type $[\alpha] \to [\alpha]$.

$$f :: [\alpha] \to [\alpha]$$
$$f\ (x : y : \_) = [seq\ x\ y]$$

$$g :: [\alpha] \to [\alpha]$$
$$g\ (x : y : \_) = [seq\ y\ x]$$

If one of the first two elements of the argument list is $\bot$, then both functions yield $[\bot]$. If both elements are $()$, then both functions yield $[()]$. That is, we have $f_{()} \preceq g_{()}$, or, more precisely, we even have $f_{()} \equiv g_{()}$. However, there are obviously types τ such that f_τ and g_τ are incomparable. For example, we have $f_{Bool}\ [False, True] \equiv [False]$ but $g_{Bool}\ [False, True] \equiv [True]$. This example shows that Theorem 6.3.1 fails in the presence of *seq*.

So, how about Theorem 6.3.2, does it break as well? It holds for the previous example, but it breaks for the following functions.

$$f' :: [\alpha] \to [\alpha]$$
$$f'\ (x : y : z : \_) = [seq\ x\ z]$$

$$g' :: [\alpha] \to [\alpha]$$
$$g'\ (x : y : z : \_) = [seq\ y\ z]$$

The functions f' and g' act identically on shapes but are incomparable with respect to the less-strict relation. Hence, Theorem 6.3.2 is also not valid if we consider *seq*.

In contrast to Theorem 6.3.1 the assertion of Theorem 6.3.2 directly depends on the notion of shape. Thus, maybe we can prove a similar statement when we alter *shape*. Let us track back where the current proof is corrupted by *seq*. We start with the proof of Lemma 6.3.4.

Lemma 6.3.3 is still valid because none of the considered functions uses *seq*. In this case we are quasi considering a language without *seq*. In contrast, Lemma 6.3.4 considers an arbitrary function f that might use *seq*. According to Johann and Voigtländer (2004), the step labeled with $(\dagger)$ in the proof of Lemma 6.3.4 might not hold in the presence of *seq* as $(xs!!)$ is not total[5]. In fact, the lemma fails completely. For example, consider the function f again. We have

$$f\ [\bot, True] \equiv [\bot],$$

and, on the other hand, the following holds.

[5] A function f is total if for all x with $x \not\equiv \bot$ we have $f\ x \not\equiv \bot$.

$$
\begin{array}{l}
map\ ([\bot, True]!!)\ (f\ (shape\ [\bot, True])) \\
\equiv \quad \{\ \text{definition of } shape\ \} \\
map\ ([\bot, True]!!)\ (f\ [0,1]) \\
\equiv \quad \{\ \text{definition of } f\ \} \\
map\ ([\bot, True]!!)\ [1] \\
\equiv \quad \{\ \text{definition of } map\ \} \\
[[\bot, True]\ !!\ 1] \\
\equiv \quad \{\ \text{definition of } (!!)\ \} \\
[True]
\end{array}
$$

That is, we have $f\ xs \not\equiv map\ (xs!!)\ (f\ (shape\ xs))$.

The function f is able to distinguish the list $[\bot, True]$ from the list $[0,1]$ by employing *seq*, although these lists have the same shape. Hence, in the presence of *seq* a polymorphic function cannot only distinguish two lists by their shape in the sense of the function *shape*. It can also distinguish them by the occurrences of $\bot$ as list element. Thus, to take *seq* into account we have to consider a different notion of shape, implemented by the function $shape_{seq}$.

$$
\begin{array}{l}
shape_{seq} :: [\alpha] \to [Int] \\
shape_{seq} = zipWith\ (\lambda n\ x \to seq\ x\ n)\ [0\,..]
\end{array}
$$

The function $shape_{seq}$ replaces all elements x of a list with $x \not\equiv \bot$ by their position in the list. However, in contrast to *shape*, $shape_{seq}$ keeps all elements x with $x \equiv \bot$. For example, we have

$$shape_{seq}\ [False, \bot, True] \equiv [0, \bot, 2]$$

while

$$shape\ [False, \bot, True] \equiv [0, 1, 2].$$

In the remaining part of this section we prove that a theorem like Theorem 6.3.2 holds in the presence of *seq* if we replace *shape* by $shape_{seq}$. The proofs in particular highlight the role of free theorems.

If we replace *shape* by $shape_{seq}$ in Lemma 6.3.4 we can provide a characterization of polymorphic functions in the presence of *seq*. To prove the characterization, first we have to ensure that Lemma 6.3.3 also holds for $shape_{seq}$. This is not obvious, because $shape_{seq}$ uses *seq*, and consequently its free theorem is more restrictive. We employ an extension of free theorems by Seidel and Voigtländer (2009) to manage the proof. According to Johann and Voigtländer (2004), in the presence of *seq*, we have to guarantee that certain functions are total. However, the extension by Seidel and Voigtländer (2009) allows for less restrictive free theorems. Given a function definition, it assigns a refined type to the function and uses this refined type to derive less restrictive free theorems.

To generate a free theorem for a specific implementation of a polymorphic function we have to define the function in a simple functional language with a fixpoint operator and a *seq* primitive. Figure 6.4.1 shows the free theorem that is generated

$$\begin{array}{l}\forall f :: \tau_1 \to \tau_2,\ f \textit{ strict and total}.\\ \quad \forall g :: \tau_3 \to \tau_4,\ g \textit{ strict}.\\ \quad\quad \forall xs :: [\tau_3].\ t\ xs \not\equiv \bot \iff t\ (map\ g\ xs) \not\equiv \bot\\ \quad\quad\quad \wedge \forall ys :: [\tau_1].\ map\ g\ (t\ xs\ y) \equiv t\ (map\ g\ xs)\ (map\ f\ ys)\end{array}$$

Figure 6.4.1: The Specialized Free Theorem where $t \equiv zipWith\ (\lambda x\ n \to seq\ x\ n)$.

for $zipWith\ (\lambda n\ x \to seq\ x\ n) :: [\alpha] \to [\beta] \to [\alpha]$.[6] By inspecting the implementation the generator observes that we only use *seq* on the second argument of the function passed to *zipWith*. Therefore, in contrast to the general free theorem, the specialized version does not require g to be total. This enables us to prove the following adjusted version of Lemma 6.3.3.

Lemma 6.4.1: *For all lists* $xs :: [\tau]$ *the following holds.*

$$map\ (xs!!)\ (shape_{seq}\ xs) \equiv xs$$

Proof: We reason the same way as in Lemma 6.3.3 but replace all occurrences of *shape* by $shape_{seq}$ and all occurrences of $(\lambda n\ \_ \to n)$ by $(\lambda n\ x \to seq\ x\ n)$. Instead of the general free theorem for *zipWith* we employ the specialized free theorem for $zipWith\ (\lambda n\ x \to seq\ x\ n)$ shown in Figure 6.4.1. We replace the steps labeled with $(*)$ and $(**)$ in the proof of Lemma 6.3.3 by the following reasoning. We prove the step labeled with $(*)$ by instantiating the specialized free theorem as follows.

$$\begin{array}{ll} g = ((y : ys)!!) & x = [1..]\\ f = id & y = ys\end{array}$$

We prove the step labeled with $(**)$ by instantiating the specialized free theorem as follows.

$$\begin{array}{ll} g = (ys!!) & x = [0..]\\ f = id & y = ys\end{array}$$

Note that *id* is strict and total and that $((y : ys)!!)$ and $(ys!!)$ are strict. □

To complete the proof of an adjusted version of Lemma 6.3.4 with $shape_{seq}$, we still have to recover the step labeled with $(\dagger)$ in the original proof. That is, we want to prove that

$$f\ (map\ (xs!!)\ (shape_{seq}\ xs)) \equiv map\ (xs!!)\ (f\ (shape_{seq}\ xs)).$$

If we consider Johann and Voigtländer (2004), we have to show that $(xs!!)$ is total, which is not the case. First of all the position we are projecting to might be out of bounds, for example, $[1,2]\ !!\ 3$. Furthermore, the list xs may contain $\bot$, which

[6] The generator is available at `www-ps.iai.uni-bonn.de/cgi-bin/polyseq.cgi`.

also breaks totality if we project to the corresponding position, for example, $[1, \bot] !! 1$. We cannot employ the extension by Seidel and Voigtländer (2009) because we do not consider a concrete function like *zipWith* before but an arbitrary function of type $[\alpha] \to [\alpha]$. Thus, we cannot weaken the requirements of a free theorem by providing a concrete implementation. Nevertheless, there is rescue. For a specific list the requirements proposed by Johann and Voigtländer (2004) are unnecessarily strong. For example, for the free theorem

$$map\ g\ (f\ xs) \equiv f\ (map\ g\ xs)$$

they demand that g has to be total. However, it suffices that g is total with respect to the elements of the list xs. The following theorem proves this observation by employing the relational version of the free theorem for the type $[\alpha] \to [\alpha]$.

Theorem 6.4.1: *Let $f :: [\alpha] \to [\alpha]$. For all strict functions $g :: \tau_1 \to \tau_2$ and for all $xs :: [\tau_1]$ with*

$$g\ (xs !! n) \equiv \bot \iff xs !! n \equiv \bot$$

for all $n :: Int$, we have

$$map\ g\ (f_{\tau_1}\ xs) \equiv f_{\tau_2}\ (map\ g\ xs).$$

Proof: Let $xs :: [\tau_1]$. We define a relation that relates all elements of the list xs with the image of g of this element and use the relational free theorem for f to prove the statement. As R_{xs} has to be a continuous relation, we close the relation by relating the suprema of all possible chains.

$$R_{xs} := \{(\bigsqcup_{i \in I}(xs !! i), \bigsqcup_{i \in I} g\ (xs !! i)) \mid \langle xs !! i \rangle_{i \in I} \text{ chain}, I \subseteq Int\}$$

In particular we have $(xs !! i, g\ (xs !! i)) \in R_{xs}$ for all $i :: Int$ as the set $\{i\}$ is a chain.

First we show that R_{xs} is a strict, bottom-reflecting and continuous relation. For details about these concepts consider Definition B.1.2 in the Appendix. By definition of R_{xs} we have $(xs !! \bot, g\ (xs !! \bot)) \in R_{xs}$. Furthermore, we have $xs !! \bot \equiv \bot$ as well as $g\ (xs !! \bot) \equiv g\ \bot \equiv \bot$, that is, R_{xs} is strict.

Next we show that R_{xs} is bottom-reflecting. Let $(x, y) \in R_{xs}$. By definition of R_{xs} we have $x \equiv \bigsqcup_{i \in I}(xs !! i)$ and $y \equiv \bigsqcup_{i \in I} g(xs !! i)$ and reason as follows.

$$\begin{aligned} \bigsqcup_{i \in I}(xs !! i) \equiv \bot &\iff \forall i \in I.\ xs !! i \equiv \bot \\ &\iff \forall i \in I.\ g\ (xs !! i) \equiv \bot \\ &\iff \bigsqcup_{i \in I} g\ (xs !! i) \equiv \bot \end{aligned}$$

Finally, we have to show that R_{xs} is continuous. Let $\langle x_i \rangle_{i \in I}$ and $\langle y_i \rangle_{i \in I}$ be chains whose elements are pair-wise related by R_{xs}. Then, for every index $i \in I$ there is an index-set I_i such that $x_i = \bigsqcup_{j \in I_i}(xs !! j)$ and $y_i = \bigsqcup_{j \in I_i}(g\ (xs !! j))$. In the following,

we use the set $K := \{j \mid i \in I, j \in I_i\}$. We have

$$(\bigsqcup_{k \in K} (xs \mathbin{!!} k), \bigsqcup_{k \in K} g\ (xs \mathbin{!!} k)) = (\bigsqcup_{i \in I} x_i, \bigsqcup_{i \in I} y_i),$$

and, by definition of R_{xs}, the left-hand side is an element of R_{xs}. Thus R_{xs} is continuous.

We have $(xs, map\ g\ xs) \in \text{lift}\{[\,]\}(R_{xs})$ where $\text{lift}\{[\,]\}(R)$ is the relational structural lifting defined as follows.

$$\text{lift}\{[\,]\}(R) = \{(\bot, \bot), ([\,], [\,])\} \cup \{(x : xs, y : ys) \mid (x, y) \in R, (xs, ys) \in \text{lift}\{[\,]\})(R)\}$$

Intuitively $(xs, ys) \in \text{lift}\{[\,]\}(R_{xs})$ means that ys is the point-wise g-image of xs or, more precisely, $ys \equiv map\ g\ xs$. Because the relational free theorem states that $(xs, map\ g\ xs) \in \text{lift}\{[\,]\}(R_{xs})$ implies $(f\ xs, f\ (map\ g\ xs)) \in \text{lift}\{[\,]\}(R_{xs})$, we obtain that $f\ (map\ g\ xs)$ is the point-wise g-image of $f\ xs$, that is, $map\ g\ (f\ xs) \equiv f\ (map\ g\ xs)$. □

For proving statements about functions in the presence of *seq* the previous theorem is quite useful as we are not restricted to total functions anymore. While the previous theorem does only consider functions of type $[\alpha] \to [\alpha]$, by employing the generalization sketched in Section 6.6, we can generalize this theorem to arbitrary first-order type constructors.

Additionally, this kind of generalization of a functional free theorem is also valuable in the absence of *seq*. For example, the standard functional free theorem for a function $f :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ states that

$$h\ (p\ x) \equiv q\ (g\ x)$$

for all x of appropriate type implies

$$map\ h\ (f\ p\ xs) \equiv f\ q\ (map\ g\ xs)$$

for all xs of appropriate type. By employing the relation R_{xs} defined in the previous proof we can derive a free theorem that states for a list xs that

$$h\ (p\ (xs \mathbin{!!} n)) \equiv q\ (g\ (xs \mathbin{!!} n))$$

for all $n :: Int$ already implies

$$map\ h\ (f\ p\ xs) \equiv f\ q\ (map\ g\ xs).$$

That is, we only have to prove the relation between h, p, q, and g for the elements of the list xs and not for all x of appropriate type. Actually, we have used exactly this statement to prove Lemma 6.3.2.

We want to employ Theorem 6.4.1 to prove

$$map\ (xs!!)\ (f\ (shape_{seq}\ xs)) \equiv f\ (map\ (xs!!)\ (shape_{seq}\ xs))$$

to fix step (†) in the proof of Lemma 6.3.4. Thus, we have to show that we have $xs\,!!\,((shape_{seq}\ xs)\,!!\,n) \equiv \bot$ if and only if $(shape_{seq}\ xs)\,!!\,n \equiv \bot$. Therefore, we show a connection between $(!!)$ and *zipWith*, namely, that

$$(zipWith\ f\ xs\ ys)\,!!\,n \equiv f\ (xs\,!!\,n)\ (ys\,!!\,n).$$

This statement is quite similar to the free theorem for $(!!)$, which states that

$$(map\ f\ xs)\,!!\,n \equiv f\ (xs\,!!\,n).$$

Indeed, these statements are closely related. In the context of free theorems, mostly, only two-ary logical relations are considered. The connection between *zipWith* and $(!!)$ is the functional instantiation of the relational free theorem for $(!!)$ if we consider three-ary relations. Alternatively, we can prove the following lemma by means of structural induction.

Lemma 6.4.2: *For all strict*[7] *functions* $f :: \tau_1 \to \tau_2 \to \tau_3$ *and all lists* $xs :: [\tau_1]$, $ys :: [\tau_2]$ *the following holds.*

$$(zipWith\ f\ xs\ ys)\,!!\,n \equiv f\ (xs\,!!\,n)\ (ys\,!!\,n)$$

By employing the previous lemma we prove a characteristic property of $shape_{seq}$, which is not valid for *shape*. The shape of a list with respect to $shape_{seq}$ is $\bot$ at a certain position if and only if the "original list" is $\bot$ at that position.

Lemma 6.4.3: *For all lists* $xs :: [\tau]$ *and all* $n ::$ *Int we have*

$$(shape_{seq}\ xs)\,!!\,n \equiv \bot \iff xs\,!!\,n \equiv \bot.$$

Proof: We start with the following calculation.

$$\begin{array}{ll}
& (shape_{seq}\ xs)\,!!\,n \\
\equiv & \quad \{\ \text{definition of } shape_{seq}\ \} \\
& (zipWith\ (\lambda n\ x \to seq\ x\ n)\ [0\,..]\ xs)\,!!\,n \\
\equiv & \quad \{\ \text{Lemma 6.4.2, } seq \text{ strict}\ \} \\
& seq\ (xs\,!!\,n)\ ([0\,..]\,!!\,n)
\end{array}$$

If we have $n \equiv \bot$ or $n < 0$, then $seq\ (xs\,!!\,n)\ ([0\,..]\,!!\,n) \equiv seq\ \bot\ \bot \equiv \bot$ as well as $xs\,!!\,n \equiv \bot$. If we have $n >= 0$, then $seq\ (xs\,!!\,n)\ ([0\,..]\,!!\,n) \equiv seq\ (xs\,!!\,n)\ n$. That is, $(shape_{seq}\ xs)\,!!\,n \equiv \bot$ if and only if $xs\,!!\,n \equiv \bot$. □

Employing the characterization of $shape_{seq}$ we prove the analogue to Lemma 6.3.4 in the presence of *seq*.

Lemma 6.4.4: *For all functions* $f :: [\alpha] \to [\alpha]$ *and all lists* $xs :: [\tau]$ *we have the following.*

$$f_\tau\ xs \equiv map\ (xs!!)\ (f_{Int}\ (shape_{seq}\ xs))$$

[7] Here strict means $f\ x\ \bot \equiv \bot$ and $f\ \bot\ y \equiv \bot$ for all x, y.

Proof: We reason as follows.

$$
\begin{array}{ll}
& f_\tau\ xs \\
\equiv & \{\ \text{Lemma 6.4.1}\ \} \\
& f_\tau\ (map\ (xs!!)\ (shape_{seq}\ xs)) \\
\equiv & \{\ \text{Theorem 6.4.1, Lemma 6.4.3}\ \} \\
& map\ (xs!!)\ (f_{Int}\ (shape_{seq}\ xs))
\end{array}
$$

This proof is very similar to the proof of Lemma 6.3.4, but instead of a general free theorem we use the specialized version proved in Theorem 6.4.1. □

Now we have the means at hand to prove the analogue of Theorem 6.3.2 in the presence of *seq*. The proof of this statement is analogous to the proof of Theorem 6.3.2, but instead of Lemma 6.3.4 we employ Lemma 6.4.4.

Theorem 6.4.2: *For all* $f, g :: [\alpha] \to [\alpha]$ *the following holds.*

$$f_{Int} \circ shape_{seq} \preceq g_{Int} \circ shape_{seq} \Longleftrightarrow f \preceq g$$

The previous lemma states that we can check whether a function f is less strict than a function g in the presence of *seq* by checking them for all images of $shape_{seq}$. In fact, we have observed that, in this setting, we probably can reduce the number of test cases significantly. More precisely, in the following we illustrate that it is possible to consider linearly many test cases in the length of a list instead of exponentially many test cases in the length of the list to distinguish two functions of type $[\alpha] \to [\alpha]$ in the presence of *seq*.

Let us consider a function of type $\alpha \to \alpha \to \alpha \to \alpha$ in a language without *seq*. By employing free theorems we can show that every function $f :: \alpha \to \alpha \to \alpha \to \alpha$ is semantically equivalent to one of the following functions.

$$
\begin{array}{l}
f_1 :: \alpha \to \alpha \to \alpha \to \alpha \\
f_1\ x\ y\ z = \bot
\end{array}
$$

$$
\begin{array}{l}
f_2 :: \alpha \to \alpha \to \alpha \to \alpha \\
f_2\ x\ y\ z = x
\end{array}
$$

$$
\begin{array}{l}
f_3 :: \alpha \to \alpha \to \alpha \to \alpha \\
f_3\ x\ y\ z = y
\end{array}
$$

$$
\begin{array}{l}
f_4 :: \alpha \to \alpha \to \alpha \to \alpha \\
f_4\ x\ y\ z = z
\end{array}
$$

This observation can also be considered as the basis for the validity of free theorems. In other words, for a function with a polymorphic type there are only a couple of semantically distinguishable implementations. And, therefore, we can check whether

two functions of type $[\alpha] \to [\alpha]$ are equal by only checking them for a small number of arguments, namely, all images of *shape*.

We can regard a function of type $\alpha \to \alpha \to \alpha \to \alpha$ as a function of type $[\alpha] \to [\alpha]$ that takes lists with exactly three elements and yields lists with exactly one element. As Theorem 6.3.2 states, two functions of this kind can be distinguish by checking their behavior for the argument $[0, 1, 2]$. In other words, to check whether two functions of type $\alpha \to \alpha \to \alpha \to \alpha$ are equal we only have to apply them to the arguments 0, 1, and 2. If we consider the definitions above, this becomes quite obvious. That is, for a function of type $f :: \alpha \to \alpha \to \alpha \to \alpha$ we can check whether it is equivalent to f_1, f_2, f_3, or f_4 by checking whether the application $f\ 0\ 1\ 2$ yields $\bot$, 0, 1, or 2.

Now let us consider a function of type $\alpha \to \alpha \to \alpha \to \alpha$ in the presence of *seq*. If we consider a function $f :: \alpha \to \alpha \to \alpha \to \alpha$, this function has to be sequential if we consider a sequential language. However, free theorems do not consider sequentiality. For example, by the corresponding free theorem, f might satisfy $f\ x\ \bot\ \bot \equiv x$, $f\ \bot\ x\ \bot \equiv x$, and $f\ \bot\ \bot\ x \equiv x$ for $x \not\equiv \bot$. Though, the function f would not be sequential as no argument is a sequential position.

In a sequential language we have either $f\ \bot\ y\ z \equiv \bot$, $f\ x\ \bot\ z \equiv \bot$, or $f\ x\ y\ \bot \equiv \bot$ for all x, y, and z. By sequentiality at least two of the applications $f\ x\ \bot\ \bot$, $f\ \bot\ x\ \bot$, or $f\ \bot\ \bot\ x$ yield $\bot$ for all x as one of the arguments is a sequential position. Without loss of generality we assume that there exists an x such that the application $f\ x\ \bot\ \bot$ does not yield $\bot$. A free theorem states that we have $f\ x\ \bot\ \bot \equiv x$ in this case. Therefore, regarding a function $f :: \alpha \to \alpha \to \alpha \to \alpha$, the behavior of the applications $f\ x\ \bot\ \bot$, $f\ \bot\ x\ \bot$, and $f\ \bot\ \bot\ x$ for all x is already determined by sequentiality. Thus, we do not have to consider the corresponding test cases.

If we consider a function of type $\alpha \to \alpha \to \alpha \to \alpha$ as a function of type $[\alpha] \to [\alpha]$, again, we would check it for the images of $shape_{seq}$, that is, for the lists

$$[\bot, \bot, \bot],\ [0, \bot, \bot],\ [\bot, 1, \bot],\ [\bot, \bot, 2],\ [\bot, 1, 2],\ [0, \bot, 2],\ [0, 1, \bot],\ \text{and}\ [0, 1, 2].$$

However, there are no two functions of type $\alpha \to \alpha \to \alpha \to \alpha$ that can be distinguished by the test cases $f\ \bot\ \bot\ \bot$, $f\ 0\ \bot\ \bot$, $f\ \bot\ 1\ \bot$, or $f\ \bot\ \bot\ 2$ and cannot be distinguished by one of the other test cases. Thus, using the test cases $f\ 0\ 1\ \bot$, $f\ 0\ \bot\ 2$, $f\ \bot\ 1\ 2$, and $f\ 0\ 1\ 2$ we can already distinguish all functions of type $\alpha \to \alpha \to \alpha \to \alpha$.

We assume that we can generalize this observation from functions of type $\alpha \to \alpha \to \alpha \to \alpha$ to functions on type $f :: [\alpha] \to [\alpha]$ in a setting with *seq* primitive. Instead of checking functions of type $[\alpha] \to [\alpha]$ for all images of $shape_{seq}$ we can reduce the number of test cases. We have to check functions of type $[\alpha] \to [\alpha]$ only for all shapes where we replace one element of the list with $\bot$. That is, we do not have to consider test cases like $[0, \bot, \bot]$ or $\bot : 1 : \bot : 2 : \bot$. Note, that this reduces the number of test cases from exponentially many in the length of the list to linearly many in the length of the list. However, note that this observation does not hold in the presence of a feature like $unamb :: \alpha \to \alpha \to \alpha$ (Elliott 2009) as it allows to define non-sequential functions. Refining the concept of a shape in the presence of *seq* as illustrated above is future work. More precisely, we would have to incorporate sequentiality into the logical relation that is the basis for free theorems.

6.5 Minimally Strict Functions on Lists

We return to a setting without *seq* to consider minimally strict functions. If we reconsider Theorem 6.3.2, one might assume that we can check whether a polymorphic function $f :: [\alpha] \to [\alpha]$ is minimally strict by checking whether there exists a function $h :: [Int] \to [Int]$ that is less strict than f_{Int} with respect to all shapes. Consider the following function of type $[\alpha] \to [\alpha]$ that yields a singleton list that contains the first element of its argument list.

$$f :: [\alpha] \to [\alpha]$$
$$f\ (x{:}\ \_) = [x]$$

The following function $h :: [Int] \to [Int]$ is less strict than f_{Int} with respect to all shapes as we have $h\ \bot \equiv [0] \sqsupset \bot \equiv f_{Int}\ \bot$, for example.

$$h :: [Int] \to [Int]$$
$$h\ \_ = [0]$$

Though, obviously there exists no polymorphic function whose integer instance behaves like h. The function h "invents" a list element, namely, 0. As Lemma 6.3.5 tells us, the elements of the result list of a polymorphic function of type $[\alpha] \to [\alpha]$ are taken from the argument list or they are $\bot$. To prevent the monomorphic function from inventing elements we do not compare $f_{Int}\ xs$ with $h\ xs$ but with $map\ (xs!!)\ (h\ xs)$. In the example above, $h\ \bot$ is more defined than $f_{Int}\ \bot$, but $map\ (\bot!!)\ (h\ \bot)$ is as defined as $f_{Int}\ \bot$. Therefore, h is not a witness that shows that f is unnecessarily strict.

If we consider a less-strict relation that only relates functions if they agree for total arguments, then h is not less strict than f_{Int} as we have $h\ [\,] \equiv [0]$, but $f_{Int}\ [\,] \equiv \bot$. Note that we cannot resolve this issue by adding a rule for the empty list to f as it would have to invent a list element. However, even in a setting that only relates functions by the less-strict relation if they agree for total arguments, we can define a similar example by considering the following data type of non-empty lists.

$$\textbf{data}\ NonEmpty\ \alpha = NonEmpty\ \alpha\ [\alpha]$$

We have to define a notion of *shape* and indexing for the data type *NonEmpty*, which is straightforward. Then, we can define a polymorphic function similar to f and a monomorphic function similar to h such that h is less strict than f_{Int} but f is still minimally strict. Furthermore, h agrees with f_{Int} for all total arguments.

Theorem 6.5.1: *Let $f :: [\alpha] \to [\alpha]$. The function f is not minimally strict if and only if there exists a function $h :: [Int] \to [Int]$ such that*

$$g \prec f$$

where g is defined as follows.

$$g :: [\alpha] \to [\alpha]$$
$$g\ xs = map\ (xs!!)\ (h\ (shape\ xs))$$

Proof: $\Longrightarrow$**:** As f is not minimally strict, there exists a polymorphic function $g :: [\alpha] \to [\alpha]$ that is less strict than f. By Lemma 6.3.4 we have

$$g\ xs \equiv map\ (xs!!)\ (g_{Int}\ (shape\ xs))$$

and take g_{Int} as h.

$\Longleftarrow$**:** The function f is not minimally strict since $g :: [\alpha] \to [\alpha]$ is less strict than f. □

We can prove a similar theorem in the presence of *seq* by employing $shape_{seq}$ instead of *shape*.

Sloth uses a slightly different approach to check whether a polymorphic function is minimally strict. Instead of considering a polymorphic function g as defined in Theorem 6.5.1, with respect to a function $h :: [Int] \to [Int]$ Sloth considers a monomorphic function $g\ xs = map\ (xs!!)\ (h\ xs)$. To check whether a polymorphic function f is minimally strict we check whether we have $g \circ shape \prec f_{Int} \circ shape$. It is easy to prove this refined statement by Theorem 6.5.1 and Theorem 6.3.2.

