Tools for Reasoning about Effectful Declarative Programs

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

vorgelegt von
Stefan Georg Mehner
aus
Iserlohn

Bonn 2015

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

Erstgutachter: Jun.-Prof. Dr. Janis Voigtländer
Zweitgutachter: Prof. Dr. Michael Hanus
Tag der Promotion: 16. Oktober 2015
Erscheinungsjahr: 2015

Zusammenfassung

In der rein funktionalen Programmiersprache Haskell müssen fast alle Seiteneffekte, die eine Funktion erzeugen kann, in ihrem Typ vermerkt werden. Darunter fallen Interaktion mit der Umwelt, Propagation eines Zustandes und auch Nichtdeterminismus. Sind keine Seiteneffekte vermerkt, verhält sich so eine Funktion wie eine Funktion im Sinne der Mathematik, also als eindeutige Zuordnung. In diesem Fall kann man über Ausdrücke in einem Programm Beweise führen wie über mathematische Ausdrücke. Neben diesem sogenannten gleichungsbasierten Schließen ermöglicht das Typsystem auch typbasiertes Schließen. Ein Beispiel dafür sind freie Theoreme – Gleichungen zwischen Ausdrücken, die allein aufgrund der Typen der beteiligten Funktionen gelten. Solche Aussagen lassen sich zum Teil nutzen, um Optimierungsstrategien in Compilern formal zu rechtfertigen.
Die vorliegende Arbeit untersucht zwei Verallgemeinerungen solcher Methoden für Programme, die nicht frei von Seiteneffekten, also effektbehaftet, sind. Erstens werden effektbehaftete Traversierungen von Datenstrukturen untersucht. Der wichtigste Beitrag hier ist, dass eine Datenstruktur genau dann regelkonform traversiert werden kann, wenn sie isomorph zu einem polynomiellen Funktor ist. Dieses Ergebnis verbindet das verbreitete Interface des Traversierens mit einer klaren Vorstellung über den Aufbau und die Verhaltensweise der Datenstruktur. Weiterhin werden Werkzeuge vorgestellt, um über effektbehaftete Traversierungen bequem Beweise führen zu können.
Zweitens werden freie Theoreme für die funktional-logische Sprache Curry hergeleitet. Aufgrund der nahen Verwandtschaft der beiden Sprachen, lässt sich Curry als Haskell mit integriertem Nichtdeterminismus, also einem integrierten Seiteneffekt, verstehen. Sowohl gleichungsbasiertes als auch typbasiertes Schließen lassen sich bis zu einem gewissen Grad auf Curry übertragen. Insbesondere wird Short-Cut-Fusion – eine sehr ergiebige Laufzeitoptimierung – für Curry ermöglicht.

Abstract

In the pure functional language Haskell, nearly all side-effects that a function can produce have to be noted in its type. This includes input/output, propagation of a state, and nondeterminism. If no side-effects are noted, such a function acts like a mathematical function, i.e., mapping arguments to unique results. In that case, expressions in a program can be reasoned about like mathematical expressions. In addition to this so-called equational reasoning, the type system also enables type based reasoning. One example are free theorems – equations between expressions that are true only due to the types of the expressions involved. Some such statements serve as formal justification for optimization strategies in compilers.

The thesis at hand investigates two generalizations of such methods for programs not free of side-effects, i.e., effectful programs. First, effectful traversals of data structures are being studied. The most important contribution in this part is that a data structure can be lawfully traversed if, and only if, it is isomorphic to a polynomial functor. This result links the widespread interface of traversing to a clear intuition regarding the structure and behavior of the data type. Furthermore, tools are presented facilitating convenient proofs about effectful traversals.

Second, free theorems for the functional-logic language Curry are derived. Due to the close relationship between both languages, Curry can be understood as Haskell with built-in nondeterminism, i.e., a built-in side-effect. Equational and type based reasoning can both be adapted to Curry to a certain degree. In particular, short cut fusion – a very fertile runtime optimization – is enabled for Curry.