By employing the standard approach for monomorphic functions we can check whether for a polymorphic function $f :: [\alpha] \to [\alpha]$ there exists a function $h :: [Int] \to [Int]$ that is less strict than f_{Int} with respect to some shape. To consider shapes only, the test case generator for the type $[A]$ simply generates list test cases as usually but replaces occurrences of elements of type A by the index of the element in the list. After we have discovered a shape $ys :: [Int]$ such that $h\ ys \sqsupset f\ ys$, we check whether we also have $map\ (ys!!)\ (h\ ys) \sqsupset f\ ys$. If this is the case, then ys is actually a counter-example, and we propose $map\ (ys!!)\ (h\ ys)$ as result of f for the argument ys. Sloth presents the counter-example ys by substituting indices by unique names.

Note that we cannot check whether a polymorphic function that uses *seq* is minimally strict by using the A-instance as we use the notion of shape as defined by *shape* and not by $shape_{seq}$.

6.6 Generalization

In this section we consider polymorphic functions that do not have the type $[\alpha] \to [\alpha]$. For example, consider the following data type for binary trees.

$$\textbf{data}\ Tree\ \alpha = Empty \mid Node\ (Tree\ \alpha)\ \alpha\ (Tree\ \alpha)$$

We define a function *breadthFirst* that enumerates the elements of a tree in breath-first order. The function *partition* takes a list of values and a list of subtrees and adds the components of a *Node* to the corresponding lists.

$$\begin{aligned}
&partition :: Tree\ \alpha \to ([\alpha], [Tree\ \alpha]) \to ([\alpha], [Tree\ \alpha])\\
&partition\ Empty\ \ \ \ level\ \ \ \ = level\\
&partition\ (Node\ l\ v\ r)\ (vs, trees) = (v : vs, l : r : trees)
\end{aligned}$$

The function *breadthLevel* takes one level of a tree, splits the level into values and the next level by employing *partition* and concatenates the values of the current level with the recursive result.

$$
\begin{aligned}
&\mathit{breadthLevel} :: [\mathit{Tree}\ \alpha] \to [\alpha] \\
&\mathit{breadthLevel}\ [\,] \quad = [\,] \\
&\mathit{breadthLevel}\ \mathit{level} = \mathit{values} +\!\!+ \mathit{breadthLevel}\ \mathit{nextLevel} \\
&\quad \textbf{where} \\
&\qquad (\mathit{values}, \mathit{nextLevel}) = \mathit{foldr}\ \mathit{partition}\ ([\,],[\,])\ \mathit{level}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{breadthFirst} :: \mathit{Tree}\ \alpha \to [\alpha] \\
&\mathit{breadthFirst}\ \mathit{tree} = \mathit{breadthLevel}\ [\mathit{tree}]
\end{aligned}
$$

We can check whether *breadthFirst* is minimally strict by instantiating all type variables with the data type A.

```
> strictCheck (breadthFirst :: Tree A -> [A]) 3
5: \(Node (Node ⊥ a ⊥) b ⊥) -> b:a:⊥
Finished 9 tests.
```

To investigate why *breadthFirst* is unnecessarily strict we check whether *partition* is minimally strict. Sloth reports the following counter-examples.

```
> strictCheck
      (partition :: Tree A -> ([A],[Tree A]) -> ([A],[Tree A])) 3
1: \⊥ ⊥ -> (⊥,⊥)
2: \Empty ⊥ -> (⊥,⊥)
3: \(Node ⊥ a ⊥) ⊥ -> (a:⊥,⊥:⊥:⊥)
Finished 5 tests.
```

The first counter-example states that *partition* applied to $\bot$ and $\bot$ yields $\bot$ while there exists is less strict implementation that yields $(\bot, \bot)$ instead. And, even if the first argument is not $\bot$ the function still yields $\bot$ as result. As *partition* performs pattern matching on the tuple in the second argument, as long as the second argument is $\bot$, the result is $\bot$ as well. Although this looks like a minor problem and will not lead to bad performance in most cases, the consequences for *breadthFirst* are heavy. The implementation presented above does not yield any value of a certain level if any of the trees on the same level is $\bot$. In other words, to yield the first value of a level we have to evaluate all trees of this level to head normal form. Note that these are exponentially many trees in the depth of the tree on a particular level .

In order to improve the implementation we simply have to replace the tuple pattern matching in *partition* by a lazy pattern matching. This minor change has a quite significant effect on the runtime of the function. Furthermore, this performance gain increases with larger tree sizes.

To illustrate the benefit of the less strict implementation we perform a breadth first traversal of a complete binary tree and calculate the length of the resulting list. The following table presents the run-times for the presented implementation of *breadthFirst* and a less strict implementation with a lazy pattern matching, called $\mathit{breadthFirst}'$.

depth	*breadthFirst*	*breadthFirst′*
20	2.61s	0.78s
21	8.67s	1.65s
22	30.91s	3.16s

These times were measured with an initial stack size of 50 megabytes as the unnecessarily strict implementation runs out of stack otherwise.

To test whether *breadthFirst* is minimally strict, Sloth uses a generalization of the results presented in the previous sections. More precisely, Sloth, for example, only uses the following test cases to check *breadthFirst* for trees up to size three.

$$\bot,\ \mathit{Empty},\ \mathit{Node}\ \bot\ a\ \bot,\ \mathit{Node}\ \mathit{Empty}\ a\ \bot,\ \mathit{Node}\ (\mathit{Node}\ \bot\ a\ \bot)\ b\ \bot$$

The values a and b can be considered as integers that uniquely identify the position of these values in the term. Sloth does not check the test cases $\mathit{Node}\ \bot\ a\ \mathit{Empty}$ as well as $\mathit{Node}\ \bot\ a\ (\mathit{Node}\ \bot\ b\ \bot)$ because the last argument of *Node* is not a sequential position in $\mathit{Node}\ \bot\ a\ \bot$ with respect to *breadthFirst* and, therefore, these values are not elements of the characteristic set. In this section we sketch the idea that allows us to reduce the number of test case for *breadthFirst* to a linear number of test cases in the number of elements in the tree. More details about this approach and the corresponding proofs for the generalization can be found in (Christiansen 2011).

Instead of considering functions of type $[\alpha] \to [\alpha]$ only, we consider functions of type $\varphi\ \alpha \to \psi\ \alpha$ where φ and ψ are functors that are isomorphic to a functor composed of the unit, the identity, the constant, the product, and the sum functor. In the following, we refer to functors constructed from these building blocks as generic functors. The approach to generic functors is very similar to the approach by Magalhães et al. (2010). For arbitrary functors φ the user has to provide instances of overloaded functions *from* and *to*, which constitute an embedding projection pair. The function *from* transforms a value of type $\varphi\ \tau$ into a value of type $\mathit{Gen}\ \varphi\ \tau$, where *Gen* is a function on the type level (Chakravarty et al. 2005) that maps a type constructor to the isomorphic generic functor. In other words, $\mathit{Gen}\ \varphi$ denotes a functor that is a generic representation of φ. In the same way, the function *to* maps generic values, that is, values of type $\mathit{Gen}\ \varphi\ \tau$ to elements of the original type, namely, elements of type $\varphi\ \tau$.

To generalize the statements from the previous sections we have to provide generalizations of *map*, (!!), and *shape* for arbitrary first-order functors. Therefore, we define instances of the type class *Functor* for all generic functors. Furthermore, we define two type classes that provide functions called *gproj* and *gshape* and define instances of these type classes for all generic functors. By means of a structural induction over the structure of generic functors we can then prove the following statement.

Lemma 6.6.1: *For all generic functors φ and all $x :: \varphi\ \tau$ the following holds.*

$$\mathit{fmap}\ (\mathit{gproj}\ x)\ (\mathit{gshape}\ x) \equiv x$$

By employing the functions *from* and *to* and the instances of *fmap*, *gproj*, and *gshape* for generic functors we get instances of the functions for all functors that can be

represented by generic functors. This way we can easily generalize the previous lemma to all functors that provide an embedding projection pair *from* and *to*. To generalize all the other statements from functions of type $[\alpha] \to [\alpha]$ to functions of type $\varphi\ \alpha \to \psi\ \alpha$ we basically have to replace occurrences of *map*, $(!!)$, and *shape* in the proofs by *fmap*, *gproj*, and *gshape*.

7 Case Studies

In this chapter we present several case studies of applications of Sloth. Before we apply Sloth to check whether several functions are minimally strict, in Section 7.1 we show how we can, in some cases, derive a less strict implementation. In Section 3.2 we have already used this technique to derive a less strict implementation of *intersperse*, and in Section 6.1 we have used this technique to derive a less strict implementation of *inits*.

When we have the means to derive less strict implementations at hand, in Section 7.2 we check whether the standard implementation of the multiplication of Peano numbers is minimally strict. This function is a simple monomorphic example of an unnecessarily strict function that furthermore demonstrates that we can benefit from less strict function definitions if we consider infinite data structures.

In Section 7.3 we consider an implementation of binary numbers as algebraic data type by Braßel et al. (2008). This example is of particular interest as it demonstrates that minimally strict functions provide additional benefits in a functional logic programming language like Curry (Hanus 2006).

In Section 7.4 we consider a somehow more practical example. We examine the implementation of a function from the Haskell library split, which provides a variety of functions to split a list into sublists.

Finally, in Section 7.5 we show that minmal strictness cannot be put on a level with low memory footprint or even fast execution. More precisely, we consider a function that reverses a list and show that the benefit of a minimally strict implementation highly depends on the context of an application of a minimally strict function.

7.1 Deriving a Less Strict Implementation

As we have observed in Section 3.2, in some cases we can derive a less strict implementation of a function by distributing a context that occurs on all right-hand sides of a **case** expression over the **case** expression. That is, we consider a function f of the following form where C and D are contexts.

$$\begin{aligned} f\ x_1 \ldots x_n = C[\ &\textbf{case}\ e\ \textbf{of} \\ &\quad p_1 \ \to D[e_1] \\ &\quad \ldots \\ &\quad p_m \to D[e_m]] \end{aligned}$$

Here, a context is an expression with the additional symbol $[\cdot]$, called hole. By $C[e]$ we denote the substitution of the hole by the expression e.

If the variables of the patterns p_1 to p_m do not occur free in D, we can transform the function f into a function g.

$$g\ x_1 \ldots x_n = C[D[\ \textbf{case}\ e\ \textbf{of}\ \begin{array}{l} p_1 \rightarrow e_1 \\ \quad \ldots \\ p_m \rightarrow e_m \end{array}]]$$

In the following we refer to the transformation from the former into the latter form as **case** *deferment*.

The function g is at least as little strict as the function f. Though, in general, g is not necessarily less strict than f. To determine the conditions to get a less strict implementation by **case** deferment, we consider this transformation in the simple functional programming language introduced in Section 2.2. As this programming language does not provide arbitrary algebraic data types, we examine the special case of a **case** expression that matches lists. That is, we have a function of the following form where y and ys do not occur free in the context D.

$$f\ x = C[\ \textbf{case}\ e\ \textbf{of}\ \begin{array}{ll} \mathit{Nil} & \rightarrow D[e_1] \\ \mathit{Cons}\langle y, ys\rangle & \rightarrow D[e_2] \end{array}]$$

We calculate the semantics of f with respect to the semantics defined in Figure 2.2.4. The semantics of a context C, denoted by $[\![C]\!]_a$ where a is an environment, is a function that takes a semantic value v, fills the hole in the context with a fresh variable x and updates the environment such that occurrences of x are replaced by v. More precisely, we define

$$[\![C]\!]_a = \lambda v.[\![C[x]]\!]_{a[x \mapsto v]}.$$

To calculate the semantics of f we use the fact that the semantics is compositional, in other words, we use the equation

$$[\![C[e]]\!]_a = [\![C]\!]_a\ [\![e]\!]_a.$$

In the following we abbreviate the environment $a[x \mapsto v]$ to a' and calculate the semantics of f.

$$\begin{aligned}
[\![f]\!]_a &= \lambda v.[\![C[\ \textbf{case}\ e\ \textbf{of}\ \{\mathit{Nil} \rightarrow D[e_1]; \mathit{Cons}\langle y, ys\rangle \rightarrow D[e_2]\}]]\!]_{a'} \\
&= \lambda v.[\![C]\!]_{a'} \begin{cases} \bot & \text{if } [\![e]\!]_{a'} = \bot \\ [\![D[e_1]]\!]_{a'} & \text{if } [\![e]\!]_{a'} = \mathit{Nil} \\ [\![D[e_2]]\!]_{a'[y \mapsto v_1, ys \mapsto v_2]} & \text{if } [\![e]\!]_{a'} = \mathit{Cons}\ \langle v_1, v_2\rangle \end{cases} \\
&= \lambda v.[\![C]\!]_{a'} \begin{cases} \bot & \text{if } [\![e]\!]_{a'} = \bot \\ [\![D]\!]_{a'}\ [\![e_1]\!]_{a'} & \text{if } [\![e]\!]_{a'} = \mathit{Nil} \\ [\![D]\!]_{a'[y \mapsto v_1, ys \mapsto v]}\ [\![e_2]\!]_{a'[y \mapsto v_1, ys \mapsto v_2]} & \text{if } [\![e]\!]_{a'} = \mathit{Cons}\ \langle v_1, v_2\rangle \end{cases} \\
&= \lambda v.[\![C]\!]_{a'} \begin{cases} \bot & \text{if } [\![e]\!]_{a'} = \bot \\ [\![D]\!]_{a'}\ [\![e_1]\!]_{a'} & \text{if } [\![e]\!]_{a'} = \mathit{Nil} \\ [\![D]\!]_{a'}\ [\![e_2]\!]_{a'[y \mapsto v_1, ys \mapsto v_2]} & \text{if } [\![e]\!]_{a'} = \mathit{Cons}\ \langle v_1, v_2\rangle \end{cases}
\end{aligned}$$

$$= \lambda v. \begin{cases} [\![C]\!]_{a'}\ \bot & \text{if } [\![e]\!]_{a'} = \bot \\ [\![C]\!]_{a'}\ ([\![D]\!]_{a'}\ [\![e_1]\!]_{a'}) & \text{if } [\![e]\!]_{a'} = Nil \\ [\![C]\!]_{a'}\ ([\![D]\!]_{a'}\ [\![e_2]\!]_{a'[y \mapsto v_1, ys \mapsto v_2]}) & \text{if } [\![e]\!]_{a'} = Cons\ \langle v_1, v_2 \rangle \end{cases}$$

The last but one step of the calculation is valid because we only consider contexts D that do not contain free occurrences of y and ys. In the same way we calculate the semantics of the corresponding function g.

$$\begin{aligned} [\![g]\!]_a &= \lambda v. [\![C[D[\ \textbf{case}\ e\ \textbf{of}\ \{Nil \to e_1; Cons\langle y, ys\rangle \to e_2\}]]]\!]_{a'} \\ &= \lambda v. [\![C]\!]_{a'} \left([\![D]\!]_{a'} \begin{cases} \bot & \text{if } [\![e]\!]_{a'} = \bot \\ [\![e_1]\!]_{a'} & \text{if } [\![e]\!]_{a'} = Nil \\ [\![e_2]\!]_{a'[y \mapsto v_1, ys \mapsto v_2]} & \text{if } [\![e]\!]_{a'} = Cons\ \langle v_1, v_2 \rangle \end{cases} \right) \\ &= \lambda v. \begin{cases} [\![C]\!]_{a'}\ ([\![D]\!]_{a'}\ \bot) & \text{if } [\![e]\!]_{a'} = \bot \\ [\![C]\!]_{a'}\ ([\![D]\!]_{a'}\ [\![e_1]\!]_{a'}) & \text{if } [\![e]\!]_{a'} = Nil \\ [\![C]\!]_{a'}\ ([\![D]\!]_{a'}\ [\![e_2]\!]_{a'[y \mapsto v_1, ys \mapsto v_2]}) & \text{if } [\![e]\!]_{a'} = Cons\ \langle v_1, v_2 \rangle \end{cases} \end{aligned}$$

Thus, if we consider functions g and f of type $\tau_1 \to \tau_2$ we have $[\![g]\!] \sqsupset [\![f]\!]$ if there exists $v \in [\![\tau_1]\!]$ such that

$$[\![e]\!]_{[x \mapsto v]} = \bot$$

and

$$[\![C]\!]_{[x \mapsto v]}\ ([\![D]\!]_{[x \mapsto v]}\ \bot) \sqsupset [\![C]\!]_{[x \mapsto v]}\ \bot.$$

As **case** deferment is a purely syntactical transformation, a compiler could apply this transformation automatically by inspecting the right-hand sides of a **case** expression. In some cases, we get a less strict implementation by applying this simple transformation. However, even if we do not get a less strict function, the resulting function is as little strict as the original one. Furthermore, Gustavsson and Sands (1999) show that **case** deferment preserves heap usage and improves stack usage if D is a context that evaluates the hole to head normal form if the context itself is evaluated to head normal form. They refer to this kind of contexts as reduction contexts. For all reduction contexts D we have $[\![D]\!]_{[x \mapsto v]}\ \bot = \bot$ and, therefore, $[\![C]\!]_{[x \mapsto v]}\ ([\![D]\!]_{[x \mapsto v]}\ \bot) = [\![C]\!]_{[x \mapsto v]}\ \bot$. Thus, they consider a form of **case** deferment that does not yield a function that less strict. Note that it is not a necessary condition for $[\![C]\!]_{[x \mapsto v]}\ ([\![D]\!]_{[x \mapsto v]}\ \bot) = [\![C]\!]_{[x \mapsto v]}\ \bot$ that D is a reduction context. They do not consider the more general transformation presented here as it is not semantics preserving. We did not check whether, using the theory by Gustavsson and Sands (1999, 2001), we can prove that **case** deferment never yields a function that consumes more memory than the original implementation. However, a syntactical approach to less strict implementations is certainly a direction for future considerations.

In the following sections, we will see some examples of functions that can be improved with respect to strictness by applying **case** deferment. However, in most cases, we have to apply other transformations first before we can apply **case** deferment. Furthermore, we will as well see several examples where **case** deferment

cannot be used to derive a less strict implementation. In these cases we have to alter the order of pattern matching.

The following lemma summarizes the observations so far, but, in contrast to the considerations above, we do not use our simple functional language but full Haskell and arbitrary algebraic data types instead of lists. Furthermore, we use an informal notation where we do not separate syntax and semantics in the same line as we do in Chapter 6. We do not present a proof for this statement as we would have to define a semantics that provides arbitrary algebraic data types.

Lemma 7.1.1: *Let $f, g :: \tau_1 \to \cdots \to \tau_n \to \tau$ be functions defined as follows where C and D are contexts of appropriate types, $p_1, \ldots, p_m$ are patterns of appropriate types, and $e_1, \ldots, e_m$ and e are expressions of appropriate types. Furthermore, the variables of $p_1, \ldots, p_m$ do not occur free in the context D.*

$$
\begin{aligned}
f\ x_1 \ldots x_n = C[\ &\textbf{case}\ e\ \textbf{of} \\
&\quad p_1 \ \to D[e_1] \\
&\quad\ \ldots \\
&\quad p_m \to D[e_m]]
\end{aligned}
$$

$$
\begin{aligned}
g\ x_1 \ldots x_n = C[D[\ &\textbf{case}\ e\ \textbf{of} \\
&\quad p_1 \ \to e_1 \\
&\quad\ \ldots \\
&\quad p_m \to e_m]]
\end{aligned}
$$

We have $g \sqsupset f$ if and only if there exist arguments $v_1 :: \tau_1, \ldots, v_n :: \tau_n$ such that

$$e_{[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]} \equiv \bot$$

and

$$D[C[\bot]]_{[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]} \sqsupset D[\bot]_{[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]}.$$

Here, $e_{[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]}$ denotes the semantics of e with respect to the given substitutions.

Let us consider two examples of functions that do not become less strict by applying **case** deferment. Consider the following functions of type $\tau \to [\tau]$.

$$
\begin{aligned}
f\ x = \ &\textbf{case}\ [\,]\ \textbf{of} \\
&\quad p_1 \ \to x : e_1 \\
&\quad\ \ldots \\
&\quad p_m \to x : e_m
\end{aligned}
$$

$$
\begin{aligned}
g\ x = x : \ &\textbf{case}\ [\,]\ \textbf{of} \\
&\quad p_1 \ \to e_1 \\
&\quad\ \ldots \\
&\quad p_m \to e_m
\end{aligned}
$$

As we have $[\,] \not\equiv \bot$ for all arguments $v :: \tau$, we get $g \equiv f$ by Lemma 7.1.1. As another example, consider the following functions of type $\tau \to [\tau]$.

$$
\begin{aligned}
f\ x = tail\ (&\textbf{case}\ e\ \textbf{of}\\
&\quad p_1\ \to x : e_1\\
&\quad \ldots\\
&\quad p_m \to x : e_m)
\end{aligned}
$$

$$
\begin{aligned}
g\ x = tail\ (x : &\textbf{case}\ e\ \textbf{of}\\
&\quad p_1\ \to e_1\\
&\quad \ldots\\
&\quad p_m \to e_m)
\end{aligned}
$$

As we have $tail\ (v : \bot) \equiv \bot$ for all arguments $v :: \tau$, we get $g \equiv f$ by Lemma 7.1.1. Note that this is a case where the context D is not a reduction context as considered by Gustavsson and Sands (1999) because $x : [\cdot]$ does not evaluate its hole to head normal form if it is evaluated to head normal form itself.

As an example for a successful application of Lemma 7.1.1 we consider the definition of *intersperse* from Section 3.2. First, we define *intersperse* by means of the Haskell fixpoint operator $fix :: (\alpha \to \alpha) \to \alpha$. That is, we consider the following functional, as we have $intersperse \equiv fix\ body$.

$$
\begin{aligned}
&body :: (\alpha \to [\alpha] \to [\alpha]) \to \alpha \to [\alpha] \to [\alpha]\\
&body\ \_\ \_\ \ [\,] \qquad = [\,]\\
&body\ f\ \ sep\ (x : xs) =\\
&\quad \textbf{case}\ xs\ \textbf{of}\\
&\qquad [\,] \to x : [\,]\\
&\qquad \_ \to x : sep : f\ sep\ xs
\end{aligned}
$$

We apply **case** deferment to derive the following less strict implementation of the functional *body*.

$$
\begin{aligned}
&body' :: (\alpha \to [\alpha] \to [\alpha]) \to \alpha \to [\alpha] \to [\alpha]\\
&body'\ \_\ \_\ \ [\,] \qquad = [\,]\\
&body'\ f\ \ sep\ (x : xs) =\\
&\quad x : \textbf{case}\ xs\ \textbf{of}\\
&\qquad [\,] \to [\,]\\
&\qquad \_ \to sep : f\ sep\ xs
\end{aligned}
$$

To prove that $body'$ is indeed less strict than *body*, we have to provide arguments f, *sep*, and $x : xs$ such that $xs \equiv \bot$ and $x : \bot \sqsupset \bot$. These requirements are met for an arbitrary function $f :: \tau \to [\tau] \to [\tau]$, an arbitrary element $sep :: \tau$ and a list $x : \bot$, where x is an arbitrary value of type τ. In particular this proves that we have $body'\ f \sqsupset body\ f$ for all functions $f :: \tau \to [\tau] \to [\tau]$. We will later employ this observation to show that $fix\ body'$ is less strict than $fix\ body$.

As we are not interested in a less strict functional but in a less strict recursive definition of *intersperse*, we have to provide some means to prove that $fix\ body'$ is indeed less strict than $fix\ body$. Note that **case** deferment can only be applied to non-recursive definitions, as the right-hand sides of the **case** expressions of the functions f

and g have to agree except for the context D. In other words, we cannot apply **case** deferment to functions f and g if the right-hand sides of the **case** expressions of f and g contain recursive applications of the corresponding functions. Therefore, we prove the following Lemma by means of fixpoint induction (Bakker 1980).

Lemma 7.1.2: *Let $f, g :: \tau \to \tau$. If we have $g\ x \sqsupseteq f\ x$ for all $x :: \tau$, then $fix\ g \sqsupseteq fix\ f$.*

Proof: We prove this statement by fixpoint induction. We define a predicate $P(x, y)$ that takes two values of type τ and is satisfies if and only if $g\ x \sqsupseteq f\ y$. For the base case of the induction we have to prove that $P(\bot, \bot)$ holds. As we have $g\ x \sqsupseteq f\ x$ for all $x :: \tau$ by precondition, the base case holds. For the inductive step we have to show that for all $x, y :: \tau$ if $P(x, y)$ holds, then $P(g\ x, f\ y)$ also holds. We reason as follows.

$$
\begin{array}{ll}
& g\ (g\ x) \\
\sqsupseteq & \quad \{ \text{ precondition } \} \\
& f\ (g\ x) \\
\sqsupseteq & \quad \{ \text{ induction hypothesis, monotonicity of } f \} \\
& f\ (f\ y)
\end{array}
$$

Thus, $P(g\ x, f\ y)$ holds and, therefore, the inductive step holds as well. By the fixpoint induction principle this implies $P(fix\ g, fix\ f)$ or, in other words $g\ (fix\ g) \sqsupseteq f\ (fix\ f)$ which is equivalent to $fix\ g \sqsupseteq fix\ f$. □

As we have observed before, we have $body'\ f \sqsupseteq body\ f$ for all $f :: \tau \to [\tau] \to [\tau]$. Thus, by the previous lemma we also have $fix\ body' \sqsupseteq fix\ body$. This shows that *intersperse′* from Section 3.2 is less strict than *intersperse*.

The notion of "less strict" considered above refers to the definition of the less-strict relation from Definition 6.1.1. There a function f is less strict than a function g if we have $f \sqsupseteq g$, that is, $f\ x \sqsupseteq g\ x$ for all x of appropriate type. In contrast, according to the original definition of the less-strict relation (Definition 4.1.2) a function f is called less strict than a function g if we have $f \sqsupseteq g$, and f and g agree for total arguments. Therefore, we additionally have to show that two functions agree for total arguments to apply these results to the definition presented in Chapter 4. In most cases we consider functions that yield total results for all total arguments. If a function g yields total results for all total arguments and we have $f \sqsupseteq g$, that is, $f\ x \sqsupseteq g\ x$ for all x of appropriate type, then f also yields total results for all total arguments. For example, *intersperse* yields total results for all total arguments. In other words, by applying Lemma 7.1.1 and Lemma 7.1.2 we already know that *intersperse′* is less strict than *intersperse* in the sense of Definition 4.1.2 because *intersperse* yields total results for all total arguments. We only have to take additional care if we consider a function that yields a non-total result for a total argument like the division function on integers. All functions considered in the following sections yield total results for all total arguments, so we can apply Lemma 7.1.1 and Lemma 7.1.2 to derive less strict implementations in the sense of Definition 4.1.2.

7.2 Peano Multiplication

In this section we observe that the standard implementation of the multiplication of Peano numbers is too strict and present a benefit of less strict functions besides memory usage. Later we show that the standard implementation of an intersection of lists shows the same bad behavior as Peano multiplication.

Consider the following data type of Peano numbers.

$$\textbf{data}\ \mathit{Peano} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Peano}$$

We define addition and multiplication of Peano numbers as follows.

$$\begin{aligned}&\mathit{addP} :: \mathit{Peano} \to \mathit{Peano} \to \mathit{Peano}\\ &\mathit{addP}\ \mathit{Zero}\ \ \ \ \ \ n = n\\ &\mathit{addP}\ (\mathit{Succ}\ m)\ n = \mathit{Succ}\ (\mathit{addP}\ m\ n)\end{aligned}$$

$$\begin{aligned}&\mathit{multP} :: \mathit{Peano} \to \mathit{Peano} \to \mathit{Peano}\\ &\mathit{multP}\ \mathit{Zero}\ \ \ \ \ \ \_ = \mathit{Zero}\\ &\mathit{multP}\ (\mathit{Succ}\ m)\ n = \mathit{addP}\ n\ (\mathit{multP}\ m\ n)\end{aligned}$$

Furthermore, we define an infinite Peano number called *infinity*.

$$\begin{aligned}&\mathit{infinity} :: \mathit{Peano}\\ &\mathit{infinity} = \mathit{Succ}\ \mathit{infinity}\end{aligned}$$

This is a standard implementation of Peano number arithmetic. For example, the numbers package by Augustsson (2009) provides an identical implementation.

The evaluation of *multP Zero infinity* yields *Zero* in a non-strict programming language like Haskell. On the contrary, the evaluation of *multP infinity Zero* does not terminate. So, do all implementations of the multiplication of Peano numbers behave this way? aThat is, can we give an implementation of Peano multiplication in Haskell that yields *Zero* in both cases (without using parallelism)?

Sloth reports two counter-examples if we check *multP* up to size six.

```
> strictCheck multP 6
4: \(Succ ⊥) Zero -> Zero
7: \(Succ (Succ ⊥)) Zero -> Zero
Finished 23 tests.
```

As the counter-examples presented by Sloth are potential counter-examples, we have to verify them. For all total inputs that are more defined than *Succ* $\bot$ and *Zero* the function *multP* yields *Zero*, which is as defined as the recommended result *Zero*. Therefore, the counter-examples are definitely counter-examples.

In particular, the counter-examples show that a minimally strict implementation of Peano multiplication only has to evaluate the outermost *Succ* constructor of its first argument if the second argument is *Zero*. Therefore, a minimally strict implementation yields *Zero* for the arguments *infinity* and *Zero*. Thus, now we are able

to answer the question. There is an implementation of the multiplication of Peano numbers that terminates no matter whether it is applied to *Zero* and *infinity* or to *infinity* and *Zero*.

The evaluation of *multP infinity Zero* does not terminate because *multP* is inductively defined over its first argument. Hence, even if the second argument is *Zero*, the first argument is completely evaluated. Note that it, therefore, takes linear time in the size of p to evaluate the application *multP p Zero* for any Peano number p.

To derive a less strict implementation, we inline the definition of *addP* into *multP* and get the following equivalent definition.

$$
\begin{array}{l}
\mathit{multP} :: \mathit{Peano} \to \mathit{Peano} \to \mathit{Peano} \\
\mathit{multP}\ \mathit{Zero}\quad\ \_ = \mathit{Zero} \\
\mathit{multP}\ (\mathit{Succ}\ m)\ n = \\
\quad \textbf{case}\ n\ \textbf{of} \\
\qquad \mathit{Zero}\ \ \to \mathit{multP}\ m\ \mathit{Zero} \\
\qquad \mathit{Succ}\ n' \to \mathit{Succ}\ (\mathit{addP}\ m\ (\mathit{multP}\ (m\ (\mathit{Succ}\ n'))))
\end{array}
$$

If we consider this implementation, we observe that

$$\mathit{multP}\ m\ \mathit{Zero}$$

in the second rule yields *Zero* for all Peano numbers m. Well, in fact, not for all Peano numbers, if m is terminated by $\bot$, for example, $m \equiv \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \bot))$, then we have $\mathit{multP}\ m\ \mathit{Zero} \equiv \bot$. Therefore, we can replace the application *multP m Zero* by the constant *Zero* and get the following less strict implementation of *multP*. We slightly simplify the definition by employing the fact that pattern matching is performed from top to bottom. However, we still use the pattern $(\mathit{Succ}\ \_)$ instead of $\_$ in the second rule to emphasize that there is a bias towards the first argument with respect to strictness.

$$
\begin{array}{l}
\mathit{multP'} :: \mathit{Peano} \to \mathit{Peano} \to \mathit{Peano} \\
\mathit{multP'}\ \mathit{Zero}\quad\ \_\quad\ = \mathit{Zero} \\
\mathit{multP'}\ (\mathit{Succ}\ \_)\ \ \mathit{Zero} = \mathit{Zero} \\
\mathit{multP'}\ (\mathit{Succ}\ m)\ n\quad\ = \mathit{addP}\ n\ (\mathit{multP'}\ m\ n)
\end{array}
$$

Sloth does not report any counter-examples if we check *multP'* for Peano numbers up to size `50`. Note that for any Peano number p, it takes only constant time to evaluate *multP' p Zero* rather than linear time as for *multP p Zero*.

This is an example where we cannot apply **case** deferment to derive a less strict implementation. Instead we have to change the order of pattern matching. If the first argument has the form *Succ m*, the original implementation performs pattern matching on m. In contrast, the less strict implementation performs pattern matching on the second argument if the first argument has the form *Succ m*.

We might doubt the practical relevance of a minimally strict Peano multiplication. Therefore, we consider the function $\mathit{intersect} :: \mathit{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha]$ from the standard Haskell library *Data.List* that yields the intersection of two lists and is probably of greater practical relevance than Peano multiplication.

In fact, we do not know how we are supposed to check a function that has an equality constraint. We can regard a function with a type class constraint as a higher-order function as we can apply dictionary transformation (Wadler and Blott 1989). Therefore, we would be able to check *intersect* if we would be able to handle higher-order functions. Nevertheless, we conjecture that we can as well instantiate all type variables that have an *Eq* contraint with an integer type to check whether a function is minimally strict.

Although we did not prove this conjecture, we can easily show that it is not sufficient to check a monomorphic instance with respect to a type that has only finitely many values. For example, consider the function $nub :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$ that removes duplicate elements from a list. If we check the Boolean instance of *nub*, Sloth reports the following counter-examples.

```
> strictCheck (nub :: [Bool] -> [Bool]) 7
11: \(True:False:⊥) -> True:False:[]
12: \(False:True:⊥) -> False:True:[]
17: \(True:False:⊥:⊥) -> True:False:[]
19: \(False:True:⊥:⊥) -> False:True:[]
24: \(True:False:⊥:[]) -> True:False:[]
26: \(False:True:⊥:[]) -> False:True:[]
Finished 45 tests.
```

These counter-examples state that, if the first two elements of a list are the Boolean values *False* and *True*, the function *nub* does not have to inspect more elements as the result will be a list that only contains the elements *False* and *True* anyway. For every monomorphic instance with respect to a type that has only finitely many values we get similar counter-examples. This example demonstrates that we cannot use a monomorphic instance with respect to a type that has only finitely many values to check whether a polymorphic function with an *Eq* constraint is minimally strict. Thus, in the following we consider the integer instance of a polymorphic function with *Eq* constraint to check whether the function is minimally strict.

Back on the road we consider the following implementation of *intersect*. For simplicity, this implementation is an equivalent variation of the actual implementation in *Data.List*. The function $elem :: Eq\ \alpha \Rightarrow \alpha \to [\alpha] \to Bool$ takes a value and a list and checks whether the value is an element of the list.

$$
\begin{aligned}
&intersect :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha] \\
&intersect\ [\,]\quad\ \ \_ = [\,] \\
&intersect\ (x : xs)\ ys \\
&\quad \mid x\ `elem`\ ys = x : intersect\ xs\ ys \\
&\quad \mid otherwise\ \ = intersect\ xs\ ys
\end{aligned}
$$

Note that, if the first argument of *intersect* contains duplicate elements, then the result list will contain duplicate elements as well.

We check the implementation of the integer instance of *intersect* for arguments up to size five. Sloth reports the following counter-example. Note that we, in fact, use a data type with a finite number of constructors again as *Int* is restricted to a

finite number of bits[1]. Nevertheless, as the number of constructors is quite large, technically, the finiteness of *Int* does not cause problems.

```
> strictCheck (intersect :: [Int] -> [Int] -> [Int]) 5
4: \(⊥:⊥) [] -> []
Finished 13 tests.
```

As we have $intersect\ xs\ [\,] \equiv [\,]$ for all total list $xs :: [Int]$, the potential counter-example is definitely a counter-example. In fact, we do not only have $intersect\ (\bot : \bot)\ [\,] \equiv \bot$, as stated by the counter-example, but $intersect\ xs\ [\,] \equiv \bot$ for all lists xs that are terminated by $\bot$. For example, we have

$$intersect\ (replicate\ n\ x +\!\!+ \bot)\ [\,] \equiv \bot$$

for all $n :: Int$ and all values $x :: \tau$. Here, the application $replicate\ n\ x$ yields a list with n elements that only contains the value x. Furthermore, in the same way as the evaluation of $multP\ p\ Zero$ takes linear time in the size of p the evaluation of $intersect\ xs\ [\,]$ takes linear time in the size of xs.

We can define a less strict implementation of *intersect* by checking whether the second argument is the empty list in the same way as $multP'$ checks whether the second argument is *Zero*.

$$\begin{aligned}
&intersect' :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha] \\
&intersect'\ [\,]\ \ \ \ \ \ \ \ _ = [\,] \\
&intersect'\ (_ : _)\ [\,] = [\,] \\
&intersect'\ (x : xs)\ ys \\
&\quad \mid x\ `elem`\ ys = x : intersect'\ xs\ ys \\
&\quad \mid otherwise\ \ \ = intersect'\ xs\ ys
\end{aligned}$$

Sloth does not report a counter-example if we check this implementation for integer lists up to size ten. Note that the evaluation of $intersect'\ xs\ [\,]$ takes only constant time.

7.3 Binary Arithmetics

In this section we want to emphasize that the concept of minimal strictness is of particular interest if we consider functional logic programming languages. A functional logic programming language like Curry (Hanus 2006) can be considered as a functional language with non-determinism and free variables. In the following, we assume that the reader is familiar with the basic concepts of non-determinism and free variables.

Let us consider the algebraic data type for binary natural numbers presented by Braßel et al. (2008) for the functional logic programming language Curry. The syntax of Curry is very similar to the syntax of Haskell and the deterministic subset of Curry resembles Haskell.

[1] The type *Int* uses 32 or 64 bits depending on the architecture of the system, the GHC is running on.

$$\textbf{data}\ \mathit{Nat} = \mathit{IHi} \mid \mathit{O}\ \mathit{Nat} \mid \mathit{I}\ \mathit{Nat}$$

This data type defines a little-endian representation of unsigned binary numbers without leading zeros. The constructor *IHi* represents the most significant one-bit of a binary number, while the constructors *O* and *I* represent a zero and a one-bit, respectively. For example, the term $O\ (I\ \mathit{IHi})$ represents the decimal number six. Note that, in contrast to *Peano*, the data type *Nat* does not include zero. Braßel et al. (2008) furthermore define the following addition and multiplication of binary numbers, where *succN* yields the successor of a binary number.

$$\begin{aligned}
&\mathit{succN} :: \mathit{Nat} \to \mathit{Nat}\\
&\mathit{succN}\ \mathit{IHi}\ \ \ \ = O\ \mathit{IHi}\\
&\mathit{succN}\ (O\ n) = I\ n\\
&\mathit{succN}\ (I\ n)\ \ = O\ (\mathit{succN}\ n)
\end{aligned}$$

$$\begin{aligned}
&\mathit{addN} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{addN}\ \mathit{IHi}\ \ \ n\ \ \ \ \ \ = \mathit{succN}\ n\\
&\mathit{addN}\ (O\ m)\ \mathit{IHi}\ \ \ \ = I\ m\\
&\mathit{addN}\ (O\ m)\ (O\ n) = O\ (\mathit{addN}\ m\ n)\\
&\mathit{addN}\ (O\ m)\ (I\ n)\ \ = I\ (\mathit{addN}\ m\ n)\\
&\mathit{addN}\ (I\ m)\ \ \mathit{IHi}\ \ \ \ = O\ (\mathit{succN}\ m)\\
&\mathit{addN}\ (I\ m)\ \ (O\ n) = I\ (\mathit{addN}\ m\ n)\\
&\mathit{addN}\ (I\ m)\ \ (I\ n)\ \ = O\ (\mathit{addN}\ (\mathit{succN}\ m)\ n)
\end{aligned}$$

$$\begin{aligned}
&\mathit{multN} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{multN}\ \mathit{IHi}\ \ \ \ n = n\\
&\mathit{multN}\ (O\ m)\ n = O\ (\mathit{multN}\ m\ n)\\
&\mathit{multN}\ (I\ m)\ \ n = \mathit{addN}\ (O\ (\mathit{multN}\ m\ n))\ n
\end{aligned}$$

We use Sloth to check whether *multN* is unnecessarily strict. Note that we, in fact, check the corresponding Haskell implementation of *multN* as Sloth is implemented in Haskell. As *multN* is deterministic, it behaves identically in Haskell and Curry if we consider deterministic arguments. We check whether *multN* is minimally strict for all pairs of binary numbers up to size five.

```
> strictCheck multN 5
32: \(I ⊥) (O IHi) -> O (I ⊥)
33: \(I ⊥) (O (O ⊥)) -> O (O ⊥)
34: \(I ⊥) (O (I ⊥)) -> O (I ⊥)
Finished 103 tests.
```

Sloth reports three counter-examples. The first counter-example states that, *multN* applied to $I\ \bot$ and $O\ \mathit{IHi}$ yields $O\ \bot$ while there exists a less strict implementation that yields $O\ (I\ \bot)$ instead.

As all counter-examples consider arguments of the form $I\ \bot$ and $O\ m$ for some natural number m, we consider the evaluation of the application $\mathit{multN}\ (I\ \bot)\ (O\ m)$.

$$
\begin{array}{ll}
& multN\ (I\ \bot)\ (O\ m) \\
\equiv & \quad \{\ \text{definition of } multN\ \} \\
& addN\ (O\ (multN\ \bot\ (O\ m)))\ (O\ m) \\
\equiv & \quad \{\ \text{definition of } addN\ \} \\
& O\ (addN\ (multN\ \bot\ (O\ m))\ m) \\
\equiv & \quad \{\ \text{definition of } multN\ \} \qquad\qquad (*) \\
& O\ (addN\ \bot\ m) \\
\equiv & \quad \{\ \text{definition of } addN\ \} \\
& O\ \bot
\end{array}
$$

If we consider the step labeled with $(*)$, intuitively, we might think that *multN* can yield a more defined result. More precisely, for all total natural numbers m and n we have

$$multN\ n\ (O\ m) \equiv O\ (multN\ n\ m).$$

That is, instead of replacing $multN\ \bot\ (O\ m)$ by $\bot$ we get a less strict implementation if we replace the application by $O\ (multN\ \bot\ m)$. In other words, in the third rule of *multN* instead of performing pattern matching on m we would like to perform pattern matching on n, that is, the second argument of *multN*. We get the desired behavior if we swap the arguments of the recursive application of *multN* in the third rule of *multN*. That is, we define the following function $multN'$.

$$
\begin{array}{l}
multN' :: Nat \to Nat \to Nat \\
multN'\ IHi\quad n = n \\
multN'\ (O\ m)\ n = O\ (multN'\ m\ n) \\
multN'\ (I\ m)\ \ n = addN\ (O\ (multN'\ n\ m))\ n
\end{array}
$$

We might assume, that this change improves the implementation with respect to some arguments like $I\ \bot$ and $O\ IHi$ but worsens it with respect to other arguments. For example, what about the application $multN\ (I\ (O\ \bot))\ \bot$? In this case it seems to be disadvantageous to swap the arguments of the recursive application of *multN* as we get the application $multN\ \bot\ (O\ \bot)$ instead of the application $multN\ (O\ \bot)\ \bot$.

Let us consider the application $multN\ (I\ m)\ \bot$ for an arbitrary binary natural number m. We get the following equivalence.

$$
\begin{array}{ll}
& multN\ (I\ m)\ \bot \\
\equiv & \quad \{\ \text{definition of } multN\ \} \\
& addN\ (O\ (multN\ m\ \bot))\ \bot \\
\equiv & \quad \{\ \text{definition of } addN\ \} \\
& \bot
\end{array}
$$

As *addN* is strict in both arguments, no matter what the application of *multN* yields, the result of this application is $\bot$ as the second argument of the application of *addN* is $\bot$ as well. This examples shows that we have

$$multN\ (I\ m)\ \bot \equiv multN'\ (I\ m)\ \bot$$

for all natural numbers m. In other words, because *addN* consumes both its arguments uniformily it is advantageous if the recursive application of *multN* in the third rule of *multN* demands its second argument.

We might still not be convinced that *multN′* is an improvement with respect to strictness. Thus, we can check whether *multN′* is minimally strict using Sloth. Sloth does not report any counter-example if we check *multN′* for binary numbers up to size 10. However, note that this only states that *multN′* is probably minimally strict and not that *multN′* is less strict than *multN*. The functions *multN* and *multN′* may still be incomparable.

So, why do we bother whether *multN* is unnecessarily strict? Braßel et al. (2008) have introduced the data type *Nat* to guess numbers. In a functional logic programming language we can non-deterministically guess elements of a specific type by employing free variables. However, no Curry system can guess values of the primitive integer type. By employing the data type *Nat* it is possible to guess numbers while arithmetic operations for *Nat* are still reasonably fast (for example, in comparison to a Peano representation).

In general, in a non-deterministic context, functions that are too strict can lead to unnecessarily many non-deterministic branches, which in turn leads to unnecessary overhead. For example, let us consider the following function.

$$\begin{aligned} &\mathit{context} :: \mathit{Nat} \to \mathit{Bool} \\ &\mathit{context}\ (O\ (I\ \_)) = \mathit{True} \end{aligned}$$

The essence of an unnecessarily strict function is that it inspects a larger part of its argument than necessary to yield a certain result. For example, if we evaluate the expression $\mathit{context}\ (\mathit{multN}\ (I\ \mathit{unknown})\ (O\ \mathit{IHi})$ the function *multN* performs pattern matching on *unknown*, where *unknown* denotes a free variable. Therefore, the free variable is instantiated. When a variable of type *Nat* is instantiated, it is instantiated to all constructors of its type, applied to fresh free variables, namely, *IHi*, $O\ \mathit{unknown}$ and $I\ \mathit{unknown}$. For all these instantiations the considered expression yields *True*. Hence, the evaluation yields the result *True* three times.

```
> context (multN (I unknown) (O IHi))
True
More?

True
More?

True
More?

No more Solutions
```

On the other hand, if we use the less strict implementation *multN′* we only get a single result.

```
> context (multN' (I unknown) (O IHi))
True
More?

No more Solutions
```

Here, *multN′* does not perform pattern matching on *unknown*. Therefore, the free variable is not instantiated, and the expression is simply evaluated to *True*.

Thus, minimally strict functions are advantageous in a functional logic programming language. We can observe this advantage when we compare run-times for the generation of Pythagorean triples, using *multN* and *multN′* respectively.[2]

```
pythagorean :: (Nat, Nat, Nat)
pythagorean | addN (multN a a) (multN b b) == multN c c = (a, b, c)
  where
    a, b, c free
```

The variables *a*, *b*, and *c* are defined to be free variables by the keyword *free*. That is, *pythagorean* yields all triples of binary numbers *a*, *b*, and *c* such that $a * a + b * b = c * c$. For all numbers that do not satisfy this equation the guard yields a failure, that is, the empty result set.

number of triples	*multN*	*multN′*
100	3.87s	1.10s
200	17.81s	5.87s
300	33.42s	12.78s

It takes around three times as long to enumerate Pythagorean triples if we use the unnecessarily strict implementation of multiplication.

Most of the functions defined by Braßel et al. (2008) are unnecessarily strict. In the following we consider one additional example, namely, the function *cmpNat*, which compares two natural numbers and yields a value of type *Ordering*. The type *Ordering* has three values, namely, *EQ*, *LT*, and *GT*, which specify the relation of two values with respect to a given ordering. The following definition of *cmpNat* employs a function *isEQ* whose implementation is self-explanatory.

```
compNat :: Nat → Nat → Ordering
compNat IHi     IHi     = EQ
compNat IHi     (O _)   = LT
compNat IHi     (I _)   = LT
compNat (O _)   IHi     = GT
compNat (O m)   (O n)   = compNat m n
compNat (O m)   (I n)   =
  if isEQ cmpmn then LT else cmpmn
    where
      cmpmn = compNat m n
compNat (I _)   IHi     = GT
compNat (I m)   (O n)   =
  if isEQ cmpmn then GT else cmpmn
    where
      cmpmn = compNat m n
compNat (I m)   (I n)   = compNat m n
```

[2]The run-times are measured with the Curry compiler KiCS (Braßel and Huch 2009).

In Haskell we would probably merge the second and the third rule of *compNat* to a single rule. In Curry these definitions are not equivalent as overlapping rules are not evaluated top to bottom but non-deterministically. When we check *compNat*, Sloth reports the following counter-examples.

```
> strictCheck compNat 5
17: \(O IHi) (I ⊥) -> LT
20: \(I ⊥) (O IHi) -> GT
Finished 61 tests.
```

The first counter-example states that *compNat* yields $\bot$ if we apply it to $O\ IHi$ and $I\ \bot$, but there exists a less strict implementation that yields *LT* instead. And, indeed, if we compare $O\ IHi$ with $I\ \bot$, that is, if we compare 2 with $2 * n + 1$ where n is a natural number greater or equal to 1, the former one is certainly always smaller then the latter one.

If we apply *compNat* to $O\ IHi$ and $I\ \bot$, the function *compNat* compares *IHi* and $\bot$. In the case that the result of this test is *EQ* as well as if it is *LT* the result of $compNat\ (O\ IHi)\ (I\ \bot)$ is *LT*. If the recursive application yields *GT*, then *compNat* yields *GT* as well. That is, instead of applying the function *compNat* recursively in the case that the arguments have the form $(O\ m)$ and $(I\ n)$ we can check whether m is less or equal to n. This way *compNat* makes use of the fact that the result of *compNat* is not *EQ* if the outermost constructors already differ. Or in other words, if the outermost constructors differ we do not have to check whether the arguments of the constructors are equal. Using this idea we define the following less strict implementation of *compNat*. For readability we only present the rules that have changed in comparison with *compNat*.

$$
\begin{aligned}
&compNat' :: Nat \to Nat \to Ordering \\
&\ldots \\
&compNat'\ (O\ m)\ (I\ n) \ = \textbf{if}\ m <= n\ \textbf{then}\ LT\ \textbf{else}\ GT \\
&\ldots \\
&compNat'\ (I\ m)\ \ (O\ n) = \textbf{if}\ n <= m\ \textbf{then}\ GT\ \textbf{else}\ LT \\
&\ldots
\end{aligned}
$$

Braßel et al. (2008) have defined $(<=)$ and $(>=)$ by means of *compNat*. In this case, *compNat'* is as strict as *compNat*. In the following we analyse the behavior of the comparison functions in more detail.

As Braßel et al. (2008) correctly observe, we "can do better [..] than defining

$$
\begin{aligned}
&n == m = isEQ\ (compNat\ n\ m) \\
&n\ /= m = not\ (n == m)
\end{aligned}
$$

by sticking to the usual definition of $(==)$, which directly compares the constructors". In other words, if we implement the equality check $(==)$ by means of *compNat* (as well as *compNat'*), it is unnecessarily strict. For example, Sloth reports the following counter-examples if we check $(==)$.

```
> strictCheck (==) 5
10: \(O ⊥) (I ⊥) = False
12: \(I ⊥) (O ⊥) = False
17: \(O IHi) (I ⊥) = False
20: \(I IHi) (O ⊥) = False
Finished 61 tests.
```

The first counter-example is definitely a counter-example as for all total inputs, that are more defined than $O\ \bot$ and $I\ \bot$, the function $(==)$ yields *False* as well. Obviously, a naive implementation of $(==)$ also yields *False* for the arguments $O\ \bot$ and $I\ \bot$. An implementation of $(==)$ that is based on *compNat* cannot show the same behavior as *compNat* has to evaluate one of its arguments completely to yield a result. For example, *compNat* cannot decide whether $O\ \bot$ is greater or less than $I\ \bot$.

Although Braßel et al. (2008) have observed that it is disadvantageous to define $(==)$ by means of *compNat*, they have not observed that the same holds for the other comparison functions. We define $(<)$, $(>)$, $(<=)$ and $(>=)$ by means of *compNat* like it is suggested by Braßel et al. (2008) and observe their behavior in the following. We consider the following definitions of the comparison functions as suggested by Braßel et al. (2008).

$$\begin{aligned} n < m &= isLT\ (compNat\ n\ m) \\ n > m &= isGT\ (compNat\ n\ m) \\ n <= m &= not\ (n > m) \\ n >= m &= not\ (n < m) \end{aligned}$$

In the following we only consider the function $(<=)$. We get similar results for the other functions. When we check $(<=)$ using Sloth, we get the following counter-example.

```
> strictCheck (<=) 4
2: \IHi ⊥ -> True
Finished 37 tests.
```

The natural number *IHi* is less or equal to any other binary natural number. Therefore, we do not have to check the second argument. However, as $(<=)$ is defined by means of *compNat* it checks whether the second argument is *IHi* as in this case *compNat* yields *EQ*. If we increase the considered size, we observe that we also have $I\ \bot <= O\ IHi \equiv \bot$ while a minimally strict implementation yields *False* instead. As we have observed before, *compNat* is unnecessarily strict for the arguments $I\ \bot$ and $O\ IHi$ and, therefore, $(<=)$ is unnecessarily strict for these arguments as well. Thus, we have to provide a specific definition for $(<=)$ to get a minimally strict implementation.

In fact, this is a quite common problem that is not restricted to the comparison of binary numbers. For example, in Haskell the function $(<=) :: Bool \to Bool \to Bool$, which implements the Boolean implication, is unnecessarily strict. Sloth yields the following result.

```
> strictCheck ((<=) :: Bool -> Bool -> Bool) 3
2: \False ⊥ = True
Finished 7 tests.
```

In the same way as we expect $False \mathbin{\&\&} \bot \equiv False$ we might expect $False <= \bot \equiv True$. However, $(<=)$ is by default defined by means of *compare*. In other words, if we do not provide an implementation of $(<=)$ but an implementation of *compare*, then $(<=)$ is defined as follows.

$$
\begin{aligned}
&(<=) :: Ord\ \alpha \Rightarrow \alpha \to \alpha \to Bool \\
&x <= y = \\
&\quad \textbf{case}\ compare\ x\ y\ \textbf{of} \\
&\qquad GT \to False \\
&\qquad \_ \quad \to True
\end{aligned}
$$

This implementation of $(<=) :: Bool \to Bool \to Bool$ evaluates its second argument if its first argument is *False*.

In fact, we can show that every implementation of $(<=)$ that is implemented by means of *compare* for a data type that has a minimal element with respect to $(<=)$ is too strict. If the data type has a minimal element m with respect to $(<=)$, we can implement $(<=)$ such that $m <= \bot \equiv True$. In contrast, the Haskell function *compare* will always evaluate both arguments as they might be equal. Hence, if we implement $(<=)$ by means of *compare* we get $m <= \bot \equiv \bot$. Lastly, note that in most cases there is a minimal element with respect to $(<=)$. For example, all standard Haskell data types except for primitive data types like *Int* have a minimal element.

In Haskell, instead of only providing an implementation of *compare* we can as well only provide an implementation of $(<=)$. In this case *compare* is by default implemented by means of $(<=)$ as follows.

$$
\begin{aligned}
&compare :: Ord\ \alpha \Rightarrow \alpha \to \alpha \to Ordering \\
&compare\ x\ y = \textbf{if}\ x == y\ \textbf{then}\ EQ\ \textbf{else if}\ x <= y\ \textbf{then}\ LT\ \textbf{else}\ GT
\end{aligned}
$$

If so, both, $(<=)$ and *compare* are minimally strict, but *compare* potentially traverses its arguments twice because it first checks whether the arguments are equal and then checks whether they are related by $(<=)$. Therefore, it is advantageous to provide definitions for both functions, *compare* and $(<=)$. However, this is a very fine line because we have to trade modularity for a minimally strict implementation. Note that for all types whose $(<=)$ instance is unnecessarily strict the implementations of *min* and *max* are unnecessarily strict too as these functions are defined by means of $(<=)$ and the implementations of *minimum* and *maximum* are unnecessarily strict as these are defined by means of *min* and *max*.

Besides the *Nat* data, presented here, Braßel et al. (2008) present a data type for integer numbers that is based on *Nat*. A couple of functions for this integer type are not minimally strict because they are based on functions for *Nat* that are not minimally strict. Apart from these functions, subtraction of integers as well as division are not minimally strict.