Acknowledgments

First and foremost, I want to thank my advisor, Janis Voigtländer. He accepted me as his PhD student, without me ever having been in any of his lectures, and hired me, without ever having seen my résumé. He always trusted my instincts and let me elaborate my ideas, yet also introduced me to the scientific community and its workings. He entrusted me with responsibilities in teaching, but always made sure I had enough time left for doing research and working on this thesis. Despite of the tight budget, he always managed to extend my contract somehow and gave me (just) enough time to finish my PhD.

I had the pleasure to also co-author with other experienced colleagues, whose input and encouragement I appreciated a lot. I consider it an honor they have entrusted me with giving the talk at both conferences we published at.

The other members of our group, Daniel Seidel, Helmut Grohne and Tobias Gödderz, have always been open for discussions and offered advice. Yet, they also contributed in so many other ways while we shared an office, just by being great workmates. I am especially indebted to Helmut, who gave a hand in grading sheets so I had more time to finish this thesis. On top of that, he proof-read all of it and had many helpful suggestions regarding content, formulation, layout and typesetting.

I want to thank Michael Hanus, Stefan Kratsch, and Carsten Urbach for their work on the doctoral committee and, in particular, for sticking to the tight schedule.

Furthermore, I want to thank all the computer vision and cognitive computer vision people here in Bonn. We had countless lunches and occasional barbeques with them and in the end they even prepared my graduate cap.

Another person I owe thanks to is my dear friend Irene. She proof-read the whole thesis and without her my English would be far worse. Apart from that, she helped me to keep my morale up with our weekly walks.

I want to thank my brother for his suggestion I should switch to computer science. It surprised me at first, but turned out to be brilliant.

And finally, I have to thank my beloved fiancée Bettina. She made it possible for me to finish my PhD here in Bonn and live together with her at the same time. She decided almost six years of commuting had been enough and moved in with me, even though Bonn has never been her favorite place (and it has not become her favorite place in the meantime). I am deeply thankful for her decision and this document shall serve as a reminder that I owe her a favor when it comes to agreeing on where we live.

Contents

Chapter 1

Introduction

Declarative programming is best defined by differentiating it from imperative programming. When we use the term *program* in a non-technical way, it refers to a succession of steps that are being taken one after the other. This description also applies well to imperative programs, which are a succession of commands some machine is supposed to execute. Taking this into account, a *declarative program* almost is a contradictory notion because it is *not* a succession of steps. It can rather be seen as a mathematical definition that can be operationalized, but has a meaning independent of any machine. In particular, operationalizing the program is not the programmer's duty, but done by the compiler in a way *it* sees apt.

Two important sub-paradigms of the declarative are functional programming and logic programming. Functional programming is based on the notion of function in its mathematical meaning, i.e., one value being uniquely determined by another via some rule. Computation proceeds by repeatedly plugging in function definitions, until a result is reached. The programmer only has to provide the definitions, while the mechanism for evaluation is provided by the underlying machinery.

Logic programming is based on logic inference rules. The computer is to find an object satisfying some given requirements, much like a solution to a logic puzzle. The programmer only defines what constitutes a solution, while the task of actually finding one or all of them is left to the computer.

One argument often made in favor of declarative programming is that it lends itself to formal reasoning. We do not have to content ourselves with describing what the program makes the machine do. Instead, we have an understanding of what the machine is supposed to return after executing a program. Thus, the algorithmic intention takes the center stage.

Yet, it is not enough that a program has a meaning in an abstract sense. The machine has to act in some way, otherwise we never get to see the result.

Take the logic programming language Prolog [Colmerauer and Roussel, 1996] as an example, which probably even more than functional programming has the appeal of performing magic. At least for beginners, the interpreter seems to generate solutions

like a magician generates rabbits in his hat: We know they do not come out of nothing, but the mechanism is hidden so well, we are willing to go along with the make-believe. Yet, the illusion does not scale well. As soon as it comes to writing real programs interacting with the user, the programmer has to know the order in which program parts are evaluated. If a bad user input causes some program branch to fail, does backtracking allow the user to change his input or does it try some other branch? Since one now has to think about the mechanism, programming is again about telling the machine what to do. In doing so, part of the illusion is destroyed.
The problem is that input and output are side-effects, which do not fit into the mathematical theory the language is based upon. Suddenly, we can observe *from the outside* how often certain parts of code are run. So when doing the transition from Prolog as a language for describing constraints to Prolog as a general purpose language, we sacrifice some of our ability of formal reasoning.