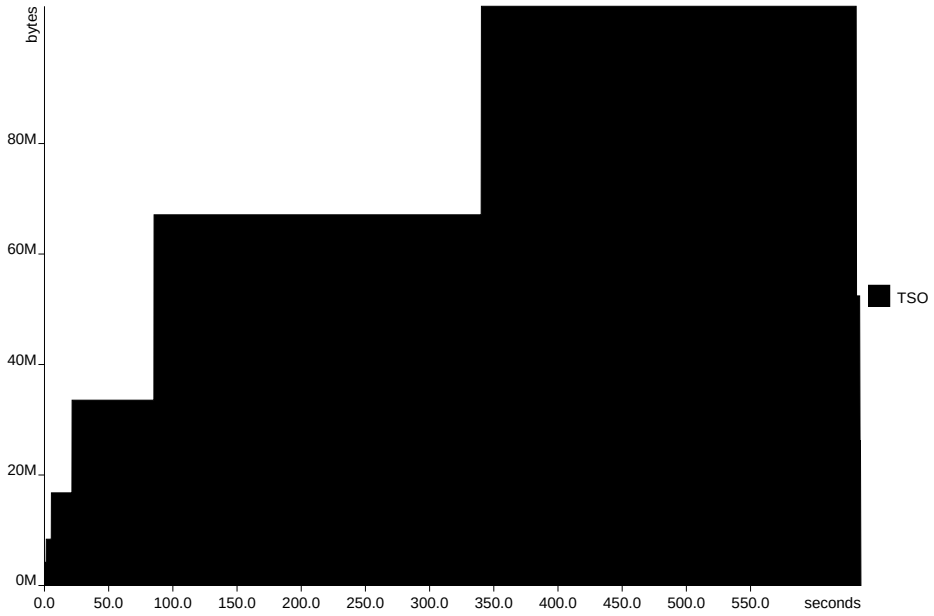


Figure 7.4.1: Heap Profile of Escaping Umlauts in Shakespeare's Collected Works using *splitWhen* from Version 0.1.2 of Split.

7.4 The split Package

In this section we check whether some functions of the Hackage package split, which provides a variety of functions to split lists into sublists, are minimally strict. We start by investigating the function *splitWhen* that splits a list by means of a predicate into a list of sublists.

First, we take a look at three different functions that solve this task, that is, we consider three functions of type $(\alpha \to \mathit{Bool}) \to [\alpha] \to [[\alpha]]$, that behave equally with respect to total arguments.

splitWhen **from version 0.1.4 of split by Yorgey (2011)** In Section 3.2 we have used this implementation to define *replaceBy*. Using this function escaping an umlaut in Shakespeare's collected works uses constant space.

chop **from the utility-ht package by Thielemann (2009)** We have implemented the function *replaceBy* by means of *chop* when we have originally observed that *intersperse* is unnecessarily strict (Christiansen 2011). Using this function escaping an umlaut in Shakespeare's collected works uses constant space as well.

splitWhen **from version 0.1.2 of split by Yorgey (2010)** We did not use the function *splitWhen* when we originally observed that *intersperse* is unnecessarily strict, because *splitWhen* from version 0.1.2 of split, which has been the current version back then, causes a space leak itself. Using this function escaping an umlaut in Shakespeare's collected works does not use constant space at all.

Figure 7.4.1 presents the heap profile of escaping an umlaut in Shakespeare's collected works using *splitWhen* from version 0.1.2 of split. This task takes more than

ten minutes and uses around one hundred megabyte of memory. In contrast, using *chop* or the current version of *splitWhen*, this process takes less than one second and less than three kilobyte of memory. That is, this application takes around 600 times as long and more than 30,000 times the memory if we use the old implementation of *splitWhen*. Even in comparison to the results for the unnecessarily strict implementation of *intersperse*, the consequences of using the implementation of *splitWhen* from version 0.1.2 of split are worse.

By using Sloth we observed that some fundamental functions of the split package are unnecessarily strict. The improvements between version 0.1.2 and 0.1.4 of the package are due to improvements of functions with respect to strictness communicated to the author of the package by private communication. In Section 7.4.1, we show how we can observe that the original implementation of *splitWhen* (from version 0.1.2 of split) is unnecessarily strict and improve its implementation. To improve *splitWhen* we improve a function that is the basis for most of the functions provided by the package split. That is, in fact, we do not only improve *splitWhen* but most of the functions provided by the package. In Section 7.4.2 we consider another function from split, called *insertBlanks* and improve it with respect to strictness.

Before we start, we want to state some preliminaries that are necessary to check a function using Sloth. The argument as well as the result type of the function have to be instances of the type classes *Typeable* and *Data* (Lämmel and Peyton Jones 2003). Haskell provides a mechanism to derive instances of these type classes by using the option *DeriveDataTypeable*. For example, using this option, to check the Peano multiplication from Section 7.2 we have to add the following line after the data type declaration for *Peano*.

deriving (*Typeable*, *Data*)

In Section 3.2, when we checked *intersperse*, we did not have to add a deriving statement as lists as well as the data type *A* are already instances of *Typeable* and *Data*.

Moreover, we can use so-called stand-alone deriving to generate these instances. More precisely, by using the option *StandaloneDeriving* we can derive instances of type classes without having to change the source code. That is, we can derive the required instances of data types that are defined in a library like split without changing the source code of the library. Without stand-alone deriving we would have to add the deriving statement to the data type declaration of the corresponding data type. The following code derives instances of *Typeable* and *Data* for the data type *Chunk*, which will be used in the following sections.

deriving instance *Typeable1 Chunk*
deriving instance *Data* $\alpha \Rightarrow$ *Data* (*Chunk* α)

Note that we have to derive an instance of *Typeable1* instead of *Typeable* because *Chunk* is a type constructor. Furthermore, if a type τ is an instance of *Typeable* and a type constructor σ is an instance of *Typeable1*, then $\sigma\ \tau$ is an instance of *Typeable* as well.

7.4.1 The Function splitWhen

After these preparations we would like to check if the function *splitWhen* is unnecessarily strict. Sadly, we cannot check whether this function is minimally strict as it has a higher-order type and Sloth does not support higher-order arguments. Therefore, we investigate the consequences of checking if *splitWhen* applied to a specific function is minimally strict. First, we check whether the partial application *splitWhen* (*const False*) is minimally strict. That is, we consider *splitWhen* applied to a predicate that is never satisfied. Sloth yields the following result.

```
> strictCheck (splitWhen (const False) :: [A] -> [[A]]) 3
1: \⊥ -> ⊥:[]
3: \(a:⊥) -> (a:⊥):[]
Finished 11 tests.
```

First of all both counter-examples stay counter-examples if we increase the considered size. The first counter-example states that *splitWhen* always yields a list with exactly one element. More precisely, if we apply *splitWhen* (*const False*) to $\bot$, we get $\bot$ while a minimally strict implementation yields $[\bot]$ instead. Obviously, *splitWhen* does not always yield a list with one element if we consider an arbitrary predicate. That is, the counter-example that shows that *splitWhen* (*const False*) is unnecessarily strict does not show that *splitWhen* is unnecessarily strict as well. Note that we cannot check which function *splitWhen* is applied to, and, therefore, we cannot define a function that satisfies the counter-examples if the functional argument of *splitWhen* is *const False*.

This example is supposed to demonstrate that we cannot check whether a higher-order function is minimally strict by checking whether its partial application to an arbitrary function is minimally strict. Nevertheless, at least in the case of *splitWhen* there exists a predicate that allows us to derive some useful informations. We use the identity function as predicate. In this case the elements of the list determine the result of the predicate. Note that we can only use *id* as a predicate because *splitWhen* is polymorphic. If we consider a monomorphic function similar to *splitWhen*, for example, of type $(Int \to Bool) \to [Int] \to [[Int]]$, we cannot use *id* as first argument. Instead we would have to define a function that interprets integers as Boolean values.

We get the following results if we check whether *splitWhen id* is minimally strict.

```
>  strictCheck (splitWhen id) 4
1: \⊥ -> ⊥:⊥
3: \(⊥:⊥) -> ⊥:⊥
4: \(False:⊥) -> (False:⊥):⊥
5: \(True:⊥) -> []:⊥:⊥
Finished 13 tests.
```

First of all, all counter-examples stay counter-examples if we increase the size. The second question to answer is whether the counter-examples depend on the specific predicate again. Let us consider the counter-example labeled with four. By evaluating the first element of the list, we are always able to determine whether the first

element satisfies the predicate no matter which predicate we consider. Therefore, if we apply *splitWhen p* to *False* : ⊥ independent, of the predicate *p* we know whether the first element satisfies the predicate. In the case of the predicate *id* the first element does not satisfy the predicate because the first element is *False*. In this case, obviously, any implementation of *splitWhen* is supposed to be able to yield the element for which the predicate is not satisfied. For example, consider the standard Haskell function *words* :: *String* → [*String*] that takes a string and splits this string at all occurrences of a whitespace. The following equivalence resembles the behavior that Sloth proposes for *splitWhen*.

$$words\ (\texttt{'a'} : \bot) \equiv (\texttt{'a'} : \bot) : \bot$$

That is, the presented counter-example is independent of the specific predicate.

To improve the implementation of *splitWhen* with respect to strictness we examine its implementation. The function *splitWhen* is defined as follows.

```
splitWhen :: (α → Bool) → [α] → [[α]]
splitWhen = split ∘ dropDelims ∘ whenElt
```

The function *whenElt* takes a predicate and yields a kind of configuration, and the function *dropDelims* enriches this configuration with additional information. The additional information is used to tell *split* that we want to drop the list elements that satisfy the predicate. As *dropDelims* and *whenElt* do not perform any pattern matching on the list, they cannot be the reason why *splitWhen* is unnecessarily strict. Therefore, we consider the implementation of *split*.

```
split :: Splitter α → [α] → [[α]]
split s = map fromElem ∘ postProcess s ∘ splitInternal (delimiter s)
```

The argument of type *Splitter α* is the configuration that, among other things, contains the predicate that is used to split the list. Without going into details about the definition of *split* let use consider the implementation of *splitInternal*, which is the basis for most functions defined in the package split.

```
splitInternal :: Delimiter α → [α] → [Chunk α]
splitInternal _ []              = []
splitInternal d xxs@(x : xs) =
  case matchDelim d xxs of
      Just ([], (r : rs))  → Delim [] : Text [r] : splitInternal d rs
      Just (match, rest)   → Delim match : splitInternal d rest
      _                    → x `consText` splitInternal d xs


consText :: α → [Chunk α] → [Chunk α]
consText z (Text c : ys) = Text (z : c) : ys
consText z ys            = Text [z] : ys
```

When we apply *splitWhen* to the predicate *id*, the function *splitInternal* is applied to *DelimEltPred id* where *DelimEltPred* is a wrapper that takes a predicate and yields

a value of the type *Delimiter* τ. To examine why the application *splitWhen id* is not minimally strict, we check whether *splitInternal* (*DelimEltPred id*) is minimally strict.

When we check *splitInternal* (*DelimEltPred id*), we get similar results as we have got for *splitWhen id*. Note that later the postprocessing of the result of *splitInternal* will remove the *Text* constructors and drop the *Delim* elements. Sloth yields the following results.

```
> strictCheck (splitInternal (DelimEltPred id)) 5
3: \(⊥:⊥) -> ⊥:⊥
4: \(False:⊥) -> Text (False:⊥):⊥
7: \(False:⊥:⊥) -> Text (False:⊥):⊥
9: \(True:⊥:⊥) -> Delim (True:[]):⊥:⊥
Finished 21 tests.
```

As expected, the function *splitInternal* is not minimally strict. So, why is the application *splitInternal* (*DelimEltPred id*) (*False* : ⊥) unnecessarily strict? The function *splitInternal* passes its arguments to *matchDelim*. By evaluating the application *matchDelim* (*DelimEltPred id*) (*False* : ⊥) we observe that it yields *Nothing*. Thus, the first two cases of the pattern matching in *splitInternal* do not match but the third one does. More precisely, the considered application of *splitInternal* yields the result of *consText False* (*splitInternal* (*DelimEltPred id*) ⊥). Furthermore, the following equivalence holds.

consText False (*splitInternal* (*DelimEltPred id*) ⊥)
≡ { definition of *splitInternal* }
consText False ⊥
≡ { definition of *consText* }
⊥

As the result of *matchDelim* is *Nothing*, and, therefore, cannot be less strict, we check whether *consText* is minimally strict. Note that we can use the more efficient type *A* in this case as *consText* is polymorphic. Moreover, we do not have to bother about higher-order as *consText* is a first-order function. Sloth yields the following counter-examples for *consText*.

```
> strictCheck (consText :: A -> [Chunk A] -> [Chunk A]) 3
1: \a ⊥ -> Text (a:⊥):⊥
3: \a (⊥:⊥) -> Text (a:⊥):⊥
Finished 5 tests.
```

Sloth reports that *consText* is unnecessarily strict. The first element of the result list is always a *Text* element. Nevertheless, the implementation performs pattern matching on its argument before it yields the first result element.

By replacing pattern matching by a **case** expression we get the following equivalent implementation of *consText*. This implementation shows that we can apply **case** deferment to derive a less strict implementation.

consText :: $\alpha \to [Chunk\ \alpha] \to [Chunk\ \alpha]$
consText x cs =

```
case cs of
    Text text : rest → Text (x : text) : rest
    _                → Text (x : [ ]) : cs
```

However, in this case we would need a context with multiple holes, more precisely, the context $C = \mathit{Text}\ (x : [\cdot]) : [\cdot]$. Instead of generalizing **case** deferment to contexts with multiple holes we introduce a tuple that combines the contents of the holes to a single value. More precisely, we introduce **let** expressions on all right-hand sides of the **case** expression and get the following equivalent implementation of *consText*. In a similar way we can handle all cases of contexts with multiple holes by introducing a **let** expression.

```
consText :: α → [Chunk α] → [Chunk α]
consText x cs =
  case cs of
    Text text : rest → let (text', rest') = (text, rest) in Text (x : text') : rest'
    _                → let (text', rest') = ([ ], cs)    in Text (x : text') : rest'
```

Now, we can use a context with a single hole, namely,

$$C = \mathbf{let}\ (\mathit{text}', \mathit{rest}') = [\cdot]\ \mathbf{in}\ \mathit{Text}\ (x : \mathit{text}) : \mathit{rest}.$$

Furthermore, we use $D = [\cdot]$ and apply Lemma 7.1.1 to derive the following implementation by moving the context C to the front of the definition.

```
consText' :: α → [Chunk α] → [Chunk α]
consText' x cs =
  let (text', rest') = case cs of
                         Text text : rest → (text, rest)
                         _                → ([ ], cs)
  in Text (x : text') : rest'
```

To show that this implementation is less strict we consider the arguments $x = a$ where a is an arbitrary value of some type τ and $cs = \bot$. In this case, we have $cs \equiv \bot$ and

$$
\begin{array}{l}
D[C[\bot]] \\
\quad \equiv \quad \{\ \text{definition of } C \text{ and } D\ \} \\
\mathbf{let}\ (\mathit{text}', \mathit{rest}') = \bot\ \mathbf{in}\ \mathit{Text}\ (a : \mathit{text}') : \mathit{rest}' \\
\quad \equiv \quad \{\ \text{semantics of } \mathbf{let} \text{ expression}\ \} \\
\mathit{Text}\ (a : \bot) : \bot \\
\quad \sqsupset \\
\bot.
\end{array}
$$

Thus, *consText'* is less strict than *consText*. Note that we do not have to consider the definition of *consText* by means of an explicit fixpoint to apply **case** deferment as its definition is non-recursive.

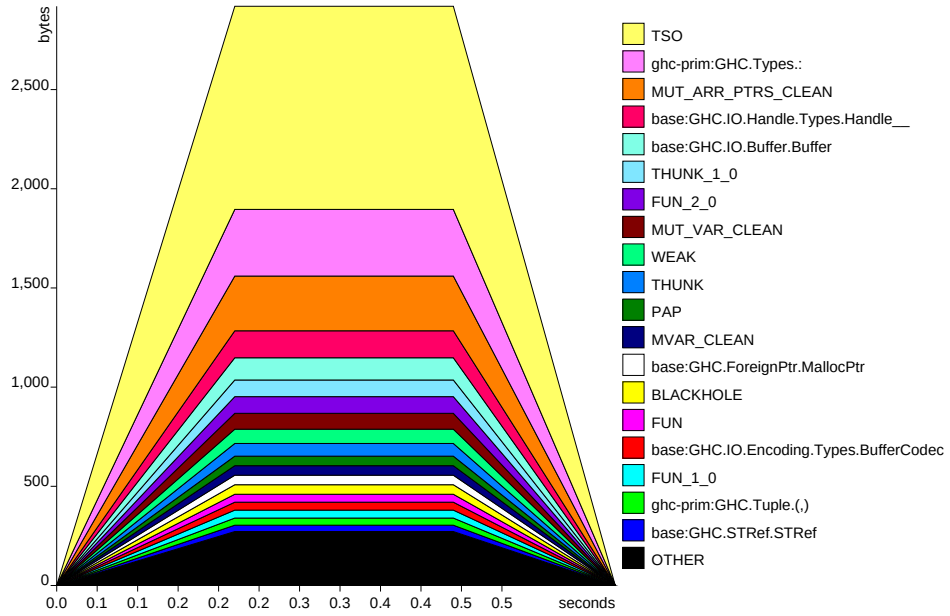


Figure 7.4.2: Heap Profile of Escaping Umlauts with the Less Strict Implementation of *splitWhen*.

As we will reuse it later, we define a function *splitText* at top-level instead of using a **case** expression and use the following definition of $consText'$ that is equivalent to the derived implementation.

$$
\begin{aligned}
&consText' :: \alpha \to [Chunk\ \alpha] \to [Chunk\ \alpha] \\
&consText'\ x\ cs = Text\ (x : text) : rest \\
&\quad \textbf{where} \\
&\qquad (text, rest) = splitText\ cs \\
&splitText :: [Chunk\ \alpha] \to ([\alpha], [Chunk\ \alpha]) \\
&splitText\ (Text\ text : rest) = (text, rest) \\
&splitText\ cs \qquad\qquad\quad = ([\,], cs)
\end{aligned}
$$

Figure 7.4.2 shows the heap profile of escaping umlauts if we use this less strict implementation of *consText* and, therefore, a less strict implementation of *splitWhen*. This profile is quite similar to the profile that we get for *chop* (Thielemann 2009) and *splitWhen* from version 0.1.4 of the split package (Yorgey 2011).

So, why do we get a significant performance improvement by improving the inconspicuous looking function *consText*? When we take a closer look at the counter-example, we observe that if the predicate yields *False* for all list elements, then *splitInternal* does not yield a first *Text* element until the list is terminated by the empty list. For example, for the function *splitInternal* from version 0.1.2 of split we have

$$splitInternal\ (DelimEltPred\ id)\ (replicate\ n\ False +\!\!+ \bot) \equiv \bot$$

for all $n :: Int$. That is, we have to completely evaluate the list structure to get the first element of the result list. As we consider a predicate that is not satisfied for any list element — we check whether any character of Shakespeare's collected works is the

umlaut "ä" —, we have to completely evaluate the argument of *splitWhen*, which is the string containing Shakespeare's collected works. If we take another look at the profile in Figure 7.4.1, we observe that the heap is filled by TSO elements. TSO is the abbreviation of Thread State Object. Although the name includes "Thread", these objects also occur in non-threaded programs where they refer to the stack usage of the main thread. In this particular example, by being unnecessarily strict, the compiler has to evaluate a cascade of non-tail recursive applications and to allocate stack space for all these applications. If the first element of a list does not satisfy the predicate, then an application of *splitInternal* reduces to an application of *consText* to the first element of the list and a recursive application of *splitInternal* to the rest of the list. However, *consText* does not yield a result before it has evaluated its second argument, namely, the application of *splitInternal* to the rest of the list. Thus, if all elements of a list do not satisfy a predicate, we have to allocate stack space for as many applications of *splitInternal* as the list has elements. In the case of Shakespeare's collected works the compiler allocates stack space for as many applications of *splitInternal* as Shakespeare has written down characters.

As we have observed, the function *splitWhen* is another example of a function whose performance can be improved by improving its implementation with respect to strictness. Yet, there are many ways to interfere with the memory behavior even if a function is minimally strict. For example, an implementation of *splitWhen* that is based on $consText'$ causes a space leak if a compiler would use a naive approach to the implementation of lazy pattern matchings like the **where** clause in $consText'$. Wadler (1987) has presented an optimization that prevents this kind of space leak.

To get the basic idea of this optimization let us consider the naive approach using the example of $consText'$. The naive approach introduces a variable, called *pair*, for the result of the application of *splitText*. Furthermore, it replaces the variable *text* by *fst pair* and *rest* by *snd pair*, where *fst* and *snd* are projections to the first and the second component of a pair, respectively. That is, we get the following implementation.

$$
\begin{aligned}
&consText'_2 :: \alpha \to [Chunk\ \alpha] \to [Chunk\ \alpha] \\
&consText'_2\ x\ cs = Text\ (x : fst\ pair) : snd\ pair \\
&\quad \textbf{where} \\
&\qquad pair = splitText\ cs
\end{aligned}
$$

To cause a space leak we have to consider a context that first evaluates the first list element of the result of $consText'_2$ and afterwards evaluates the rest of the list. When the application *fst pair* is evaluated, the application *splitText cs* is evaluated. Although, the function does not reference the first component of the pair anymore, the garbage collector is not able to deallocate the corresponding memory as the program still keeps a reference to the whole pair via the application *snd pair*. To overcome this problem, intuitively, when *fst pair* is evaluated occurrences of *snd pair* are replaced by a shortcut to the second component of the tuple. This way, there is no reference that keeps the whole pair alive. The interested reader may consider the work by Wadler (1987) and Sparud (1993) for more details about this problem and its solution.

Although this optimization prevents *consText′* from causing a space leak, we can use a denotationally equivalent implementation of *consText′* that does not allow for this kind of optimization. For example, consider the following function $consText'_3$.

$$
\begin{aligned}
&consText'_3 :: \alpha \to [Chunk\ \alpha] \to [Chunk\ \alpha] \\
&consText'_3\ x\ cs = Text\ (x : text\ cs) : rest\ cs \\
&text\ (Text\ t\colon \_) = t \\
&text\ \_ \qquad\qquad = [\,] \\
&rest\ (Text\ \_ : cs) = cs \\
&rest\ cs \qquad\qquad = cs
\end{aligned}
$$

If we define *splitWhen* by means of $consText'_3$, escaping an umlaut uses less memory than using the original definition of *splitWhen*, but it uses significantly more memory than using an implementation of *splitWhen* that is based on *consText′*.

Let us consider a context as we have done above, that is, a context that first evaluates the first list element of the result of $consText'_3$ and afterwards evaluates the rest of the list. If the first list element of the result is evaluated, the application *text cs* is evaluated. Although the function *text* projects to a part of the argument of *cs* that is not used later on, the garbage collector is not able to deallocate the occupied memory because the program still has a reference to the argument list *cs*. Instead, the function holds the part of *cs* that has been evaluated until the expression *rest cs* is evaluated.

This example reveals a general problem of minimally strict functions. From the point of the less-strict relation, as defined in this thesis, the functions *consText′* and $consText'_3$ are equivalent as we use a denotational view on functions. That is, among all minimally strict implementations we want to chose the implementation that uses the smallest amount of memory. Obviously, an operational view on functions is desirable if we want to reason about memory behavior. In Section 7.5 we will observe that even with a denotational model we can still draw some connections between the less-strict relation and memory usage.

Let us switch back to the undertaking of defining a minimally strict implementation of *splitWhen*, we observe that, in fact, *splitInternal* and, thus, *splitWhen* are still unnecessarily strict. By *splitInternal′* we denote an implementation of *splitInternal* that uses the minimally strict implementation *consText′* instead of the original implementation *consText*. Sloth reports the following counter-examples for *splitInternal′*.

```
> strictCheck (splitInternal' (DelimEltPred id)) 2
3: \(⊥:⊥) -> ⊥:⊥
Finished 5 tests.
```

If we take a second look at the definition of *splitInternal*, we observe that all right-hand sides of the **case** expression yield a list with at least one element. In other words, there is a common context that occurs on all right-hand sides of the case expression. Hence, we can use **case** deferment to derive the following less strict definition.

$$
\begin{aligned}
&splitInternal'' :: Delimiter\ \alpha \to [\alpha] \to [Chunk\ \alpha] \\
&splitInternal''\ \_\ [\,] \qquad\qquad = [\,]
\end{aligned}
$$

$$
\begin{array}{l}
splitInternal''\ d\ xxs@(x:xs) = hd:tl \\
\quad \textbf{where} \\
\quad\quad (hd, tl) = \textbf{case}\ matchDelim\ d\ xxs\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad Just\ ([\,], (r:rs)) \to (Delim\ [\,], Text\ [r]:splitInternal''\ d\ rs) \\
\quad\quad\quad\quad\quad\quad Just\ (match, rs) \to (Delim\ match, splitInternal''\ d\ rs) \\
\quad\quad\quad\quad\quad\quad \_ \to (Text\ (x:text), rest) \\
\quad\quad (text, rest) = splitText\ (splitInternal''\ d\ xs)
\end{array}
$$

However, this implementation is still unnecessarily strict as the following counter-example demonstrates.

```
> strictCheck (splitInternal'' (DelimEltPred id)) 3
6: \(⊥:[]) -> ⊥:[]
Finished 13 tests.
```

At first sight, this counter-example might look as it contradicts the sequentiality of *splitInternal''* as we have $splitInternal''\ (False:\bot) \equiv Text\ (False:\bot):\bot$. That is, we are supposed to perform pattern matching on the head of the list and the rest of the list. The counter-example does not contradict sequentiality as we are supposed to perform pattern matching on the head and the tail of the list with respect to different result positions. With respect to the head of the result list, we perform pattern matching on the head of the argument list and with respect to the tail of the result list, we perform pattern matching on the tail of the argument list.

The application $splitInternal''\ (DelimEltPred\ id)\ (\bot:[\,])$ applies *matchDelim* to the arguments $DelimEltPred\ id$ and $\bot:[\,]$. This application yields $\bot$ and the result of *splitInternal''* is $\bot:\bot$. To define a less strict implementation, we add an additional check whether the tail of the argument list is the empty list. We replace the expression

$$head:tail$$

by

$$head:\textbf{if}\ null\ xs\ \textbf{then}\ [\,]\ \textbf{else}\ tail.$$

Note that this new implementation is not more strict than the original one. If we have $xs \equiv \bot$, then $tail \equiv \bot$, and, therefore,

$$\textbf{if}\ null\ xs\ \textbf{then}\ [\,]\ \textbf{else}\ tail \equiv \bot \equiv tail.$$

However, if $x \equiv \bot$ and $xs \equiv [\,]$, then $tail \equiv \bot$ and, thus,

$$\textbf{if}\ null\ xs\ \textbf{then}\ [\,]\ \textbf{else}\ tail \equiv [\,] \sqsupset tail.$$

7.4.2 The Function insertBlanks

There are several other functions in the split package that are unnecessarily strict. In the following, we exemplarily consider the function *insertBlanks*. It takes a list of *Chunk*s, that is, a list of *Text* and *Delim* elements and inserts blanks, namely, *Text* elements with an empty list, between all pairs of succeeding delimiters. Furthermore,

the function adds a blank at the end of the list, and if the first list element is a delimiter the function adds a blank at the front. We do not discuss the implementation of *insertBlanks* in detail as we will only apply purely syntactical transformations in the following.

$$
\begin{aligned}
&\mathit{insertBlanks} :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanks}\ [\,] &&= [\mathit{Text}\ [\,]] \\
&\mathit{insertBlanks}\ (d@(\mathit{Delim}\ \_) : l) &&= \mathit{Text}\ [\,] : \mathit{insertBlanksRest}\ (d : l) \\
&\mathit{insertBlanks}\ l &&= \mathit{insertBlanksRest}\ l
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{insertBlanksRest} :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanksRest}\ [\,] &&= [\,] \\
&\mathit{insertBlanksRest}\ [d@(\mathit{Delim}\ \_)] &&= [d, \mathit{Text}\ [\,]] \\
&\mathit{insertBlanksRest}\ (d1@(\mathit{Delim}\ \_) : d2@(\mathit{Delim}\ \_) : l) &&= \\
&\quad d1 : \mathit{Text}\ [\,] : \mathit{insertBlanksRest}\ (d2 : l) \\
&\mathit{insertBlanksRest}\ (c : l) &&= c : \mathit{insertBlanksRest}\ l
\end{aligned}
$$

When we check *insertBlanks* using Sloth, we get the following result.

```
> strictCheck (insertBlanks :: [Chunk A] -> [Chunk A]) 2
1: \_ -> Text _:_
3: \(_:_) -> Text _:_
Finished 5 tests.
```

The first counter-example states that the first element of the result of *insertBlanks* is always a *Text* element. Nevertheless, *insertBlanks* first evaluates its argument before it yields the first element of the result list.

We start with a simple transformation of the definition of *insertBlanks*. The data type *Chunk* provides two constructors, namely, *Text* and *Delim*, which both take a list as argument. Hence, in the last rule of *insertBlanks* the variable l has the form $\mathit{Text}\ \mathit{text} : l$. Besides this observation, we use the following equality

$$\mathit{insertBlanksRest}\ (\mathit{Text}\ \mathit{text} : l) \equiv \mathit{Text}\ \mathit{text} : \mathit{insertBlanksRest}\ l$$

and arrive at an equivalent definition of *insertBlanks*.

$$
\begin{aligned}
&\mathit{insertBlanks} :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanks}\ [\,] &&= [\mathit{Text}\ [\,]] \\
&\mathit{insertBlanks}\ (d@(\mathit{Delim}\ \_) : l) &&= \mathit{Text}\ [\,] : \mathit{insertBlanksRest}\ (d : l) \\
&\mathit{insertBlanks}\ (\mathit{Text}\ \mathit{text} : l) &&= \mathit{Text}\ \mathit{text} : \mathit{insertBlanksRest}\ l
\end{aligned}
$$

To derive a less strict implementation of *insertBlanks* we consider the following equivalent implementation of *insertBlanks* that uses a **case** expression. Furthermore, we use the equivalence $\mathit{insertBlanksRest}\ [\,] \equiv [\,]$ and change the order of the rules in the **case** expression.

$$
\begin{aligned}
&\mathit{insertBlanks} :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanks}\ \mathit{cs} =
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{case}\ cs\ \textbf{of} \\
\quad \begin{array}{ll}
Text\ text : l & \to Text\ text : insertBlanksRest\ l \\
l & \to Text\ [\,] \quad : insertBlanksRest\ l
\end{array}
\end{array}
$$

Now, we can apply **case** deferment to derive a less strict implementation. We consider the contexts $D = [\cdot]$ and $C = Text\ [\cdot] : [\cdot]$ and apply Lemma 7.1.1. As we consider a context with multiple holes, we have to introduce a **let** expression, but we abstain from developing the implementation step by step and simply present the final result. After applying **case** deferment we observe that we can reuse *splitText* and get the following less strict implementation.

$$
\begin{array}{l}
insertBlanks' :: [Chunk\ \alpha] \to [Chunk\ \alpha] \\
insertBlanks'\ l = Text\ text : insertBlanksRest\ rest \\
\quad \textbf{where} \\
\quad\quad (text, rest) = splitText\ l
\end{array}
$$

However, this new implementation of *insertBlanks* is still unnecessarily strict as the following result shows.

```
> strictCheck (insertBlanks' :: [Chunk A] -> [Chunk A]) 4
8: \(Delim _:_) -> Text []:Delim _:Text _:_
13: \(Delim _:_:_) -> Text []:Delim _:Text _:_
Finished 23 tests.
```

Because the counter-examples only affect the tail of the result list, probably the function *insertBlanksRest* is too strict. We verify this assumption by checking whether *insertBlanksRest* is minimally strict.

```
> strictCheck (insertBlanksRest :: [Chunk A] -> [Chunk A]) 4
3: \(_:_) -> _:_
4: \(Delim _:_) -> Delim _:Text _:_
7: \(Delim _:_:_) -> Delim _:Text _:_
Finished 19 tests.
```

And, indeed, we get similar results when we check *insertBlanksRest*. Thus, we examine *insertBlanksRest* in detail. As the definition of *insertBlanksRest* is quite unwieldy, we define the following equivalent implementation of *insertBlanks* by introducing a **case** expression to perform pattern matching if the first list element is a delimiter. We replace the second and third rule of *insertBlanksRest* by the following rule.

$$
\begin{array}{l}
insertBlanksRest\ (d1@(Delim\ \_) : l1) = \\
\quad d1 : \textbf{case}\ l1\ \textbf{of} \\
\qquad \begin{array}{ll}
[\,] & \to [Text\ [\,]] \\
d2@(Delim\ \_) : l & \to Text\ [\,] : insertBlanksRest\ (d2 : l) \\
c : l & \to c : insertBlanksRest\ l
\end{array}
\end{array}
$$

When we take a closer look at the **case** expression of this rule, we observe that it is equivalent to the original definition of *insertBlanks*. Thus, we can replace the **case** expression by an application of $insertBlanks'$ as follows.

$$
\begin{aligned}
&\mathit{insertBlanksRest} :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanksRest}'\ [\,] &&= [\,] \\
&\mathit{insertBlanksRest}'\ (d@(\mathit{Delim}\ \_) : l) &&= d : \mathit{insertBlanks}'\ l \\
&\mathit{insertBlanksRest}'\ (c : l) &&= c : \mathit{insertBlanksRest}'\ l
\end{aligned}
$$

Now, we are able to apply **case** deferment and get the following less strict implementation. For simplicity, we replace the **case** expression by a predicate that checks whether its argument is a delimiter, called *isDelim*.

$$
\begin{aligned}
&\mathit{insertBlanksRest}' :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanksRest}'\ [\,] \quad = [\,] \\
&\mathit{insertBlanksRest}'\ (c : l) = \\
&\quad c : \mathbf{if}\ \mathit{isDelim}\ c\ \mathbf{then}\ \mathit{insertBlanks}'\ l\ \mathbf{else}\ \mathit{insertBlanksRest}'\ l
\end{aligned}
$$

We have presented *insertBlanks* for a particular reason. The memory performance of replacing umlauts in Shakespeare's collected works using the less strict implementation of *insertBlanks* is worse than the memory behavior if we use the original implementation. As we have mentioned before, the compiler applies an optimization to prevent a space leak in the presence of lazy pattern matching as it is used in the definition of $\mathit{insertBlanks}'$. However, in some cases, the current implementation of this optimization implemented in the GHC fails (Marlow 2008). More precisely, by applying other optimizations first, the compiler is unable to observe that the resulting program projects to the components of a tuple (Felgenhauer 2010). We can always prevent the space leak, caused by a lazy pattern matching, by replacing it by a **case** expression as follows.

$$
\begin{aligned}
&\mathit{insertBlanks}'_2 :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanks}'_2\ l = \\
&\quad \mathbf{case}\ \mathit{splitText}\ l\ \mathbf{of} \\
&\qquad (\mathit{text}, \mathit{rest}) \to \mathit{Text}\ \mathit{text} : \mathit{insertBlanksRest}'\ \mathit{rest}
\end{aligned}
$$

However, this implementation is too strict again, as we have $\mathit{insertBlanks}'_2\ \bot \equiv \bot$. Yet, there even exists an implementation that does not have a space leak and is still minimally strict. We use a mixture of a lazy pattern matching by means of a **where** clause and a strict pattern matching on the tuple. In the original definition, the compiler is unable to apply the optimization because we apply a function to one of the components of the tuple. Therefore, we introduce a **case** expression that applies this function to one of the components of the tuple but yields the complete tuple structure. The resulting tuple is destructed lazily by a **where** clause. More precisely, we consider the following function.

$$
\begin{aligned}
&\mathit{insertBlanks}'_3 :: [\mathit{Chunk}\ \alpha] \to [\mathit{Chunk}\ \alpha] \\
&\mathit{insertBlanks}'_3\ l = \mathit{Text}\ \mathit{text}' : \mathit{rest}' \\
&\quad \mathbf{where} \\
&\qquad (\mathit{text}', \mathit{rest}') = \mathbf{case}\ \mathit{splitText}\ l\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad (\mathit{text}, \mathit{rest}) \to (\mathit{text}, \mathit{insertBlanksRest}'\ \mathit{rest})
\end{aligned}
$$

Because we do not apply a function to the variable *rest'* anymore, the compiler is able to identify the tuple selectors in contrast to the original minimally strict definition *insertBlanks'*. We did not investigate whether this transformation can be applied in general when using the current version of the GHC to derive an implementation that does not cause the space leak characterized by Wadler (1987).

7.5 Reversing Lists

When we considered memory improvements that arise from less strict implementations in the previous sections, we have always considered only one specific application. We have demonstrated that minimally strict implementations of *intersperse* and *splitWhen* are advantageous with respect to memory usage when we consider a program that replaces all occurrences of an umlaut in Shakespeare's collected works. In this section we discuss the memory behavior of two minimally strict implementations in comparison to the original implementation with respect to different contexts. We consider three different implementations of *reverse* and examine their behavior with respect to three contexts. We will observe that the space usage of an implementation depends highly on the context it is used in. Gustavsson and Sands (1999, 2001) have provided a formal model for reasoning about space usage in a lazy language, but we will only informally argue about the space usage of a function here.

To begin with, consider the following linear complexity implementation of *reverse*.

$$
\begin{aligned}
&\mathit{reverse} :: [\alpha] \to [\alpha] \\
&\mathit{reverse} = \mathit{rev}\ [\,] \\
&\quad \textbf{where} \\
&\qquad \mathit{rev}\ \mathit{xs}\ [\,] \quad\ \ = \mathit{xs} \\
&\qquad \mathit{rev}\ \mathit{xs}\ (y : \mathit{ys}) = \mathit{rev}\ (y : \mathit{xs})\ \mathit{ys}
\end{aligned}
$$

When we check *reverse* using Sloth we, somehow surprisingly, observe that it is not minimally strict.

```
> strictCheck (reverse :: [A] -> [A]) 4
3: \(a:⊥) -> ⊥:⊥
5: \(a:b:⊥) -> ⊥:⊥:⊥
Finished 9 tests.
```

These counter-examples state that, although the argument list is terminated by $\bot$, *reverse* is supposed to yield the spine of the result list. If we apply *reverse* to a total list that is more defined than $a : \bot$, that is, to a list with at least one element, the result is always a list with at least one element. Therefore, the potential counter-examples are indeed counter-examples. However, every function that uses an accumulator parameter, like *reverse*, yields $\bot$ if the argument list is terminated by $\bot$.

We can define a minimally strict implementation of *reverse* as follows. The function $\mathit{last} :: [\alpha] \to \alpha$ yields the last element of a non-empty list and the function $\mathit{init} :: [\alpha] \to [\alpha]$ yields a list without this last element.

$$
\begin{aligned}
&\mathit{reverse}' :: [\alpha] \to [\alpha] \\
&\mathit{reverse}'\ [\,] = [\,] \\
&\mathit{reverse}'\ \mathit{xs} = \mathit{last}\ \mathit{xs} : \mathit{reverse}'\ (\mathit{init}\ \mathit{xs})
\end{aligned}
$$

This implementation yields $\bot : \bot$ for the argument $a : \bot$ as the following equivalence shows.

$$
\begin{aligned}
&\mathit{reverse}'\ (a : \bot) \\
&\equiv \quad \{ \text{ definition of } \mathit{reverse}' \ \} \\
&\mathit{last}\ (a : \bot) : \mathit{reverse}'\ (\mathit{init}\ (a : \bot)) \\
&\equiv \quad \{ \text{ definition of } \mathit{last} \ \} \\
&\bot : \mathit{reverse}'\ (\mathit{init}\ (a : \bot)) \\
&\equiv \quad \{ \text{ definition of } \mathit{init}, \text{ namely rule } \mathit{init}\ [x] = [\,] \ \} \\
&\bot : \mathit{reverse}'\ \bot \\
&\equiv \quad \{ \text{ definition of } \mathit{reverse}' \ \} \\
&\bot : \bot
\end{aligned}
$$

While *reverse′* is minimally strict, in contrast to *reverse* it has a quadratic complexity. Note that the naive quadratic complexity implementation of *reverse* by means of $(+\!\!+)$ is not minimally strict.

Though, there is even a linear complexity implementation of *reverse* that is minimally strict. The result of *reverse* has as many list constructors as its argument. We define a function *withShapeOf*, that takes the result of *reverse* and its argument and reconstructs the spine of the result by performing pattern matching on the argument. By using a lazy pattern matching, the function *withShapeOf* only performs pattern matching on *l* if any of the elements of the result of *withShapeOf* is evaluated.

$$
\begin{aligned}
&\mathit{withShapeOf} :: [\alpha] \to [\beta] \to [\alpha] \\
&\_\ \text{`}\mathit{withShapeOf}\text{`}\ [\,] \quad\ = [\,] \\
&l\ \text{`}\mathit{withShapeOf}\text{`}\ (\_ : \mathit{ys}) = x : (\mathit{xs}\ \text{`}\mathit{withShapeOf}\text{`}\ \mathit{ys}) \\
&\quad \textbf{where}\ x : \mathit{xs} = l
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reverse}'' :: [\alpha] \to [\alpha] \\
&\mathit{reverse}''\ \mathit{xs} = \mathit{reverse}\ \mathit{xs}\ \text{`}\mathit{withShapeOf}\text{`}\ \mathit{xs}
\end{aligned}
$$

The function *reverse″* is also minimally strict as the following equivalence illustrates.

$$
\begin{aligned}
&\mathit{reverse}''\ (a : \bot) \\
&\equiv \quad \{ \text{ definition of } \mathit{reverse}'' \ \} \\
&\mathit{reverse}\ (a : \bot)\ \text{`}\mathit{withShapeOf}\text{`}\ (a : \bot) \\
&\equiv \quad \{ \text{ definition of } \mathit{withShapeOf} \ \} \\
&\mathit{head}\ (\mathit{reverse}\ (a : \bot)) : (\mathit{tail}\ (\mathit{reverse}\ (a : \bot))\ \text{`}\mathit{withShapeOf}\text{`}\ \bot) \\
&\equiv \quad \{ \text{ definition of } \mathit{reverse} \ \} \\
&\mathit{head}\ (\mathit{reverse}\ (a : \bot)) : (\mathit{tail}\ \bot\ \text{`}\mathit{withShapeOf}\text{`}\ \bot) \\
&\equiv \quad \{ \text{ definition of } \mathit{tail} \ \} \\
&\mathit{head}\ (\mathit{reverse}\ (a : \bot)) : (\bot\ \text{`}\mathit{withShapeOf}\text{`}\ \bot) \\
&\equiv \quad \{ \text{ definition of } \mathit{withShapeOf} \ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{head}\ (\mathit{reverse}\ (a : \bot)) : \bot \\
&\equiv \quad \{ \text{ definition of } \mathit{reverse} \ \} \\
&\mathit{head}\ \bot : \bot \\
&\equiv \quad \{ \text{ definition of } \mathit{head} \ \} \\
&\bot : \bot
\end{aligned}
$$

Although the function *withShapeOf* might look somehow artificial, Chitil (2005) uses an equivalent function, called *copyListStructure*, to reimplement an imperative pretty printing algorithm in a functional language. By cleverly employing the function *copyListStructure*, the resulting algorithm has the same complexity as the imperative algorithm (Wadler 2003).

Finally, note that there are other functions that show a similar behavior as *reverse*. For example, a function that takes a binary tree and yields the list of elements in the tree in in-order shows a similar behavior. A minimally strict implementation of this function is supposed to yield the first list constructor even if it is applied to a tree whose left children is undefined.

As we have observed in the previous sections, there are minimally strict implementations that use a non-constant factor less space than their unnecessarily strict counterparts with respect to a specific context as it is the case for *intersperse* and *splitWhen*. Here, we will argue that there always exists a context such that a less strict implementation uses a non-constant factor less space than the unnecessarily strict implementation.

So, which context might be a candidate for a non-constant improvement? The counter-example that shows that a function is unnecessarily strict provides a context in which the less strict implementation uses less space than the original implementation. For example, *reverse* is not minimally strict while $\mathit{reverse}'$ is minimally strict because we have

$$\mathit{reverse}\ (\bot : \bot) \equiv \bot$$

but

$$\mathit{reverse}'\ (\bot : \bot) \equiv \bot : \bot.$$

As context we consider a function, that yields $\bot$ if we apply it to the result of the unnecessarily strict implementation. In contrast, the function yields the empty tuple, denoted by $()$, if we apply it to the result of the less strict implementation. For example, in the case of *reverse* we consider the following function *context*.

$$
\begin{aligned}
&\mathit{context} :: [\alpha] \to () \\
&\mathit{context}\ (\_ : \_) = ()
\end{aligned}
$$

Then we have

$$\mathit{context}\ (\mathit{reverse}\ (\bot : \bot)) \equiv \bot$$

but

$$\mathit{context}\ (\mathit{reverse}'\ (\bot : \bot)) \equiv ().$$

Now we look for an argument that is more defined than $\bot : \bot$ such that both applications are evaluated to $()$. For example, we have

$$context\ (reverse\ (\bot : [\,])) \equiv ()$$

and

$$context\ (reverse'\ (\bot : [\,])) \equiv ().$$

This way we observe that, in comparison to *reverse*′, the function *reverse* inspects a larger part of its argument to yield the same result. More precisely, *reverse* evaluates the second argument of the cons constructor, in this case the empty list, to yield the first list constructor as result. Note that in Chapter 1 we characterized the less-strict relation exactly this way, that is, by referring to the inspected part of an argument.

Next, we define a function that yields the first argument, namely, $\bot : \bot$, in constant space while it needs linear space in the size of some integer to yield the more defined argument, in this case $\bot : [\,]$. For example, we can use the following definition, which is inspired by the motivating example by Gustavsson and Sands (1999).

$$\begin{aligned}
&argument :: Int \to [\alpha] \\
&argument\ n = \bot : \mathbf{if}\ y == x\ \mathbf{then}\ \bot\ \mathbf{else}\ [\,] \\
&\quad \mathbf{where} \\
&\qquad xs = [0\,..\,n] \\
&\qquad x = head\ xs \\
&\qquad y = last\ xs
\end{aligned}$$

We assume that n is always greater or equal to one. Furthermore, in this context by $\bot$ we always denote a run-time error and not a non-terminating expression as the evaluation of a non-terminating expression might result in space usage while this is not the case for a run-time error. Finally, the following considerations are based on the fact that the equality check $(==)$ on integers first evaluates its first argument.

To evaluate the expression $\mathbf{if}\ y == x\ \mathbf{then}\ \bot\ \mathbf{else}\ [\,]$ we first have to evaluate y due to the left to right evaluation order of $(==)$. This causes the evaluation of the whole list structure of xs and, therefore, occupies linear space in the size of n. The garbage collector cannot deallocate the occupied memory as we hold a reference to xs by means of x.

Now we consider the following application.

$$context\ (reverse'\ (argument\ n))$$

The evaluation of this application has a constant space usage as *reverse*′ does not evaluate the conditional. In contrast, the evaluation of

$$context\ (reverse\ (argument\ n))$$

causes the evaluation of the expression $\mathbf{if}\ y == x\ \mathbf{then}\ \bot\ \mathbf{else}\ [\,]$. The evaluation of this expression uses a linear amount of space in the size of n. Therefore, by considering an arbitrary large integer n, we get a context in which *reverse* uses an arbitrary

context	*reverse*	*reverse′*	*reverse″*
null	1.90s / 460MB	0.001s / 44KB	0.001s / 44KB
head	2.65s / 460MB	0.41s / 28KB	2.65s / 460MB
double	2.99s / 460MB	>1min / >2GB	10.29s / 2GB

Figure 7.5.1: Run-Times and Space Usage for the Evaluation of an Application of a Reverse Function to $[1\,..\,2 * 10\hat{}7]$ in the Corresponding Context.

large amount of space while *reverse′* uses only a constant amount of space. The same holds if we use *reverse″* instead of *reverse′*.

In the same way we can derive a context in which an arbitrary less strict function is an improvement over its unnecessarily strict counterpart. For example, for *intersperse* we can use a quite similar construction as for *reverse*. More precisely, the evaluation of the application

$$context\ (intersperse\ \bot\ (argument\ n))$$

uses linear space while the evaluation of

$$context\ (intersperse'\ \bot\ (argument\ n))$$

uses only constant space. Obviously, as future work we have to formally verify these statements by using appropriate semantic models.

Instead of the artificial argument *argument n* we can also use a more natural one, namely, $[1\,..\,n]$. That is, we do not have to attribute the linear amount of space to the evaluation of the empty list, but we can as well attribute it to the evaluation of a linear-sized list structure. Note that there are examples where we indeed have to attribute a linear amount of space to the evaluation of a single constructor. Thus, we have used the function *argument* above to illustrate that it is possible to attribute an arbitrary amount of space usage to the evaluation of a single constructor. Furthermore, instead of the context *context*, we can as well use the function *null* that checks whether its argument is the empty list. Figure 7.5.1 presents run-times and space usages if we apply *null* to applications of *reverse*, *reverse′*, and *reverse″* to the list $[1\,..\,2 * 10\hat{}7]$. Here, the term $[1\,..\,2 * 10\hat{}7]$ denotes the list with the integers from one to 20 million. The numbers in Figure 7.5.1 together with results for smaller lists not presented here confirm that the unnecessarily strict implementation uses a linear amount of space in the length of the list, while the less strict implementations *reverse′* and *reverse″* use only a constant amount of space.

Now let us consider another context, namely, *head*. As Figure 7.5.1 shows, the space usage of *reverse′* is quite good if we consider the context *head*, because the application $head\ (reverse'\ xs)$ reduces to $last\ xs$ and $last\ [1\,..\,n]$ runs in constant space for all n. In contrast, the application $head\ (reverse\ xs)$ constructs the reversed list completely. Therefore, *reverse′* also uses a non-constant factor less space than *reverse* if we consider the context *head*. In contrast to *reverse′*, the application $head\ (reverse''\ xs)$ reduces to $head\ (reverse\ xs)$ and, thus, the space usage of $reverse''\ [1\,..\,n]$ is linear in n as

it is the case for *reverse*. In the case of the context *head* the difference between $reverse'$ and $reverse''$ with respect to space usage is not reflected by the less-strict ordering because $reverse'$ and $reverse''$ are denotationally equivalent.

However, there even exists a context in which the space usage of $reverse'$ as well as the space usage of $reverse''$ is quadratic in the size of the argument list while the space usage of *reverse* is linear in the size of the argument. We consider the function *double*. Its definition uses the predefined *length* function, which uses a strict evaluation primitive to evaluate intermediate results. If we do not evaluate intermediate results, *length* itself causes a space leak.

$$\begin{aligned} &double :: [\alpha] \to Int \\ &double\ ys = length\ (ys +\!\!+ ys) \end{aligned}$$

When we evaluate the expression $double\ (reverse'\ xs)$, the spine of $reverse'\ xs$ is evaluated by the application of *length*. Furthermore, *length* does not evaluate the elements of the list. Therefore, *ys* is evaluated to a structure, that can be illustrated by the following expression.

$$last\ xs : last\ (init\ xs) : last\ (init\ (init\ xs)) : \ldots$$

The applications of *last* and *init* to the list *xs* are unevaluated thunks. The garbage collector cannot collect this structure because in *double* we keep a reference to the list *ys* in the second argument of $(+\!\!+)$. As this example illustrates, we get a quadratic number of thunks in the length of the list. According to the space model by Gustavsson and Sands (1999), a quadratic number of thunks occupies a quadratic amount of space. Thus, the evaluation of the application $double\ (reverse'\ [1\,..\,n])$ uses a quadratic amount of space in n.

The evaluation of the application $double\ (reverse''\ [1\,..\,n])$ also uses a quadratic amount of space. In the case of $reverse''$, unevaluated applications of *last* and *init* become unevaluated projections to the head and the tail of the list *reverse xs*. As these projections are never evaluated, we cannot benefit from the optimization proposed by (Wadler 1987) in this case.

This example demonstrates that the theory and practice of minimally strict functions lacks a more thorough investigation of space usage of minimally strict functions. Furthermore, it might be advantageous to aim for implementations that are minimally strict with respect to a specific context and not minimally strict with respect to all possible contexts. For example, if we check the function $double \circ reverse :: [A] \to Int$, Sloth does not report any counter-examples. That is, if we use the context *double* we do not have to alter *reverse* because it is already minimally strict with respect to this context. However, if we consider the context *null*, that is, if we check whether $null \circ reverse :: [A] \to Bool$ is minimally strict, we observe that it is not. Nevertheless, checking whether the result of *reverse* is the empty list is an quite unusual application as we can perform the same check on the argument of *reverse*. Though, note that abstraction can still lead to this kind of application. For example, in an implementation of efficient queues by means of two lists as presented by Okasaki (1998) we have to regularly check whether the result of an application of *reverse* is

the empty list. In summary, to provide the best implementation we might have to evaluate which kinds of contexts are the most common ones.

8 Conclusion

In the first part of this chapter we summarize the most important results and observations of the previous sections. While we already provide some ideas for future work in this first part, in the second part of this chapter we discuss possibilities for more sophisticated future work within the area of functional programming as well as an extension to the area of functional logic programming.

8.1 Summary

We have started by motivating why less strict functions matter. More precisely, in Section 3.2 we have considered the function *intersperse* from the standard Haskell library *Data.List*. We have observed that *intersperse* can cause a space leak and that this leak is due to *intersperse* being unnecessarily strict. By deferring a single pattern matching of *intersperse*, we have decreased the space usage of a program that replaces an umlaut in Shakespeare's collected works by a factor of 20,000.

Then, in Section 4.1 we have observed that the current approach to unnecessarily strict functions by Chitil (2006) proposes refactorings that have to use non-pure features like non-monadic concurrency. For example, the standard implementation of the list concatenation $(+\!\!+)$ is not least strict in the sense of Chitil (2006). A least strict concatenation is supposed to satisfy $[1] +\!\!+ \bot \equiv 1 : \bot$ as well as $\bot +\!\!+ [1] \equiv \bot : \bot$ (Chitil 2011). The standard Haskell implementation of $(+\!\!+)$ satisfies the former but not the latter requirement. Moreover, there is no implementation that satisfies both requirements without using non-pure features like concurrency.

Once we are convinced that we do not want to implement functions like list concatenation using non-pure features, we have to refine the notion of least strict. A function is least strict if there exists no less strict function, where the notion of function is defined in the sense of a denotational semantics, that is, we consider monotonic and continuous functions on corresponding domains. In contrast, in Section 4.2 we state that a function is minimally strict if there exists no less strict function that is also sequential. That is, we consider only functions that are monotonic, continuous, *and* sequential. In this way we exclude functions that have to use non-pure features.

Using this approach we define that a function is minimally strict if there exists no less strict, sequential function. In the context of sequential functions we use the term "minimally strict" and not "least strict" because, regarding sequential functions, there does not always exist a least element with respect to the less-strict ordering. For example, while the parallel and is least strict, there are two implementations of the Boolean conjunction that are minimally strict, one that performs pattern matching on its first argument, sometimes called left-strict, and one that performs pattern matching on its second argument, sometimes called right-strict.

Checking manually whether the proposed improvements with respect to strictness, like the improvements proposed for $(+\!\!+)$, result in a non-sequential implementation is quite challenging. Therefore, in Section 4.3 we have presented a first step towards a more detailed investigation, namely, a criterion to check whether there exists a less strict, sequential function. Moreover, we have proved that this criterion is necessary and sufficient for the existence of a less strict, sequential function. This criterion is defined for monomorphic, first-order functions, only. While we have later considered an extension to polymorphic functions we did not examine higher-order functions.

This criterion provides a basis for further investigations into checking whether functions are minimally strict. First of all, it allows us to formally argue whether a function is minimally strict by a more mechanical approach than using the definition of sequentiality. Furthermore, the criterion provides the means to develop new tools to check whether a function is minimally strict. For instance, while we have presented a tool that regards only the "semantics" of a function in form of an higher order argument, we can use the criterion to implement more sophisticated tools that incorporate the syntactical definition of a function.

As indicated, on the basis of the formal criterion we have implemented a lightweight tool, called Sloth, that checks whether a function is unnecessarily strict for arguments up to a given size. Here, size refers to the number of constructors of a term. In Chapter 5 we have presented several optimizations to a naive implementation of the tool that are used to improve the practical applicability. Nevertheless, we consider Sloth a prototype as it still has some limitations.