The pure functional language Haskell [Hudak et al., 2007] takes a different approach to side-effects: Whenever some function potentially triggers a side-effect, this has to be recorded in the function's type. Such rigorous discipline is necessary because Haskell is evaluated lazily. Expressions can be stored in unevaluated form and will remain like this until the result is actually needed or the garbage collector disposes of the expression. Lazy evaluation has initially been implemented to save runtime. Sticking to this design choice has forced the language to rigorously distinguish between pure and effectful code. Otherwise, an unevaluated *getChar* could be stored and then be evaluated at some later point, leading to a confusing external behavior.
Haskell has found clever ways to deal with side-effects. In effectful parts of the program – **do**-blocks – the programmer can pretend to be using an imperative language for a 'virtual machine'. Here, statements have an order in which they need to be executed. The range of functions of this 'virtual machine' can be adjusted as needed to make sure only certain kinds of side-effects occur. If the 'virtual machine' has an underlying pure implementation, it can be run inside pure parts of the program that 'catch' the effects. Finally, communication with the outside world is governed by the same 'virtual machine' interface, only in this case it is wired to the actual machine.

The problem with effectful program parts is that they are less accessible to formal reasoning. Many of the approved tools only work for the pure heart of the language and are not directly applicable to effectful parts. Due to the type system we at least know which parts are afflicted, but the situation is still unsatisfactory.
One solution would be to instead argue about the pure underlying implementation, if there is one. This is not always the case and even if it is, we do not want to descend to this level. After all, there is a reason why we wanted to hide the pure implementation behind an interface while programming. So when we start to reason about the program, why would we want to remove the inserted ceiling again?
Moreover, as soon as we do not know the details of the implementation anymore, this approach is vain anyhow. Be it because we are arguing about a function that uses

input/output-primitives or be it because we are arguing about a function that is abstracted over the kind of effect it possesses – sometimes we just cannot take a look inside the black box.

This leads to the first part (chapter 3) of this thesis: a study of effectful traversals [McBride and Paterson, 2008]. Traversals are a special kind of effectful program, in which an effectful function is applied to every entry of some data structure, collecting the results in a structure of equal shape and combining the occurring effects. Despite this programming pattern being widely-used and well established, the exact set of rules that describe the expected behavior has been up for discussion until recently.
Our paper [Bird et al., 2013] shed light on the exact implications of the laws commonly required for traversals: Every lawfully traversable structure is a finitary container [Abbott et al., 2003], i.e., traversable objects can be split into shape and contents. The result also provides tools to formally reason about effectful traversals in a convenient and straightforward manner.
I have decided to rewrite the whole story here for various reasons. The underlying paper is the work of five authors, so it also contains material that is not mine and a clarification on who did what is due. This alone could have been accomplished by some paragraphs of background about the development leading to the publication. Yet, rewriting the story allows me to considerably shift the focus. There is also additional material that did not make it into the paper because of the lack of space or because it has only been developed later. The format of a thesis allows for matters to be discussed more thoroughly.

A language closely related to Haskell is the functional-logic language Curry [Hanus, 2013]. Like Haskell, it is evaluated lazily, but allowing nondeterminism as a built-in feature. From the logic side Curry borrows logic variables and constraint solving.
Haskell also allows to write nondeterministic programs by using **do**-blocks and a 'virtual machine' that does backtracking. Yet, the sequential order of statements forces the machine to branch whenever a nondeterministic computation is encountered. Even if the result of this computation is never used, the rest of the program still has to be run again and again.
Having nondeterminism as a built-in feature, Curry evaluates nondeterministic expressions lazily. Thus, the program only branches if the result of some computation actually influences how the evaluation proceeds. A big part of the search space can often be ruled out as a whole when some constraint is violated uniformly across it. In this respect, Curry behaves differently than effectful Haskell with explicit nondeterminism. Also, lazy nondeterministic evaluation can be used to implement logic variables and constraint solving [Antoy and Hanus, 2006].