Most notably, testing a function that uses a more sophisticated data type can result in a quite large number of potential counter-examples that are no counter-examples if we consider arguments of larger sizes. As an extreme example consider the data type *Char*. Conceptually, this data type has 1114112 different constructors. The current implementation of Sloth considers only the 128 most common characters. The size, that is, the number of constructors, of any element of type *Char* is one. Thus, we would already generate test cases for all characters if we check a function for arguments up to size one. Therefore, we treat characters, as well as well as integers, differently. Namely, we consider these data types in the manner proposed by Runciman et al. (2008). The size of an element of type *Int* is the absolute value of the integer. In contrast, the size of an element of type *Char* is the number of bits of its integer representation. Hence, the number of test cases of type *Char* grows exponentially in the size while the number of test cases of type *Int* grows linearly in the size. Although this choice is fixed right now, it would be desirable to have control over it, for example, to be able to get an exponential growth for integers, as well.

However, this kind of enumeration of primitive data types causes other problems. Whether a given function is minimally strict or not might depend on one specific test case. Consider, for example, the function *lines*, which breaks up a string into a list of strings at all positions of a newline character. To observe that this function is minimally strict we have to generate a test case that contains the character `'\n'`. Depending on the order of the generation of characters, we might have to generate all other characters before we generate a newline character. For all strings that do not contain a newline character, *lines* yields a list with a single element. Thus, if

we only consider strings that do not contain the character `'\n'`, it looks as if *lines* yields a list with a single element in all cases. Therefore, we might wrongly attribute the function *lines* as unnecessarily strict as it could yield a list with one element without inspecting its argument. More precisely, for every possible partial value, the function *lines* is classified as potentially unnecessarily strict as long as we do not consider a more defined argument that contains a newline character.

For example, with the current implementation of Sloth we have to check *lines* for strings up to size nine to observe that the application *lines* $(\bot : \bot)$ is not unnecessarily strict. When we consider arguments up to size nine, we, among other test cases, generate the test case `'\n'` $: \bot$ and, therefore, observe that the application *lines* $(\bot : \bot)$ is not unnecessarily strict. However, in this case, Sloth reports potential counter-examples for all applications of *lines* to arguments that are more defined than $\bot : \bot$, like `'a'` $: \bot$, as Sloth does not generate a test case that is more defined than `'a'` $: \bot$ and contains a newline character. In cases like this an interactive presentation of counter-examples, like it is used by StrictCheck (Chitil 2006), is advantageous. We have implemented a function called *interactCheck* that presents counter-examples one at a time and only presents the next counter-example if demanded. By employing this function we are able to only consider the most reliable counter-example. Sloth first presents the counter-examples based on the largest number of values for the corresponding infimum.

The problem of large numbers of potential counter-examples is not restricted to functions that use primitive data types like *Char* and *Int*. We can get similar behavior for functions that use ordinary algebraic data types with several constructors. To provide a more elaborated evaluation of the practical applicability of Sloth, we have to collect more data, that is, apply Sloth to more examples. Additionally, *lines* demonstrates another problem, we are faced with. Namely, we might have to generate a quite large number of test cases to observe that one specific application is not unnecessarily strict. Testing a function for a large number of test cases is time as well as space consuming or can even be infeasible.

To reduce the number of test cases we would like to employ background knowledge by the user. For example, to check the function *lines* it would be sufficient to consider strings that consist only of the undefined character $\bot$, an arbitrary defined character like `'a'`, and the special newline character `'\n'`. The function *lines* handles all other defined characters in the same way as it handles the character `'a'`. With this knowledge we may define a test case generator that only generates these kinds of strings. Thus, user defined test case generators are a topic for future work. In the area of property-based testing there are even approaches to automatically reduce the number of test cases in this way, using a criterion for code coverage. In other words, a test case is not considered if there exists another test case that covers the same code with respect to some kind of criterion. For example, Gill and Runciman (2007) have presented an approach for displaying code coverage while Fischer and Kuchen (2007) use code coverage to automatically reduce the number of test cases.

In Chapter 6 we have extended the less-strict relation to polymorphic functions. Furthermore, we have presented an approach that is related to the reduction of test cases for *lines*, as presented above. Namely, we use additional knowledge, provided by the type of a function, to reduce the number of test cases for polymorphic func-

tions. If a function is polymorphic, it basically cannot perform pattern matching on the polymorphic component. We have proved that we can check whether a polymorphic function is minimally strict by checking its monomorphic integer instance. We don't even have to check the polymorphic function for all possible inputs of the monomorphic integer instance but only for a small number of test cases. More precisely, we have to check only a linear number of test cases in the number of polymorphic components in the data structure. For example, to test a polymorphic function that takes lists as arguments we have to check only a linear number of test cases in the length of the list. We have integrated this approach to testing polymorphic function into Sloth and we can easily check polymorphic functions efficiently using a simple type annotation.

Although, as illustrated, interpreting potential counter-examples for complex data types is problematic in some cases, even if we consider quite simple data types like lists and trees, there are lots of interesting examples of unnecessarily strict functions. As an example, the innocent looking function *reverse* is, somehow surprisingly, not minimally strict. Furthermore, at least we would not have guessed that there are any unnecessarily strict functions in a standard Haskell library like *Data.List* or that a well-known function like Peano multiplication is unnecessarily strict. Furthermore, in Section 7.4 we have illustrated that we can use Sloth to improve the memory behavior of a real world Haskell package, namely, the package split. Finally, in Section 7.5, we have illustrated that we can always derive a context in which an unnecessarily strict implementation performs worse than the corresponding less strict implementation. This result implies that a less strict implementation is not necessarily better than the original implementation but it is also not worse. All in all, a prototype implementation like Sloth is quite useful for collecting examples of unnecessarily strict functions as we have presented throughout this work.

8.2 Future Work

In this section we present future work, first, in the area of functional programming and, then, in the area of functional logic programming. Note that we do not repeat all statements about future work that have been mentioned in the previous chapters.

8.2.1 Functional Programming

To motivate one direction for future work let us consider a case study by Runciman and Wakeling (1993b) that is "famous" for showing that some space leaks can be fixed by using an implementation that is "more lazy" (Hudak et al. 2007). We have only touched this particular example in Chapter 1 as the notion of "more lazy" with respect to this example is more sophisticated than our notion of less strict. Runciman and Wakeling (1993b) have presented a tool for profiling the memory usage of programs in a lazy functional programming language. Using this tool, they have investigated the memory usage of a program that takes a propositional formula and yields a clausal normal form of this formula. They have observed that the memory usage can be improved by using a "more lazy" implementation of one particular

function. However, in the sense of the less-strict relation, presented here, the proposed improved implementation is incomparable to the original implementation. The function in question yields a list and, with respect to some arguments, the improved implementation changes the order of the elements of the list in comparison to the original implementation. Therefore, these functions are incomparable with respect to the less-strict relation. As the algorithm employs lists as a simple implementation of sets, the order of the elements does not matter. This example demonstrates that we sometimes do not look for a less strict implementation with respect to the canonical semantic ordering that is based on the structure of the result data type but for a less strict implementation with respect to a more abstract ordering. For example, in the case at hand we would like to use an ordering that ignores the order of the elements in the list. This example can be considered as one instance of the more general case of testing whether a function that uses an abstract data type is minimally strict.

To illustrate another related limitation of the less-strict relation we consider the purely functional implementation of a certain on-line algorithm by Bird et al. (1997). By employing a less strict matrix transposition they were able to provide an implementation of the considered on-line algorithm that has linear complexity. In this case a matrix is represented by a list of lists. If we check the original, unnecessarily strict implementation of the transposition, Sloth does not report any counter-examples. In fact, the standard implementation as well as the implementation presented by Bird et al. (1997) are minimally strict. But, again, these functions are incomparable as they yield incomparable results for total arguments. More precisely, these function do not agree for arguments that are outside of the intended domain of the functions. A transposition that takes a list of lists as argument is only well defined if we apply it to a list that contains lists of equal length, as other kinds of lists of lists do not represent valid matrices. We conjecture that these two implementations of transposition become comparable with respect to the less-strict relation, if we consider only valid inputs. Hence, for future work we would like to incorporate a means into Sloth to constrain the considered arguments of a function. A naive way to constrain the arguments is to apply a partial identity to the arguments of a function and map all undesired arguments to $\bot$. However, it is very difficult to implement the corresponding partial identity as we have to regard partial values.

In our opinion, the most important and probably most difficult future work is a connection between the less-strict relation and space usage. As we have illustrated in Section 7.5, we can always construct a context in which a less strict implementation is an improvement over an unnecessarily strict implementation. As a first step, we have to formally verify this informally stated conjecture. Furthermore, we have to verify a second conjecture that is concerned with *reverse*. We assume that there is no minimally strict implementation of *reverse* that is an improvement over the unnecessarily strict implementation with respect to all possible contexts. If we can show that there are functions for which no minimally strict function exists that is an improvement with respect to all contexts, we provide a strong argument for considering only functions that are minimally strict with respect to a specific context.

A syntax-driven approach by means of transformations like **case** deferment might provide an easier access to a connection between memory consumption and strict-

ness. It is much easier to reason about memory behavior of two functions that are related by a transformation like **case** deferment than reasoning about memory behavior of two arbitrary functions that are related by the less-strict relation. If we consider two arbitrary functions that are related by the less-strict relation the functions can behave quite differently from an operational point of view while two functions that are related by the **case** deferment behave operationally quite similar.

Furthermore, we would like to take a closer look at the trade-off between abstraction and minimally strict implementations. In Section 7.3 we have already observed, using the example of the function $(<=)$, that abstraction by means of a default implementation of a type class function can result in an unnecessarily strict implementation. As another example for the trade-off between modularity and a minimally strict implementation, consider the function $sequence :: Monad\ m \Rightarrow [m\ \alpha] \to m\ [\alpha]$. If we consider the list monad instance of this function, it calculates the cross product of an arbitrary number of lists. For example, we have

$$sequence\ [[1],[3,4],[5,6]] \equiv [[1,3,5],[1,3,6],[1,4,5],[1,4,6]].$$

Because *sequence* is implemented generically for all monads, its implementation for the list mond is unnecessarily strict. For example, the evaluation of the expression $sequence\ [[1..],[]]$ does not terminate. In contrast, a minimally strict implementation would yield the empty list instead. Simplified, the reason is that the list instance of *sequence* can check whether any for the elements in the argument list is the empty list while an implementation for an arbitrary monad cannot.

Finally, we could take a quite contrary approach in comparison to the basic idea of minimally strict functions. Instead of considering sequential functions only, we can stick to the concept of least strict functions and consider a more powerful language that makes it possible to define least strict functions. For example, we can employ the Haskell operator *unamb* by Elliott (2009), which is an implementation of the quite intensively studied operator *amb* by McCarthy (1961). However, we assume that this operator causes a significant run-time overhead that cannot be outweighed by the benefits of a least strict definition.

Even more revolutionary, we would like to investigate a more declarative approach to pattern matching in lazy functional programming. As we have touched on in the introduction the current approach to pattern matching in Haskell is rather non-declarative as a programmer specifies the order of evaluation by means of **case** expressions[1]. More precisely, we want to regard the definition of pattern matching only with respect to total values and provide a transformation that looks for the optimal implementation with respect to partial values. In this context we have to mention that there are even more declarative approaches to the transformation of rules into **case** expressions. More precisely, the programming language Curry uses a transformation (Antoy 1992) that can be regarded as more declarative as it abstract over the order of the rules and results in less strict functions.

[1] Rules can be considered as syntactic sugar for **case** expressions.

8.2.2 Functional-Logic Programming

We expect that non-strict functional logic programming languages like Curry are a very interesting field of future work. As we have illustrated in Section 7.3, deterministic functions that are applied to free variables can benefit from less strict implementations. But, we can also consider non-deterministic functions. To be applicable to full Curry we would have generalize the notation of minimally strict functions to non-deterministic functions. We have already presented a denotational semantics for Curry (Christiansen et al. 2011a)[2] that seems adequate for further investigations as it provides a kind of functional view on functional logic programming and, therefore, should allow for a quite natural extension of the less strict relation to non-deterministic functions.

To illustrate that the extension to non-deterministic functions is valuable let us consider the following function, called *insertND*, that non-deterministically inserts an element at any position of a list. The operator $(?)$ denotes a non-deterministic choice of its two arguments. That is, if the argument of *insertND* is a non-empty list $y : ys$, we either insert the element x at the front of $y : ys$ or we insert it non-deterministically in the list ys and put y to the front of the result.

$$\begin{aligned} &\mathit{insertND} :: \alpha \to [\alpha] \to [\alpha] \\ &\mathit{insertND}\ x\ [\,] \quad\ = [x] \\ &\mathit{insertND}\ x\ (y : ys) = (x : y : ys)\ ?\ (y : \mathit{insertND}\ x\ ys) \end{aligned}$$

This is a standard implementation of the non-deterministic insertion function that, for example, is proposed on a webpage with Curry example programs (Hanus).

However, there is another implementation of this function that behaves equally for all total arguments. As overlapping rules induce a non-deterministic choice, we can define one rule that matches in the case that the list is empty as well as in the case that the list is non-empty as follows.

$$\begin{aligned} &\mathit{insertND}' :: \alpha \to [\alpha] \to [\alpha] \\ &\mathit{insertND}'\ x\ ys \quad\ = x : ys \\ &\mathit{insertND}'\ x\ (y : ys) = y : \mathit{insertND}'\ x\ ys \end{aligned}$$

The function *insertND′* is less strict than *insertND* if we consider a less-strict relation that is based on the semantic ordering used to model a denotational semantics for a non-deterministic language (Christiansen et al. 2011a). For example, we have $\mathit{insertND}\ x\ \bot \equiv \bot$ but $\mathit{insertND}'\ x\ \bot \equiv x : \bot$. While in this case the result of both functions is actually kind of deterministic, there are also non-deterministic cases. For example, we have $\mathit{insertND}\ x\ (y : \bot) \equiv (x : y : \bot)\ ?\ (y : \bot)$ and $\mathit{insertND}'\ x\ (y : \bot) \equiv (x : y : \bot)\ ?\ (y : x : \bot)$. Therefore, to state that *insertND′* is less strict than *insertND* we have to regard non-determinism.

Let us consider an implementation of permutation sort, which is a standard example for functional logic programming. If we implement permutation sort by means of these functions we get a much more efficient implementation using the "less strict" implementation of *insertND*. For example, it takes around 20 times as

[2] A corrected version is available as technical report (Christiansen et al. 2011b).

long to sort a list with 15 elements using *insertND* than it takes to perform the same task with *insertND*′.

Furthermore, we can even observe another close relation between this example and a result presented in this thesis. We can derive the implementation *insertND*′ from the implementation *insertND* by means of a generalization of **case** deferment to functional logic programming. More precisely, if we consider an equivalent implementation of *insertND*′ that is defined by means of $(?)$, we can derive *insertND*′ from *insertND* by moving the context $(x : xs)\ ?\ [\cdot]$ over a **case** expression.

Finally, we have also started to investigate free theorems in a functional logic programming language (Christiansen et al. 2010). More precisely, we have provided an example-driven investigation that shows how we can adapt free theorems to the context of functional logic programming. A formalization by means of the semantics presented in (Christiansen et al. 2011a) will allow us to generalize the observations from Chapter 6 to functional logic languages.

A Proofs from Chapter 4

A.1 Proofs from Section 4.2

Proof (of Lemma 4.2.4): We prove this statement by structural induction over the typing rules.

Base Cases: Obviously we have $(\llbracket e \rrbracket_{a_1}^{Err}, \llbracket e \rrbracket_{a_2}) \in err_\tau^m$ for the rules $\Gamma, x :: \tau \vdash x :: \tau$, $\Gamma \vdash \mathit{True} :: \mathit{Bool}$, $\Gamma \vdash \mathit{False} :: \mathit{Bool}$, $\Gamma \vdash \mathit{Nil}_\tau :: \mathit{List}\ \tau$, and $\Gamma \vdash \mathit{undefined}_\tau :: \tau$.

Inductive Cases: We consider the rule for list construction.

$$\frac{\Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \mathit{List}\ \tau}{\Gamma \vdash \mathit{Cons}\langle e_1, e_2 \rangle :: \mathit{List}\ \tau}$$

By induction hypothesis we have $(\llbracket e_1 \rrbracket_{a_1}^{Err}, \llbracket e_1 \rrbracket_{a_2}) \in err_\tau^m$ and $(\llbracket e_2 \rrbracket_{a_1}^{Err}, \llbracket e_2 \rrbracket_{a_2}) \in err_{\mathit{List}\ \tau}^m$. We reason as follows.

$$\begin{aligned}
&(\llbracket e_1 \rrbracket_{a_1}^{Err}, \llbracket e_1 \rrbracket_{a_2}) \in err_\tau^m \wedge (\llbracket e_2 \rrbracket_{a_1}^{Err}, \llbracket e_2 \rrbracket_{a_2}) \in err_{\mathit{List}\ \tau}^m \\
&\quad \Longleftrightarrow (\mathit{Cons}\ \langle \llbracket e_1 \rrbracket_{a_1}^{Err}, \llbracket e_2 \rrbracket_{a_1}^{Err} \rangle, \mathit{Cons}\ \langle \llbracket e_1 \rrbracket_{a_1}, \llbracket e_2 \rrbracket_{a_1} \rangle) \in err_{\mathit{List}\ \tau}^m \\
&\quad \Longleftrightarrow (\mathit{Cons}\ \llbracket \langle e_1, e_2 \rangle \rrbracket_{a_1}^{Err}, \mathit{Cons}\ \llbracket \langle e_1, e_2 \rangle \rrbracket_{a_2}) \in err_{\mathit{List}\ \tau}^m \\
&\quad \Longleftrightarrow (\llbracket \mathit{Cons} \langle e_1, e_2 \rangle \rrbracket_{a_1}^{Err}, \llbracket \mathit{Cons} \langle e_1, e_2 \rangle \rrbracket_{a_2}) \in err_{\mathit{List}\ \tau}^m
\end{aligned}$$

Next we consider the rule for tuple construction.

$$\frac{\Gamma \vdash e_1 :: \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n :: \tau_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle :: \tau_1 \times \cdots \times \tau_n}$$

We have $(\llbracket e_i \rrbracket_{a_1}^{Err}, \llbracket e_i \rrbracket_{a_2}) \in err_{\tau_i}^m$ for all $i \in \{1, \ldots, n\}$ by induction hypothesis and reason as follows.

$$\begin{aligned}
&\forall i \in \{1, \ldots, n\}.\ (\llbracket e_i \rrbracket_{a_1}^{Err}, \llbracket e_i \rrbracket_{a_2}) \in err_{\tau_i}^m \\
&\quad \Longleftrightarrow (\llbracket \langle e_1, \ldots, e_n \rangle \rrbracket_{a_1}^{Err}, \llbracket \langle e_1, \ldots, e_n \rangle \rrbracket_{a_2}) \in err_{\tau_1 \times \cdots \times \tau_n}^m
\end{aligned}$$

In the case of a function application we have to consider the following rule.

$$\frac{\Gamma \vdash f :: \sigma \to \tau \qquad \Gamma \vdash e :: \sigma}{\Gamma \vdash f\ e :: \sigma}$$

By induction hypothesis we have $(\llbracket f \rrbracket^{Err}_{a_1}, \llbracket f \rrbracket_{a_2}) \in err^m_{\sigma\to\tau}$ and $(\llbracket e \rrbracket^{Err}_{a_1}, \llbracket e \rrbracket_{a_2}) \in err^m_\sigma$ and reason as follows.

$$\begin{aligned}
&(\llbracket f \rrbracket^{Err}_{a_1}, \llbracket f \rrbracket_{a_2}) \in err^m_{\sigma\to\tau} \wedge (\llbracket e \rrbracket^{Err}_{a_1}, \llbracket e \rrbracket_{a_2}) \in err^m_\sigma \\
&\quad \Longleftrightarrow \forall (a,b) \in err^m_a.\, (\llbracket f \rrbracket^{Err}_{a_1}\, a, \llbracket f \rrbracket_{a_2}\, b) \in err^m_\tau \wedge (\llbracket e \rrbracket^{Err}_{a_1}, \llbracket e \rrbracket_{a_2}) \in err^m_\sigma \\
&\quad \Longrightarrow (\llbracket f \rrbracket^{Err}_{a_1}\, \llbracket e \rrbracket^{Err}_{a_1}, \llbracket f \rrbracket_{a_1}\, \llbracket e \rrbracket_{a_2}) \in err^m_\tau \\
&\quad \Longrightarrow (\llbracket f\ e \rrbracket^{Err}_{a_1}, \llbracket f\ e \rrbracket_{a_2}) \in err^m_\tau
\end{aligned}$$

We consider the rule for function types.

$$\frac{f :: \sigma \to \tau \in P}{\Gamma \vdash f :: \sigma \to \tau}$$

As the program P is well-typed we have $\Gamma \vdash f :: \sigma \to \tau$. Therefore, by induction hypothesis we have $(\llbracket f \rrbracket^{Err}_{a_1}, \llbracket f \rrbracket_{a_2}) \in err^m_{\sigma\to\tau}$.

We consider the rule for list case expressions.

$$\frac{\Gamma \vdash e :: List\ \tau \qquad \Gamma \vdash e_1 :: \tau' \qquad \Gamma, x :: \tau, xs :: List\ \tau \vdash e_2 :: \tau'}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{Nil_\tau \to e_1; Cons\langle x, xs\rangle \to e_2\} :: \tau'}$$

We have $(\llbracket e \rrbracket^{Err}_{a_1}, \llbracket e \rrbracket_{a_2}) \in err^m_{List\ \tau}$ by induction hypothesis and distinguish five cases.

- If $\llbracket e \rrbracket^{Err}_{a_1} = \bot_{List\ \tau}$, by definition of **case** we have to show that

  $$(\bot_{\tau'}, \llbracket \mathbf{case}\ e\ \mathbf{of} \ldots \rrbracket_{a_2}) \in err^m_{\tau'}.$$

  This statement is true because for all types τ we have $(\bot_\tau, v) \in err^m_\tau$ for all $v \in \llbracket \tau \rrbracket$.

- If $\llbracket e \rrbracket^{Err}_{a_1} = Error_{List\ \tau}\ n$ and $m \neq n$ by definition of **case** we have to show that

  $$(Error_{\tau'}\ n, \llbracket \mathbf{case}\ e\ \mathbf{of} \ldots \rrbracket_{a_2}) \in err^m_{\tau'}.$$

  If $m \neq n$ for all types τ we have $(Error_\tau\ n, v) \in err^m_\tau$ for all $v \in \llbracket \tau \rrbracket$.

- If $\llbracket e \rrbracket^{Err}_{a_1} = Error_{List\ \tau}\ m$ and $\llbracket e \rrbracket_{a_2} = \bot_{List\ \tau}$ by definition of **case** we have to show that $(Error_{\tau'}\ m, \bot_{\tau'}) \in err^m_{\tau'}$ which is true for all types τ'.

- If $\llbracket e \rrbracket^{Err}_{a_1} = Nil_\tau$ and $\llbracket e \rrbracket_{a_2} = Nil_\tau$ by definition of **case** we have to show that $(\llbracket e_1 \rrbracket^{Err}_{a_1}, \llbracket e_1 \rrbracket_{a_2}) \in err^m_{\tau'}$ which is given by induction hypothesis.

- If $\llbracket e \rrbracket^{Err}_{a_1} = Cons\ \langle v_1, v_2\rangle$ and $\llbracket e \rrbracket_{a_2} = Cons\ \langle v_3, v_4\rangle$ with $(v_1, v_3) \in err^m_\tau$ and $(v_2, v_4) \in err^m_{List\ \tau}$ by definition of **case** we have to show that

  $$(\llbracket e_2 \rrbracket^{Err}_{a_1[x\mapsto v_1, xs\mapsto v_2]}, \llbracket e_2 \rrbracket_{a_2[x\mapsto v_3, xs\mapsto v_4]}) \in err^m_{\tau'}$$

  which is given by induction hypothesis.

The proof for Boolean case expressions is analogous to the proof for list case expressions. □

Proof (of Lemma 4.2.5): Because p is a demanded position in pv at position rp, there exists $ev \in [\![\sigma]\!]^{Err}$ such that $(ev, pv) \in err^n_\sigma$. Furthermore, position p is the only position such that

$$ev|_p = Error\ n$$

and

$$([\![f]\!]^{Err} ev)|_{rp} = Error\ n.$$

To show that p is a sequential position in pv at position rp with respect to $[\![f]\!]$ we first have to show that $([\![f]\!]\ pv)|_{rp} = \bot$. We have $(ev, pv) \in err^n_\sigma$, and by Lemma 4.2.4 we get $([\![f]\!]^{Err}, [\![f]\!]) \in err^n_\tau$ and, thus, $([\![f]\!]^{Err} ev, [\![f]\!]\ pv) \in err^n_\tau$. By definition of demanded positions we get

$$([\![f]\!]^{Err}\ ev)|_{rp} = Error\ n$$

and get

$$([\![f]\!]\ pv)|_{rp} = \bot$$

because $([\![f]\!]^{Err}\ ev, [\![f]\!]\ pv) \in err^n_\tau$.

Finally, we have to show that for all $pv' \in [\![\sigma]\!]$ the following holds.

$$pv' \sqsupseteq pv \wedge pv'|_p = \bot \Longrightarrow ([\![f]\!]\ pv)|_{rp} = \bot$$

Let $pv' \in [\![\sigma]\!]$ such that $pv' \sqsupseteq pv$ and $pv'|_p = \bot$. As position p of ev contains an exception, that is, $ev|_p = Error\ n$, by definition of demanded positions, p is the only position that contains the error labeled with n. This implies $(ev, pv') \in err^n_\sigma$. By Lemma 4.2.4 we have $([\![f]\!]^{Err}, [\![f]\!]) \in err^n_\tau$ and, thus, $([\![f]\!]^{Err}\ ev, [\![f]\!]\ pv') \in err^n_\tau$. Furthermore, as

$$([\![f]\!]^{Err}\ ev)|_{rp} = Error\ n$$

we get

$$([\![f]\!]\ pv')|_{rp} = \bot$$

because $([\![f]\!]^{Err}\ ev, [\![f]\!]\ pv') \in err^n_\tau$. This finally completes the proof that p is a sequential position in pv at position rp. □

A.2 Proofs from Section 4.3.1

Proof (of Lemma 4.3.1): As f is sequential, there exists a sequential position p' in v at result position rp. We have $p' \not\leq p$ and $p \not\leq p'$. By definition of sequential position we have $v|_{p'} = \bot$ and, therefore, $(v[v']_p)|_{p'} = v|_{p'} = \bot$ by Lemma 4.2.2. Finally, because $v[v']_p \sqsupseteq v[\bot]_p = v$ and p' is a sequential position in v we get $(f\ v[v']_p)|_{rp} = \bot$. □

Proof (of Lemma 4.3.2): Because $v \neq \bot$, there exists $v' \in [\![\tau]\!]$ and $p \in Pos\, v'$ such that $v' \lessdot_p v$. By monotonicity there exists $rp' \in Pos\,(f\, v')$ such that $(f\, v')|_{rp'} = \bot$ and $rp' \leq rp$. Because we have $(f\, v')|_{rp'} = \bot$, we have either $v' \rightsquigarrow_{rp'} v$ or $v' \rightarrowtail_{rp'} v$. If we have a non-sequential step $v' \rightsquigarrow_{rp'} v$, by Lemma 4.3.1 we have $(f\, v)|_{rp'} = \bot$, and by definition of $\sqsubseteq$ we get $rp = rp'$. □

Lemma A.2.1: *If we have $v_1 \rightsquigarrow_{rp_1} v_2$ and $(f\, v_2)|_{rp_2} = \bot$ with $rp_1 \leq rp_2$, then $rp_1 = rp_2$.*

Proof: We have $v_2 = v_1[C\ \bot]_p$ for some constructor C and a non-sequential position p. By Lemma 4.3.1 we have $(f\, v_2)|_{rp_1} = (f\, v_1[C\ \bot]_p)|_{rp_1} = \bot$. By definition of $\sqsubseteq$ we get $rp_1 = rp_2$. □

Lemma A.2.2: *If $v_1 \rightsquigarrow_{rp} v_2 \rightarrowtail_{rp} v_3$ and $v_1 \lessdot_p v_2 \lessdot_{p'} v_3$ then $p \not\leq p'$.*

Proof: As $v_1 \rightsquigarrow_{rp} v_2$ there exists $v \in [\![\tau]\!]$ with $v \sqsupseteq v_1$, $v|_p = \bot$ and $(f\, v)|_{rp} \neq \bot$. We have $v = v[\bot]_p \sqsubseteq v[v_2|_p]_p$ by monotonicity of $v[\cdot]_p$ and by monotonicity of f we have

$$\bot \neq (f\, v)|_{rp} \sqsubseteq (f\, v[v_2|_p]_p)|_{rp},$$

that is, $(f\, v[v_2|_p]_p)|_{rp} \neq \bot$. By monotonicity of $\cdot[v_2|_p]_p$ we have $v_2 = v_1[v_2|_p]_p \sqsubseteq v[v_2|_p]_p$. If $p \leq p'$ we have $v[v_2|_p]_p|_{p'} = v_2|_{p'}$ by Lemma 4.2.2 and, therefore, p' is a non-sequential position in v_2 at position rp. This is a contradiction as p' is a sequential position in v_2 at position rp. That is, we have $p \not\leq p'$. □

Proof (of Lemma 4.3.3): There exist positions p and p' such that $v_1 \lessdot_p v_2$ and $v_2 \lessdot_{p'} v_3$. Because we have $v_1 \lessdot_p v_2$ there exists a constructor C such that $v_2|_p = C\ \bot$ and $v_1|_p = \bot$. Because we have $v_2 \lessdot_{p'} v_3$ there exists a constructor D such that $v_3|_{p'} = D\ \bot$ and $v_2|_{p'} = \bot$. By Lemma A.2.2 we have $p \not\leq p'$ and $p' \not\leq p$ and set $v := v_1[D\ \bot]_{p'}$. We get $v[C\ \bot]_p = v_3$ and, thus, $v_1 \lessdot_{p'} v$ and $v \lessdot_p v_3$.

We have to show that p' is a sequential position in v_1 at position rp_1. That is, we have to show that for all $v' \in [\![\tau]\!]$ the following holds.

$$v' \sqsupseteq v_1 \wedge v'|_{p'} = \bot \Longrightarrow (f\, v')|_{rp_1} = \bot$$

Let $v' \in [\![\tau]\!]$ with $v' \sqsupseteq v_1$ and $v'|_{p'} = \bot$. We distinguish two cases.

Case 1 $(v' \sqsupseteq v_2)$**:** Because $v' \sqsupseteq v_2$, $v'|_{p'} = \bot$, and p' is a sequential position in v_2 at position rp_1, we have $(f\, v')|_{rp_1} = \bot$.

Case 2 $(v' \not\sqsupseteq v_2)$**:** We consider the values $v_1[v'|_p]_p$ and $v'[C\ \bot]_p$. We have

$$v_1 = v_1[\bot]_p \sqsubseteq v_1[v'|_p]_p$$

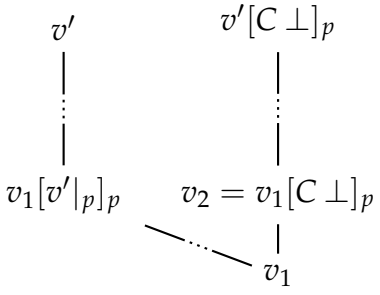
by monotonicity of $v_1[\cdot]_p$,

$$v_1[v'|_p]_p \sqsubseteq v'[v'|_p]_p = v'$$

by monotonicity of $\cdot[v'|_p]_p$, and

$$v_2 = v_1[C\ \bot]_p \sqsubseteq v'[C\ \bot]_p$$

by monotonicity of $\cdot[C\ \bot]_p$. The following Hasse diagram depicts the situation.



In the following we consider the value $v'[\bot]_p$, which is both less equal $v'[C\ \bot]_p$ and less equal v'. As $p \not\leq p'$ and $p' \not\leq p$ we have $(v'[C\ \bot]_p)|_{p'} = v'|_{p'}$ by Lemma 4.2.2. Because we have $v'[C\ \bot]_p \sqsupseteq v_2$ and p' is a sequential position in v_2 we get $(f\ v'[C\ \bot]_p)|_{rp_1} = \bot$. Furthermore, by monotonicity we get $(f\ v'[\bot]_p)|_{rp_1} = \bot$ as well.

Next we show that we can decompose v' into $v'[\bot]_p$ and $v_1[v'|_p]_p$.

$$\begin{aligned}
v'[\bot]_p \sqcup v_1[v'|_p]_p &= (v' \sqcup v_1)[\bot \sqcup v'|_p]_p && \text{continuity of } \cdot[\cdot]_p \\
&= v'[v'|_p]_p && v_1 \sqsubseteq v' \text{ and } \bot \sqsubseteq v' \\
&= v' && \text{property of } \cdot|_p \text{ and } \cdot[\cdot]_p
\end{aligned}$$

We use this decomposition and continuity to show that we have $(f\ v')|_{rp_1} = \bot$.

$$\begin{aligned}
(f\ v')|_{rp_1} &= (f\ (v'[\bot]_p \sqcup v_1[v'|_p]_p))|_{rp_1} && \text{decomposition} \\
&= (f\ v'[\bot]_p \sqcup f\ v_1[v'|_p]_p)|_{rp_1} && \text{continuity of } f \\
&= (f\ v'[\bot]_p)|_{rp_1} \sqcup (f\ (v_1[v'|_p]_p)|_{rp_1} && \text{continuity of } \cdot|_{rp_1} \\
&= (f\ v'[\bot]_p)|_{rp_1} \sqcup \bot && \text{Lemma 4.3.1} \\
&= \bot \sqcup \bot && \text{argumentation above} \\
&= \bot
\end{aligned}$$

This shows that p' is a sequential position in v_1, that is, $v_1 \rightarrowtail_{rp_1} v$.

We finally have to show that either $v \rightsquigarrow_{rp_2} v_3$ or there exists rp_2 with $rp_1 \leq rp_2 \leq rp_3$ and $v \rightarrowtail_{rp_2} v_3$. Because we have $(f\ v_1)|_{rp_1} = \bot$ and $(f\ v_3)|_{rp_3} = \bot$ by monotonicity there exists rp_2 with $rp_1 \leq rp_2 \leq rp_3$ and $(f\ v)|_{rp_2} = \bot$. This implies either $v \rightsquigarrow_{rp_2} v_3$ or $v \rightarrowtail_{rp_2} v_3$. By Lemma A.2.1 we have $rp_2 = rp_3$ if $v \rightsquigarrow_{rp_2} v_3$. □

Proof (of Lemma 4.3.4): By induction over n we show that for all $n \in \mathbb{N}_0$ if

$$v_1(\rightsquigarrow_{rp_1})^n v_2 \rightarrowtail_{rp_1} v_3$$

then

$$v_1 \rightarrowtail_{rp} v \rightarrowtail^* v'(\rightsquigarrow_{rp'})^* v_3.$$

In the induction hypothesis we assume that the statement holds for all $m \in \mathbb{N}_0$ with $m \leq n$.

Base Case: We have $v_2 \rightarrowtail_{rp_1} v_3$ and set $v := v_3$ and $v' := v_3$.

Inductive Case: We have $v_1(\rightsquigarrow_{rp_1})^{n+1} v_2 \rightarrowtail_{rp_1} v_3$. There exists $v \in [\![\tau]\!]$ such that $v_1(\rightsquigarrow_{rp_1})^n v \rightsquigarrow_{rp_1} v_2 \rightarrowtail_{rp_1} v_3$. We consider the sub-sequence $v \rightsquigarrow_{rp_1} v_2 \rightarrowtail_{rp_1} v_3$. By Lemma 4.3.3 there exists $v_2' \in [\![\tau]\!]$ such that $v \rightarrowtail_{rp_1} v_2' \rightsquigarrow_{rp_3} v_3$ or there exists rp_2 with $rp_1 \leq rp_2 \leq rp_3$ such that $v \rightarrowtail_{rp_1} v_2' \rightarrowtail_{rp_2} v_3$. We distinguish these two cases.

Case 1 ($v \rightarrowtail_{rp_1} v_2' \rightsquigarrow_{rp_3} v_3$)**:** We have a sequence of the form

$$v_1(\rightsquigarrow_{rp_1})^n v \rightarrowtail_{rp_1} v_2'.$$

As we have $(f\, v_2')|_{rp_3} = \bot$ by induction hypothesis there exist $v', v'' \in [\![\tau]\!]$ such that

$$v_1 \rightarrowtail_{rp_1} v' \rightarrowtail^* v''(\rightsquigarrow_{rp_3})^* v_2',$$

which is montone with respect to result positions. Finally we get the monotonic sequence

$$v_1 \rightarrowtail_{rp_1} v' \rightarrowtail^* v''(\rightsquigarrow_{rp_3})^* v_3.$$

Case 2 ($v \rightarrowtail_{rp_1} v_2' \rightarrowtail_{rp_2} v_3$)**:** We have a sequence of the form

$$v_1(\rightsquigarrow_{rp_1})^n v \rightarrowtail_{rp_1} v_2'.$$

As we have $(f\, v_2')|_{rp_2} = \bot$ by induction hypothesis there exist $v', v'' \in [\![\tau]\!]$ such that

$$v_1 \rightarrowtail_{rp_1} v' \rightarrowtail^* v''(\rightsquigarrow_{rp_2})^* v_2',$$

which is montone with respect to result positions. Furthermore, there exists $m \in \mathbb{N}_0$ such that

$$v''(\rightsquigarrow_{rp_2})^m v_2' \rightarrowtail_{rp_2} v_3.$$

By definition of sequential steps we have $m \leq n - 1$. Because we have $(f\, v_3)|_{rp_3} = \bot$, by induction hypothesis there exist $v''', v'''' \in [\![\tau]\!]$ such that

$$v'' \rightarrowtail_{rp_2} v''' \rightarrowtail^* v''''(\rightsquigarrow_{rp_3})^* v_3.$$

That is, we get a sequence

$$v_1 \rightarrowtail_{rp_1} v' \rightarrowtail^* v'' \rightarrowtail_{rp_2} v''' \rightarrowtail^* v''''(\rightsquigarrow_{rp_3})^* v_3,$$

which is monotonic with respect to result positions.

If we have $v_1(\rightsquigarrow_{rp})^* v_2 \rightarrowtail_{rp} v_3$ there exists $n \in \mathbb{N}_0$ such that $v_1(\rightsquigarrow_{rp_1})^n v_2 \rightarrowtail_{rp_1} v_3$ with $(f\ v_3)|_{rp_3} = \bot$. □

Proof (of Lemma 4.3.5): As we consider only finite values we can prove the statement by induction over the number of constructors in v.

Base Case: For all $(\bot, rp)$ with $(f\ \bot)|_{rp} = \bot$ we have $(\bot, rp) \in C_f$.

Inductive Case: Let $v \in [\![\tau]\!]$ and $rp \in Pos\ (f\ v)$ such that $(f\ v)|_{rp} = \bot$ and $(v, rp) \notin C_f$. By Lemma 4.3.2 there exists (v_1, rp') with $v_1 \rightsquigarrow_{rp} v$ or $v_1 \rightarrowtail_{rp'} v$ and $rp' \leq rp$. We distinguish two cases.

Case 1 $(v_1 \rightsquigarrow_{rp} v)$**:** If $(v_1, rp) \in C_f$ then we set $v' := v_1$ and are finished. If $(v_1, rp) \notin C_f$ because $(f\ v_1)|_{rp} = \bot$ by induction hypothesis there exists $(v_2, rp) \in C_f$ such that $v_2(\rightsquigarrow_{rp})^* v_1$ and, therefore, $v_2(\rightsquigarrow_{rp})^* v$.

Case 2 $(v_1 \rightarrowtail_{rp'} v)$**:** Because $(v, rp) \notin C_f$ we have $(v_1, rp') \notin C_f$. Because $(f\ v_1)|_{rp} = \bot$ by induction hypothesis there exists $(v_2, rp') \in C_f$ such that $v_2(\rightsquigarrow_{rp'})^* v_1$. That is, we have a sequence of the form

$$v_2(\rightsquigarrow_{rp'})^* v_1 \rightarrowtail_{rp'} v.$$

Because $(f\ v)|_{rp} = \bot$ by Lemma 4.3.4 there exist $v', v'' \in [\![\tau]\!]$ such that

$$v_2 \rightarrowtail_{rp'} v' \rightarrowtail^* v''(\rightsquigarrow_{rp})^* v.$$

As this sequence is monotonic with respect to result positions $(v_2, rp) \in C_f$ implies $(v'', rp) \in C_f$. That is, $(v'', rp) \in C_f$ and $v''(\rightsquigarrow_{rp})^* v$. □

A.3 Proofs from Section 4.3.2

Proof (of Lemma 4.3.8): The functions $\overline{f}_{(v,rp)}$ and f agree for total values because we have $\inf_f\ tv = f\ tv$ for all total values tv.

Let $v' \in [\![\tau]\!]$. We show that we have $\overline{f}_{(v,rp)}\ v \sqsupset f\ v$ and $\overline{f}_{(v,rp)}\ v' \sqsupseteq f\ v'$ if $v' \neq v$. We distinguish three cases.

Case 1 $(v' = v)$**:** Because $(v, rp) \in C_f$ we have $(f\ v)|_{rp} = \bot$. Furthermore, because $(\inf_f\ v)|_{rp} \sqsupset \bot$ we have

$$(f\ v)|_{rp} \sqcup (\inf_f\ v)|_{rp} \sqsupset (f\ v)|_{rp}.$$

By continuity of $\cdot|_{rp}$ we get

$$(f\ v \sqcup \inf_f\ v)|_{rp} \sqsupset (f\ v)|_{rp}.$$

Because $(f\ v)[\cdot]_{rp}$ is strictly increasing and by a property of $\cdot|_{rp}$ we get

$$\overline{f}_{(v,rp)}\ v = (f\ v)[(f\ v \sqcup \inf_f\ v)|_{rp}]_{rp} \sqsupset (f\ v)[(f\ v)|_{rp}]_{rp} = f\ v.$$

Case 2 $(v' \sqsupset v)$**:** We have

$$f\ v' \sqcup \inf_f\ v \sqsupseteq f\ v'$$

By monotonicity of $\cdot|_{rp}$ we get

$$(f\ v' \sqcup \inf_f\ v)|_{rp} \sqsupseteq (f\ v')|_{rp}.$$

By monotonicity of $(f\ v')[\cdot]_{rp}$ we get

$$(f\ v')[(f\ v' \sqcup \inf_f\ v)|_{rp}]_{rp} \sqsupseteq (f\ v')[(f\ v')|_{rp}]_{rp}.$$

Finally by a property of $\cdot|_{rp}$ we get

$$(f\ v')[(f\ v')|_{rp}]_{rp} = f\ v'$$

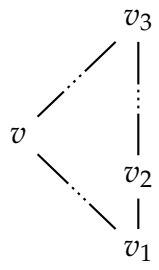
and, therefore, we get

$$\overline{f}_{(v,rp)}\ v' = (f\ v')[(f\ v' \sqcup \inf_f\ v)|_{rp}]_{rp} \sqsupseteq f\ v'.$$

Case 3 $(v' \not\sqsupseteq v)$**:** By definition of $\overline{f}_{(v,rp)}$ we have $\overline{f}_{(v,rp)}\ v' = f\ v'$. □

Lemma A.3.1: *Let $v_1 \in [\![\tau]\!]$, $v_2 \in [\![\tau]\!]$, $v_3 \in [\![\tau]\!]$, and $p \in Pos\ v_1$ a position such that $v_1 \lessdot_p v_2$ and $v_2 \sqsubseteq v_3$. For all $v \in [\![\tau]\!]$ with $v_1 \sqsubseteq v$, $v \sqsubseteq v_3$, and $v_2 \not\sqsubseteq v$ we have $v|_p = \bot$.*

Proof: The following diagram illustrates the situation.



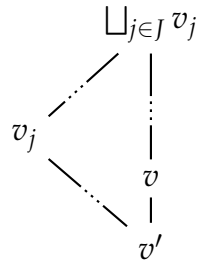
Additionally, v_1 and v_2 are related by $v_1 \lessdot_p v_2$ for some position p. We have $p \in Pos\ v$ because $v_1 \sqsubseteq v$ and $p \in Pos\ v_1$. Because $v_1 \lessdot_p v_2$, we have $v_1|_p = \bot$ and there exists a constructor C such that $v_2|_p = C\ \bot$. As we furthermore have $v_2 \sqsubseteq v_3$, the outermost constructor of v_3 at position p is also C. Because $v \sqsubseteq v_3$, we have $v|_p = \bot$ or the outermost constructor of v at position p is C. Finally, $v \sqsubseteq v_3$ and $v_2 \not\sqsubseteq v$ imply $v|_p = \bot$. □

Proof (of Lemma 4.3.9): If $(v, rp) \in C_f$ there exists a sequence of the form $\bot \rightarrowtail^n v$. By induction on n we show that v is finite.

Base Case $(\bot \rightarrowtail^0 \bot)$**:** Let $\langle v_i \rangle_{i \in I}$ be a chain such that $\bot \sqsubseteq \bigsqcup_{i \in I} v_i$. We have $\bot \sqsubseteq v_i$ for all $i \in I$. Thus, $\bot$ is finite.

Inductive Case $(\bot \rightarrowtail^{n+1} v)$**:** Let $\langle v_i \rangle_{i \in I}$ be a chain with $v \sqsubseteq \bigsqcup_{i \in I} v_i$. There exists v' such that $\bot \rightarrowtail^n v' \rightarrowtail v$. Because $v' \sqsubseteq v$ we have $v' \sqsubseteq \bigsqcup_{i \in I} v_i$. By induction hypothesis there exists $i \in I$ such that $v' \sqsubseteq v_i$. We consider a subset J of I such that $J = \{i \in I \mid v_i \sqsupseteq v'\}$. As there exists at least one element $v_i \sqsubseteq v'$ we have $\bigsqcup_{i \in I} v_i = \bigsqcup_{j \in J} v_j$.

We assume that we have $v \not\sqsubseteq v_j$ for all $j \in J$. Let $j \in J$. We have $v' \lessdot_p v$, $v' \sqsubseteq v_j$, $v \not\sqsubseteq v_j$, and $v_j \sqsubseteq \bigsqcup_{j \in J} v_j$ and $v \sqsubseteq \bigsqcup_{j \in J} v_j$. The following diagram illustrates the situation.



By Lemma A.3.1 we get $v_j|_p = \bot$. That is, we have $v_j|_p = \bot$ for all $j \in J$. Because $\cdot|_p$ is continuous we get

$$(\bigsqcup_{j \in J} v_j)|_p = \bigsqcup_{j \in J} (v_j|_p) = \bot.$$

This is a contradiction to $v \sqsubseteq \bigsqcup_{i \in I} v_i$ as we have $v|_p \neq \bot$. Thus there exists $j \in J$ with $v \sqsubseteq v_j$, that is, v is finite. □

Proof (of Lemma 4.3.10): Let $v_1, v_2 \in [\![\tau]\!]$ with $v_1 \sqsubseteq v_2$. We distinguish two cases.

Case 1 $(v_1 \not\sqsupseteq v)$**:** We reason as follows and employ monotonicity of f and $\overline{f} \prec f$.

$$\overline{f}\, v_1 = f\, v_1 \sqsubseteq f\, v_2 \sqsubseteq \overline{f}\, v_2$$

Case 2 $(v_1 \sqsupseteq v)$**:** Because $v_1 \sqsupseteq v$ and $v_1 \sqsubseteq v_2$ we have $v_2 \sqsupseteq v$. By $v_1 \sqsubseteq v_2$ and monotonicity of f we get

$$f\, v_1 \sqsubseteq f\, v_2.$$

By monotonicity of $\cdot \sqcup \inf_f v$ we get

$$f\, v_1 \sqcup \inf_f v \sqsubseteq f\, v_2 \sqcup \inf_f v$$

and by monotonicity $\cdot|_{rp}$ we have

$$(f\ v_1 \sqcup \inf_f\ v)|_{rp} \sqsubseteq (f\ v_2 \sqcup \inf_f\ v)|_{rp}.$$

By monotonicity of $(f\ v_1)[\cdot]_p$ we get

$$(f\ v_1)[(f\ v_1 \sqcup \inf_f\ v)|_{rp}]_{rp} \sqsubseteq (f\ v_1)[(f\ v_2 \sqcup \inf_f\ v)|_{rp}]_{rp}$$

and by monotonicity of $\cdot[(f\ v_2 \sqcup \inf_f\ v)|_{rp}]_p$ we get

$$(f\ v_1)[(f\ v_2 \sqcup \inf_f\ v)|_{rp}]_{rp} \sqsubseteq (f\ v_2)[(f\ v_2 \sqcup \inf_f\ v)|_{rp}]_{rp}.$$

This final inequality shows that we have $\overline{f}\ v_1 \sqsubseteq \overline{f}\ v_2$. □

Proof (of Lemma 4.3.11): Let $\langle v_i \rangle_{i \in I}$ be a chain in $[\![\tau]\!]$. We have to prove the following equality.

$$\overline{f}_{(v,rp)}\ (\bigsqcup_{i \in I} v_i) = \bigsqcup_{i \in I} (\overline{f}_{(v,rp)}\ v_i)$$

Case 1 ($\bigsqcup_{i \in I} v_i \not\sqsupseteq v$)**:** In this case we have $v_i \not\sqsupseteq v$ for all $i \in I$ and reason as follows.

$$\begin{aligned}
\overline{f}\ (\bigsqcup_{i \in I} v_i) &= f\ (\bigsqcup_{i \in I} v_i) && \text{definition of } \overline{f} \\
&= \bigsqcup_{i \in I} (f\ v_i) && \text{continuity of } f \\
&= \bigsqcup_{i \in I} (\overline{f}\ v_i) && \text{definition of } \overline{f}, v_i \not\sqsupseteq v
\end{aligned}$$

Case 2 ($\bigsqcup_{i \in I} v_i \sqsupseteq v$)**:** We consider a subset J of I such that $J = \{i \in I \mid v_i \sqsupseteq v\}$. By Lemma 4.3.9 v is finite. That is, there exists $i \in I$ such that $v_i \sqsupseteq v$ and, therefore, J is not empty. Because $\langle v_i \rangle_{i \in I}$ is a chain we only remove dominated elements, that is, $\bigsqcup_{i \in I} v_i = \bigsqcup_{i \in J} v_i$. Furthermore, by monotonicity of $\overline{f}$ we have $\bigsqcup_{i \in I} (\overline{f}\ v_i) = \bigsqcup_{i \in J} (\overline{f}\ v_i)$. We reason as follows.

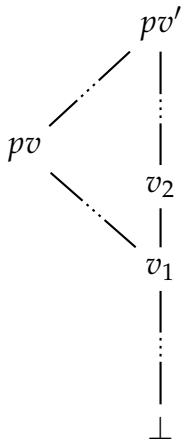
$$\begin{aligned}
\overline{f}\ (\bigsqcup_{i \in I} v_i) &= \overline{f}\ (\bigsqcup_{i \in J} v_i) && \text{supremum property} \\
&= (f\ (\bigsqcup_{i \in J} v_i))[(f\ (\bigsqcup_{i \in J} v_i) \sqcup \inf_f\ v)|_{rp}]_{rp} && \text{definition of } \overline{f} \\
&= (\bigsqcup_{i \in J} f\ v_i)[(\bigsqcup_{i \in J} f\ v_i) \sqcup \inf_f\ v)|_{rp}]_{rp} && \text{continuity of } f \\
&= (\bigsqcup_{i \in J} f\ v_i)[(\bigsqcup_{i \in J} (f\ v_i \sqcup \inf_f\ v))|_{rp}]_{rp} && \text{supremum property} \\
&= (\bigsqcup_{i \in J} f\ v_i)[\bigsqcup_{i \in J} (f\ v_i \sqcup \inf_f\ v)|_{rp}]_{rp} && \text{continuity of } \cdot|_{rp} \\
&= \bigsqcup_{i \in J} (f\ v_i)[(f\ v_i \sqcup \inf_f\ v|_{rp}]_{rp}) && \text{continuity of } \cdot[\cdot]_{rp}
\end{aligned}$$

$$= \bigsqcup_{i \in J} (\overline{f}\ v_i) \qquad \text{definition of } \overline{f}, v_i \sqsupseteq v$$
$$= \bigsqcup_{i \in I} (\overline{f}\ v_i) \qquad \text{supremum property}$$

This shows that $\overline{f}$ is continuous. □

Proof (of Lemma 4.3.12): We only have to consider the case that there exists $pv' \in [\![\tau]\!]$ with $pv' \sqsupseteq pv$ and $pv'|_p = \bot$ and $pv' \sqsupseteq v$. Otherwise p already satisfies the requirements we are looking for and we can set $p' := p$.

Let $(v, rp) \in C_\tau$ with $v \not\sqsubseteq pv$. Then there exists a sequence of sequential steps of the form $\bot \rightarrowtail^* v$. There exists a sub-sequence of the form $v_1 \rightarrowtail^* v_2$ with $v_1 \sqsubseteq pv$ and $v_2 \not\sqsubseteq pv$ because $\bot \sqsubseteq pv$ and $v \not\sqsubseteq pv$. We consider such a sequence of minimal length. This sequence has length one because there exists no value v' such that $v' \not\sqsubseteq pv$ and $v' \sqsubseteq pv$. Therefore, we have a step of the form $v_1 \rightarrowtail_{rp'} v_2$ with $v_1 \sqsubseteq pv$ and $v_2 \not\sqsubseteq pv$. The following Hasse-diagram illustrates the situation.



Let p' be the corresponding sequential position of $v_1 \rightarrowtail_{rp'} v_2$. We show that p' is a sequential position in pv at position rp. We have $v_1 \lessdot_{p'} v_2$, $v_2 \sqsubseteq pv'$, $v_1 \sqsubseteq pv$, $pv \sqsubseteq pv'$, and $v_2 \not\sqsubseteq pv$. By Lemma A.3.1 we have $pv|_{p'} = \bot$.

Because p' is a sequential position in v_1 at position rp' and we have $v_1 \sqsubseteq pv$ and $pv|_{p'} = \bot$ by definition of sequential positions p' is also a sequential position in pv at position rp'. By definition of the characteristic set C_f we have $rp \geq rp'$ which implies $rp = rp'$ because $(f\ pv)|_{rp} = \bot$ and $(f\ pv)|_{rp'} = \bot$.

That is, p' is a sequential position in pv at position rp. Because $v_2|_{p'} \neq \bot$ and $v \sqsupseteq v_2$ we have $v|_{p'} \neq \bot$. This implies $pv' \not\sqsupseteq v$ for all $pv' \in [\![\tau]\!]$ with $pv'|_{p'} = \bot$. □

Proof (of Lemma 4.3.13): Let pv be a partial value of type τ and $rp' \in Pos\ (\overline{f}\ pv)$ such that $(\overline{f}\ pv)|_{rp'} = \bot$. We have to show that there exists $p \in Pos\ pv$ such that $pv|_p = \bot$ and for all $pv' \in [\![\tau]\!]$ the following holds.

$$pv' \sqsupseteq pv \wedge pv'|_p = \bot \Longrightarrow (\overline{f}\ pv')|_{rp'} = \bot$$

We distinguish two cases.

Case 1 $(pv \sqsupseteq v)$**:** We have

$$\bot = (\overline{f}\ pv)|_{rp'} = ((f\ pv)[(f\ pv \sqcup \inf_f\ v)|_{rp}]_{rp})|_{rp'}$$

and distinguish further sub-cases.

Case a $(rp' \geq rp)$**:** We reason as follows.

$$\begin{aligned}
&((f\ pv)[(f\ pv \sqcup \inf_f\ v)|_{rp}]_{rp})|_{rp'} && \\
&\quad = (f\ pv \sqcup \inf_f\ v)|_{rp'} && rp' \geq rp \\
&\quad = (f\ v)|_{rp'} \sqcup (\inf_f\ v)|_{rp'} && \text{continuity of } \cdot|_{rp'}
\end{aligned}$$

This implies $(f\ pv)|_{rp'} = \bot$ and $(\inf_f\ v)|_{rp'} = \bot$. Because f is sequential, there exists a sequential position p such that $pv|_p = \bot$.

Let $pv' \in [\![\tau]\!]$ with $pv' \sqsupseteq pv$ and $pv'|_p = \bot$. The following equality proves that p is a sequential position in this case.

$$\begin{aligned}
(\overline{f}\ pv')|_{rp'} &= ((f\ pv')[(f\ pv' \sqcup \inf_f\ v)|_{rp}]_{rp})|_{rp'} && \text{definition of } \overline{f},\ pv' \sqsupseteq pv \\
&= (f\ pv' \sqcup \inf_f\ v)|_{rp'} && rp' \geq rp \\
&= (f\ pv')|_{rp'} \sqcup (\inf_f\ v)|_{rp'} && \text{continuity of } \cdot|_{rp'} \\
&= \bot \sqcup (\inf_f\ v)|_{rp'} && p \text{ sequential position} \\
&= \bot && (\inf_f\ v)|_{rp'} = \bot
\end{aligned}$$

Case b $(rp' \not\geq rp \wedge rp \not\geq rp')$**:** We have

$$((f\ pv)[(f\ pv \sqcup \inf_f\ v)|_{rp}]_{rp})|_{rp'} = (f\ pv)|_{rp'}.$$

Because f is a sequential function, there exists a sequential position p such that $pv|_p = \bot$.

Let $pv' \in [\![\tau]\!]$ with $pv' \sqsupseteq pv$ and $pv'|_p = \bot$. We reason as follows.

$$\begin{aligned}
(\overline{f}\ pv')|_{rp'} &= ((f\ pv')[(f\ pv' \sqcup \inf_f\ v)|_{rp}]_{rp})|_{rp'} && \text{definition of } \overline{f},\ pv' \sqsupseteq pv \\
&= (f\ pv')|_{rp'} && rp' \not\geq rp \wedge rp \not\geq rp' \\
&= \bot && p \text{ sequential position}
\end{aligned}$$

Case 2 $(pv \not\sqsupseteq v)$**:** We have

$$\bot = (\overline{f}\ pv)|_{rp} = (f\ pv)|_{rp}.$$

Because f is sequential, there exists a sequential position p in pv at position rp. By Lemma 4.3.12 there exists a sequential position p' in pv at position rp such that for all $pv' \in [\![\tau]\!]$ with $pv' \sqsupseteq pv$ and $pv'|_{p'} = \bot$ we have $pv' \not\sqsupseteq v$.

Let pv' such that $pv' \sqsupseteq pv$ and $pv'|_{p'} = \bot$. We show that p' is a sequential position as follows.

$$\begin{aligned} (\overline{f}\ pv')|_{rp} &= (f\ pv')|_{rp} && pv' \not\sqsupseteq v \\ &= \bot && p' \text{ sequential position} \end{aligned}$$

This completes the proof that $\overline{f}$ is a sequential function. □

B Proofs from Chapter 6

B.1 Proofs from Section 6.3

To prove Lemma 6.3.1 we define a function *strictConst* that behaves like the constant function *const* but which is strict in its second argument.

Definition B.1.1 (StrictConst): The function $\mathit{strictConst} :: \alpha \to \beta \to \alpha$ satisfies

$$\mathit{strictConst}\ x\ y = \begin{cases} \bot & y = \bot \\ x & \text{otherwise} \end{cases}.$$

We can define *strictConst* as $\mathit{strictConst}\ x\ y = \mathit{seq}\ y\ x$. If we are not equipped with *seq* we can as well define a function $\mathit{strictConst}_\tau :: \alpha \to \tau \to \alpha$ for every type τ by pattern matching. The functions *strictConst* and $\mathit{strictConst}_\tau$ are strict and total with respect to to their second argument. □

Proof (of Lemma 6.3.1): Let $xs :: [()]$. Let y be an element of type τ. We consider the list $ys = \mathit{map}\ (\mathit{strictConst}\ y)\ xs$. That is, we replace all occurrences of $()$ in xs by y and all occurrences of $\bot_{()}$ by $\bot_\tau$. We can reconstruct xs from ys if we replace all occurrences of y in ys by $()$ and all occurrences of $\bot_\tau$ by $\bot_{()}$, that is, $xs \equiv \mathit{map}\ (\mathit{strictConst}\ ())\ ys$. We reason as follows.

$$\begin{aligned}
& f_{()}\ xs \\
\equiv\ & \quad \{\ xs \equiv \mathit{map}\ (\mathit{strictConst}\ ())\ ys\ \} \\
& f_{()}\ (\mathit{map}\ (\mathit{strictConst}\ ())\ ys) \\
\equiv\ & \quad \{\ \text{free theorem for } f, \mathit{strictConst}\ ()\ \text{strict}\ \} \\
& \mathit{map}\ (\mathit{strictConst}\ ())\ (f_\tau\ ys) \\
\sqsupseteq\ & \quad \{\ \text{monotonicity and } f_\tau \preceq g_\tau\ \} \\
& \mathit{map}\ (\mathit{strictConst}\ ())\ (g_\tau\ ys) \\
\equiv\ & \quad \{\ \text{free theorem for } g, \mathit{strictConst}\ ()\ \text{strict}\ \} \\
& g_{()}\ (\mathit{map}\ (\mathit{strictConst}\ ())\ ys) \\
\equiv\ & \quad \{\ xs \equiv \mathit{map}\ (\mathit{strictConst}\ ())\ ys\ \} \\
& g_{()}\ xs
\end{aligned}$$

Thus, we have $f_{()}\ xs \sqsupseteq g_{()}\ xs$ for all $xs :: [()]$, that is, $f_{()} \preceq g_{()}$. □

Definition B.1.2 (Free Theorem Basics): In this definition we present some of the basic definitions by Johann and Voigtländer (2004).

strict A relation R is *strict* if $(\bot, \bot) \in R$.

bottom-reflecting A relation R is *bottom-reflecting* if for every $(x, y) \in R$ we have $x \neq \bot$ if and only if $y \neq \bot$.

continuous A relation R is continuous if for all chains $\langle x_i\rangle_{i\in I}$ and $\langle y_i\rangle_{i\in I}$ whose elements are pair-wise related by R we have $(\bigsqcup_{i\in I} x_i, \bigsqcup_{i\in I} y_i) \in R$. □

To prove Lemma 6.3.2 we prove the following generalization of the functional free theorem for a function of type $f :: (\alpha \to \beta) \to [\alpha] \to [\beta]$. The standard functional free theorem states that

$$h\ (p\ x) \equiv q\ (g\ x)$$

for all $x :: \tau_1$ implies

$$map\ h\ (f\ p\ xs) \equiv f\ q\ (map\ g\ xs)$$

for all $xs :: [\tau_1]$. In contrast, the following lemma shows that we do not have to show $h\ (p\ x) \equiv q\ (g\ x)$ for all $x :: \tau_1$ but only for the elements of xs.

Lemma B.1.1: *Let $f :: (\alpha \to \beta) \to [\alpha] \to [\beta]$. For strict functions $g :: \tau_1 \to \tau_2$, $h :: \tau_3 \to \tau_4$, $p :: \tau_1 \to \tau_3$, and $q :: \tau_2 \to \tau_4$ such that*

$$h\ (p\ (xs\ !!\ n)) \equiv q\ (g\ (xs\ !!\ n))$$

for all $n ::$ Int we have

$$map\ h\ (f\ p\ xs) \equiv f\ q\ (map\ g\ xs)$$

for all $xs :: [\tau_1]$.

Proof: Let $xs :: [\tau_1]$. We define the relations

$$R_{xs} := \{(\bigsqcup_{i\in I}(xs\ !!\ i), \bigsqcup_{i\in I} g\ (xs\ !!\ i)) \mid \langle xs\ !!\ i\rangle_{i\in I} \text{ chain}, I \subseteq Int\}$$

and $S := \{(x, h\ x)\}$. Theorem 6.4.1 shows that R_{xs} is strict and continuous. Furthermore, S is strict and continuous because h is strict and continuous.

The relational free theorem for $f :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ states that, if for all $(x, y) \in R_{xs}$ we have $(p\ x, q\ y) \in S$, then for all $(xs, ys) \in \text{lift}\{[\,]\}(R_{xs})$ we have $(f\ p\ xs, f\ q\ ys) \in \text{lift}\{[\,]\}(S)$. Let $(x, y) \in R_{xs}$. To show that $(p\ x, q\ y) \in S$ we have to show that $h\ (p\ x) \equiv q\ y$. By definition of R_{xs} we have $x = \bigsqcup_{i\in I}(xs\ !!\ i)$ and $y = \bigsqcup_{i\in I} g\ (xs\ !!\ i)$. We reason as follows.

$$\begin{aligned}
h\ (p\ (\bigsqcup xs\ !!\ i)) &\equiv \bigsqcup h\ (p\ (xs\ !!\ i)) && \text{continuity}\\
&\equiv \bigsqcup q\ (g\ (xs\ !!\ i)) && \text{precondition}\\
&\equiv q\ (\bigsqcup g\ (xs\ !!\ i)) && \text{continuity}
\end{aligned}$$

That is, $h\ (p\ x) \equiv q\ y$.

Thus, for all $(xs, ys) \in \text{lift}\{[\,]\}(R_{xs})$ we as well have $(f\ p\ xs, f\ q\ ys) \in \text{lift}\{[\,]\}(S)$. As we have $(xs, map\ g\ xs) \in \text{lift}\{[\,]\}(R_{xs})$ we get $(f\ p\ xs, f\ q\ (map\ g\ xs)) \in \text{lift}\{[\,]\}(S)$, that is, $map\ h\ (f\ p\ xs) \equiv f\ q\ (map\ g\ xs)$. □

The following lemma shows a simple property about $[n\,..]$ and $(!!)$. Here and in the following we consider $[n\,..]$ as shortform of *iterate* $(+1)\ n$.

Lemma B.1.2: *For all $n :: Int$ and all $m :: Int$ with $m \geq 0$ we have $[n\,..\,] \,!!\, m \equiv n + m$.*

Proof: By induction over m.
Base Case:

$$
\begin{array}{l}
\mathit{iterate}\ (+1)\ n \,!!\, 0 \\
\quad \equiv \quad \{ \text{ definition of } \mathit{iterate} \ \} \\
(n : \mathit{iterate}\ (+1)\ (n+1)) \,!!\, 0 \\
\quad \equiv \quad \{ \text{ definition of } (!!) \ \} \\
n
\end{array}
$$

Inductive Case:

$$
\begin{array}{l}
\mathit{iterate}\ (+1)\ n \,!!\, (m+1) \\
\quad \equiv \quad \{ \text{ definition of } \mathit{iterate} \ \} \\
(n : \mathit{iterate}\ (+1)\ (n+1)) \,!!\, (m+1) \\
\quad \equiv \quad \{ \text{ definition of } (!!) \ \} \\
\mathit{iterate}\ (+1)\ (n+1) \,!!\, m \\
\quad \equiv \quad \{ \text{ induction hypothesis } \} \\
n + 1 + m
\end{array}
$$

Proof (of Lemma 6.3.2): Let $x :: \tau$ and $xs :: [\tau]$. We reason as follows.

$$
\begin{array}{ll}
\mathit{map}\ ((x : xs)!!)\ [1\,..\,] & \\
\quad \equiv \quad \{ \text{ definition of } [1\,..\,] \ \} & \\
\mathit{map}\ ((x : xs)!!)\ (\mathit{iterate}\ (+1)\ 1) & \\
\quad \equiv \quad \{ \text{ free theorem for } \mathit{iterate}, (+1) \text{ strict \& total } \} & \\
\mathit{map}\ ((x : xs)!!)\ (\mathit{map}\ (+1)\ (\mathit{iterate}\ (+1)\ 0)) & \\
\quad \equiv \quad \{ \text{ Lemma B.1.1 and Lemma B.1.2 } \} & (*) \\
\mathit{map}\ (xs!!)\ (\mathit{iterate}\ (+1)\ 0) & \\
\quad \equiv \quad \{ \text{ definition of } [0\,..\,] \ \} & \\
\mathit{map}\ (xs!!)\ [0\,..\,] &
\end{array}
$$

To prove the step labeled with $(*)$ we have to prove the following equality.

$$(x : xs) \,!!\, ([1\,..\,] \,!!\, n) \equiv xs \,!!\, ([0\,..\,] \,!!\, n)$$

To prove this statement we distinguish two cases.

Case 1 $(n \equiv \bot \vee n < 0)$**:** We reason as follows.

$$
\begin{array}{l}
(x : xs) \,!!\, ([1\,..\,] \,!!\, n) \\
\quad \equiv \quad \{ \text{ definition of } (!!) \ \} \\
(x : xs) \,!!\, \bot \\
\quad \equiv \quad \{ \text{ definition of } (!!) \ \} \\
xs \,!!\, \bot \\
\quad \equiv \quad \{ \text{ definition of } (!!) \ \} \\
xs \,!!\, ([0\,..\,] \,!!\, n)
\end{array}
$$

Case 2 $(n \geq 0)$**:** We reason as follows.

$$
\begin{array}{ll}
& (x : xs) \mathbin{!!} ([1\,..\,] \mathbin{!!} n) \\
\equiv & \quad \{ \text{ Lemma B.1.2 } \} \\
& (x : xs) \mathbin{!!} n + 1 \\
\equiv & \quad \{ \text{ definition of } (!!) \} \\
& xs \mathbin{!!} n \\
\equiv & \quad \{ \text{ Lemma B.1.2 } \} \\
& xs \mathbin{!!} ([0\,..\,] \mathbin{!!} n)
\end{array}
$$

This step finally concludes the proof . □

Proof (of Lemma 6.3.6): Let $xs :: [\tau]$ and $n :: Int$ such that $f_\tau\ xs \mathbin{!!} n \not\equiv \bot$, $g_\tau\ xs \mathbin{!!} n \not\equiv \bot$, and $f_\tau\ xs \mathbin{!!} n \not\equiv g_\tau\ xs \mathbin{!!} n$. By Lemma 6.3.5 there exist $k, l :: Int$ such that the following holds.

$$xs \mathbin{!!} k \equiv f_\tau\ xs \mathbin{!!} n \not\equiv g_\tau\ xs \mathbin{!!} n \equiv xs \mathbin{!!} l$$

This implies $k \not\equiv l$. Furthermore we have $k \not\equiv \bot$ and $l \not\equiv \bot$ because $f_\tau\ xs \mathbin{!!} n \not\equiv \bot$ and $g_\tau\ xs \mathbin{!!} n \not\equiv \bot$. This also implies $xs \mathbin{!!} k \not\equiv \bot$ and $xs \mathbin{!!} l \not\equiv \bot$.

If we replace the element of xs at position k by $\bot$ we get a list $ys :: [\tau]$ with $ys \mathbin{!!} k \equiv \bot$ and $ys \mathbin{!!} l \not\equiv \bot$. We have $shape\ xs \equiv shape\ ys$ and by Lemma 6.3.5 we get $f_\tau\ ys \mathbin{!!} n \equiv ys \mathbin{!!} k \equiv \bot$ and $g_\tau\ ys \mathbin{!!} n \equiv ys \mathbin{!!} l \equiv xs \mathbin{!!} l \not\equiv \bot$. □

Proof (of Lemma 6.3.7): Because we have $f_\tau \not\preceq g_\tau$ there exists $xs :: [\tau]$ such that $f_\tau\ xs \not\sqsupseteq g_\tau\ xs$. We distinguish three cases.

Case 1 $(shape\ (f_\tau\ xs) \not\sqsupseteq shape\ (g_\tau\ xs))$**:** We set $ys \equiv map\ (strictConst\ ())\ xs$ and reason as follows.

$$
\begin{array}{ll}
& shape\ (f_{()}\ ys) \\
\equiv & \quad \{ ys \equiv map\ (strictConst\ ())\ xs \} \\
& shape\ (f_{()}\ (map\ (strictConst\ ())\ xs)) \\
\equiv & \quad \{ \text{ free theorem for } f, strictConst\ () \text{ strict } \} \\
& shape\ (map\ (strictConst\ ())\ (f_\tau\ xs)) \\
\equiv & \quad \{ \text{ free theorem for } shape, strictConst\ () \text{ strict } \} \\
& shape\ (f_\tau\ xs) \\
& \not\sqsupseteq \\
& shape\ (g_\tau\ xs) \\
\equiv & \quad \{ \text{ free theorem for } shape, strictConst\ () \text{ strict } \} \\
& shape\ (map\ (strictConst\ ())\ (g_\tau\ xs)) \\
\equiv & \quad \{ \text{ free theorem for } g, strictConst\ () \text{ strict } \} \\
& shape\ (g_{()}\ (map\ (strictConst\ ())\ xs)) \\
\equiv & \quad \{ ys \equiv map\ (strictConst\ ())\ xs \} \\
& shape\ (g_{()}\ ys)
\end{array}
$$

Therefore, there exists $ys :: [()]$ such that $shape\ (f_{()}\ ys) \not\sqsupseteq shape\ (g_{()}\ ys)$ which implies $f_{()} \not\preceq g_{()}$.

Case 2 $(\exists n :: Int.\, f_\tau\ xs\ !!\ n \equiv \bot \wedge g_\tau\ xs\ !!\ n \not\equiv \bot)$**:** We set $ys \equiv map\ (strictConst\ ())\ xs$ and reason as follows.

$$
\begin{array}{ll}
& f_{()}\ ys\ !!\ n \\
\equiv & \quad \{\ ys \equiv map\ (strictConst\ ())\ xs\ \} \\
& f_{()}\ (map\ (strictConst\ ())\ xs)\ !!\ n \\
\equiv & \quad \{\ \text{free theorem for } f, strictConst\ ()\ \text{strict}\ \} \\
& map\ (strictConst\ ())\ (f_\tau\ xs)\ !!\ n \\
\equiv & \quad \{\ \text{free theorem for } (!!), strictConst\ ()\ \text{strict}\ \} \\
& strictConst\ ()\ (f_\tau\ xs\ !!\ n) \\
\equiv & \quad \{\ \text{definition of } strictConst, f_\tau\ xs\ !!\ n \equiv \bot\ \} \\
& \bot
\end{array}
$$

For g we reason the same way.

$$
\begin{array}{ll}
& g_{()}\ ys\ !!\ n \\
\equiv & \quad \{\ ys \equiv map\ (strictConst\ ())\ xs\ \} \\
& g_{()}\ (map\ (strictConst\ ())\ xs)\ !!\ n \\
\equiv & \quad \{\ \text{free theorem for } g, strictConst\ ()\ \text{strict}\ \} \\
& map\ (strictConst\ ())\ (g_\tau\ xs)\ !!\ n \\
\equiv & \quad \{\ \text{free theorem for } (!!), strictConst\ ()\ \text{strict}\ \} \\
& strictConst\ ()\ (g_\tau\ xs\ !!\ n) \\
\not\equiv & \quad \{\ \text{definition of } strictConst, g_\tau\ xs\ !!\ n \not\equiv \bot\ \} \\
& \bot
\end{array}
$$

That is, there exists $ys :: [()]$ and $n :: Int$ such that $f_{()}\ ys\ !!\ n \equiv \bot$ and $g_{()}\ ys\ !!\ n \not\equiv \bot$ which implies $f_{()} \not\preceq g_{()}$.

Case 3 $(\exists n :: Int.\, f_\tau\ xs\ !!\ n \not\equiv \bot \wedge g_\tau\ xs\ !!\ n \not\equiv \bot \wedge f_\tau\ xs\ !!\ n \not\equiv g_\tau\ xs\ !!\ n)$**:** According to Lemma 6.3.6 there exists $ys :: [\tau]$ and $n :: Int$ such that $f_\tau\ ys\ !!\ n \equiv \bot$ and $g_\tau\ ys\ !!\ n \not\equiv \bot$. That is, we reason as in Case 2. □

Bibliography

Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 62–73, New York, NY, USA, 2005. ACM.

Sergio Antoy. Definitional trees. In Hélène Kirchner and Giorgio Levi, editors, *Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer Berlin / Heidelberg, 1992.

Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 268–279, New York, NY, USA, 1994. ACM.

Lennart Augustsson. numbers package. `http://hackage.haskell.org/package/numbers`, August 2009. version 2009.8.9.

Jaco W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1980.

Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In Andrew Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144. Springer Berlin / Heidelberg, 2010.

Richard Bird, Geraint Jones, and Oege de Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7:541–547, September 1997.

Sascha Böhme. Free theorems for sublanguages of haskell, 2007. Master's thesis, Technische Universität Dresden.

Bernd Braßel and Frank Huch. The Kiel Curry System KiCS. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *Applications of Declarative Programming and Knowledge Management*, volume 5437 of *Lecture Notes in Computer Science*, pages 195–205. Springer Berlin / Heidelberg, 2009.

Bernd Braßel, Sebastian Fischer, and Frank Huch. Declaring numbers. *Electronic Notes in Theoretical Computer Science*, 216:111–124, July 2008.

Alan Bundy and Julian Richardson. Proofs about lists using ellipsis. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning*, volume 1705 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 1999.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM.

Olaf Chitil. Pretty printing with lazy dequeues. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27:163–184, January 2005.

Olaf Chitil. Promoting non-strict programming. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages*, IFL '06, 2006.

Olaf Chitil. StrictCheck: a tool for testing whether a function is unnecessarily strict. Technical Report 2-11, University of Kent, School of Computing, June 2011.

Jan Christiansen. Sloth – a tool for checking minimal-strictness. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL '11, pages 160–174. Springer Berlin / Heidelberg, 2011.

Jan Christiansen and Daniel Seidel. Minimally strict polymorphic functions. In *Proceedings of the 2011 Symposium on Principles and Practice of Declarative Programming*, PPDP '11, pages 53–64, New York, NY, USA, 2011. ACM.

Jan Christiansen, Daniel Seidel, and Janis Voigtländer. Free theorems for functional logic programs. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 39–48, New York, NY, USA, 2010. ACM.

Jan Christiansen, Daniel Seidel, and Janis Voigtländer. An adequate, denotational, functional-style semantics for typed flatcurry. In Julio Mariño, editor, *Functional and Constraint Logic Programming*, WFLP '11, pages 119–136. Springer Berlin / Heidelberg, 2011a.

Jan Christiansen, Daniel Seidel, and Janis Voigtländer. An adequate, denotational, functional-style semantics for typed FlatCurry without letrec. Technical Report IAI-TR-2011-1, University of Bonn, March 2011b.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

Nils A. Danielsson and Patrik Jansson. Chasing Bottoms, a case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *MPC '04*, pages 85–109. Springer Berlin / Heidelberg, July 2004.

Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

Bertram Felgenhauer. Wadler space leak. Glasgow Haskell Users Mailing List, October 2010.

João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 95–106, New York, NY, USA, 2007. ACM.

Daniel Fischer. Unnecessarily strict implementations. Haskell-Cafe Mailing List, September 2010.

Daniel Fischer, Chris Kuklewicz, and Justin Bailey. stringsearch Package. `http://hackage.haskell.org/package/stringsearch`, 2010. Version 0.3.1.

Sebastian Fischer and Herbert Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 63–74, New York, NY, USA, 2007. ACM.

GHC. The Glasgow Haskell Compiler. `http://haskell.org/ghc`.

Jeremy Gibbons. A pointless derivation of radix sort. *Journal of Functional Programming*, 9:339–346, May 1999.

Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19:227–251, March 2009.

Andy Gill and Colin Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 1–12, New York, NY, USA, 2007. ACM.

Jean-Yves Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. *Electronic Notes in Theoretical Computer Science*, 26:69 – 86, 1999.

Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 265–276, New York, NY, USA, 2001. ACM.

Michael Hanus. Curry: Example programs. `http://www.informatik.uni-kiel.de/~curry/examples`.

Michael Hanus. Curry: An integrated functional logic language (version 0.8.2). `http://curry-language.org`, 2006.

Furio Honsell and Donald Sannella. Pre-logical relations. In *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 826–826. Springer Berlin / Heidelberg, 1999.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

John Hughes. Why functional programming matters. *The Computer Journal*, 32: 98–107, April 1989.

Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9:355–372, July 1999.

Patricia Johann. A generalization of short-cut fusion and its correctness proof. *Higher Order Symbolic Computation*, 15:273–300, December 2002.

Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 99–110, New York, NY, USA, 2004. ACM.

Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 52–61, New York, NY, USA, 1993. ACM.

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*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.

Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, March 1966.

John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

Simon Marlow. Inlining defeats selector thunk optimisation. Glasgow Haskell Compiler Trac, September 2008. Ticket #2607.

John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '61 (Western), pages 225–238, New York, NY, USA, 1961. ACM.

John Mitchell and Albert Meyer. Second-order logical relations. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 225–236. Springer Berlin / Heidelberg, 1985.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 54–67, New York, NY, USA, 1996. ACM.

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. Effectiveadvice: disciplined advice with explicit effects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 109–120, New York, NY, USA, 2010. ACM.

Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 25–36, New York, NY, USA, 1999. ACM.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, January 2007.

Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

Nicholas Pippenger. Pure versus impure Lisp. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19:223–238, March 1997.

Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.

Rawle Prince, Neil Ghani, and Conor McBride. Proving properties about lists using containers. In *Proceedings of the 9th international conference on Functional and logic programming*, FLOPS '08, pages 97–112. Springer Berlin / Heidelberg, 2008.

Project Gutenberg. `http://www.gutenberg.org`, 2011.

Colin Runciman and David Wakeling. Heap profiling of a lazy functional compiler. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 203–214, London, UK, 1993a. Springer Berlin / Heidelberg.

Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(02):217–245, 1993b.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 37–48, New York, NY, USA, 2008. ACM.

George Russell. List.partition a bit too eager. Haskell-Cafe Mailing List, December 2000.

Patrick M. Sansom and Simon Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 355–366, New York, NY, USA, 1995. ACM.

David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. McGraw-Hill Professional, 1987.

Dana S. Scott, Carl A. Gunter, and Peter D. Mosses. Semantic domains and denotational semantics. Technical Report DAIMI PB-276, Computer Science Department, Aarhus University, Denmark, 1989.

Daniel Seidel and Janis Voigtländer. Taming selective strictness. In *Arbeitstagung Programmiersprachen, Lübeck, Germany, Proceedings*, volume 154 of *Lecture Notes in Informatics*, pages 2916–2930. GI, October 2009.

Daniel Seidel and Janis Voigtländer. Automatic generation of free theorems. `http://www-ps.iai.uni-bonn.de/ft`, 2009.

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, Imperial College London, February 2011.

Jan Sparud. Fixing some space leaks without a garbage collector. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 117–122, New York, NY, USA, 1993. ACM.

Henning Thielemann. utility-ht Package. `http://hackage.haskell.org/package/utility-ht`, 2009. Version 0.0.5.1.

David A. Turner. The SASL language manual. Technical report, University of St Andrews, 1976.

Twan van Laarhoven. Add 'subsequences' and 'permutations' to data.list (ticket #1990). Libraries Mailing List, December 2007.

Janis Voigtländer. Proving correctness via free theorems: the case of the destroy/build-rule. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 13–20, New York, NY, USA, 2008a. ACM.

Janis Voigtländer. Much ado about two (pearl): a pearl on parallel prefix computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 29–35, New York, NY, USA, 2008b. ACM.

Janis Voigtländer. Bidirectionalization for free! In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 165–176, New York, NY, USA, 2009. ACM.

Jean Etienne Vuillemin. *Proof-techniques for recursive programs.* PhD thesis, Stanford University, 1974.

Philip Wadler. Fixing some space leaks with a garbage collector. *Software: Practice and Experience*, 17(9):595–608, 1987.

Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

Philip Wadler. *The Fun of Programming*, chapter 11, pages 223–244. Cornerstones in Computing. Palgrave, 2003.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

Christopher P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. Ph.D. thesis, Programming Research Group, Oxford University, September 1971.

Johannes Waldmann. List.partition is too strict. Haskell Mailing List, September 2000.

Brent Yorgey. split Package. `http://hackage.haskell.org/package/split`, December 2010. version 0.1.2.

Brent Yorgey. split Package. `http://hackage.haskell.org/package/split`, April 2011. Version 0.1.4.