When it comes to reasoning formally about Curry, surprisingly little is known. The close relationship to Haskell suggests that many of the known techniques might still work, at least in modified form. Obviously, nondeterminism has to be taken into account somehow and what we know about the pure part of Haskell does not simply carry over to a nondeterministic setting. Also, Curry is not the same as Haskell with nondeterminism as an effect, as we have discussed above. So even if we knew how to argue about effectful Haskell sufficiently well, these results would not simply carry over by viewing Curry as nondeterministic Haskell. Instead, new techniques are necessary.

This motivates the second half (chapter 4) of this thesis: an investigation on the semantics of Curry. The aiming point are free theorems [Wadler, 1989] for Curry because they turned out to be so useful for Haskell. On the way a number of tools will be presented, that hopefully prove useful in other contexts as well: A functional-style denotational semantics for (a simplified version of) Curry is given and proved to be equivalent to the standard operational semantics. Also, a translation procedure into SaLT (a lambda calculus with sets), that makes Curry's inherent nondeterminism visible, is defined.

Aspects of this have already been published before: A precursor of the denotational semantics already appears in [Christiansen et al., 2011a]. A revised version of the denotational semantics, the translation into SaLT, a parametricity theorem [Reynolds, 1983] for SaLT, and how to formulate and prove some easy free theorems using these tools can be found in [Mehner et al., 2014]. Establishing a connection between a functional-style denotational and the operational semantics for Curry has already been attempted [Christiansen et al., 2011a], but never carried out. Here, the first proof is given of two such semantics for a functional-logic language to be related.

This requires a lot of fine tuning in the denotational semantics, which led me to change many of the details as compared to both [Christiansen et al., 2011a] and [Mehner et al., 2014]. The parametricity theorem from [Mehner et al., 2014] is adapted to the latest version of the denotational semantics, but also generalized to an inequational setting as in [Johann and Voigtländer, 2006]. The language SaLT and the translation procedure have been altered slightly to make working with translated code easier. Finally, short cut fusion appears as a first application of a free theorem in Curry.

The remainder of this thesis is organized as follows: Some relevant foundations are reviewed in chapter 2. Both of the two main chapters 3 and 4 start with a discussion of the story behind their respective developments and a detailed description of their respective internal structures. Afterwards, chapter 5 offers some closing remarks and perspectives.

Chapter 2

Foundations

Contents

We assume the reader to be superficially familiar with the language Haskell [Hudak et al., 2007] already. Yet, there are some concepts worth discussing before proceeding to the main contents of this thesis. There is much more to be said concerning every one of these foundational topics, but since there is a lot of other material available we confine ourselves to the bare necessities.

Section 2.1 gives some superficial understanding of what free theorems [Wadler, 1989] are, why they hold and what they can be used for. They will show up occasionally in chapter 3, but are not in any way central there. They do play an important role in chapter 4, for which they are the main motivation. Despite their importance for the development, they only enter the picture towards the end.

Section 2.2 revolves around Haskell's approach of monadic effect handling [Wadler, 1992]. Also, it introduces a running example for chapter 3 taken from [Hutton and Fulger, 2008]. The section is also relevant regarding Curry in different respects. For one thing, Curry does input/output much the same way Haskell does. Also, the language SaLT we will be using in some parts of chapter 4 handles nondeterminism in a monadic fashion.

The language Curry is introduced – again assuming some familiarity with Haskell – in section 2.3.

LaTeX	$\circ$	$\cdot$	$+\!\!+$	$==$	$\circledast$	$\wedge$	$\vee$	$\gg\!=$	λ	$\forall.$	$\leftarrow$	$\rightarrow$	$\Rightarrow$
ASCII	`.`	`*`	`++`	`==`	`<*>`	`&&`	`||`	`>>=`	`\`	`forall .`	`<-`	`->`	`=>`

Figure 2.1: ASCII representations of symbols used by lhs2TeX

Before we start, some conventions need to be pointed out that are used throughout this thesis. This document is generated using lhs2TeX[1] for setting code. Among other features, it allows to replace the actual ASCII representation of operators in code by more readable symbols. The correspondence is documented in figure 2.1 and allows to translate the symbols back into proper code.

Haskell uses two different equality symbols, $=$ and $==$. The former is used in definitions of functions and variables and the latter is the Boolean-valued equality test. In this thesis, the single equality symbol is also used for semantic equivalence.

We distinguish between type variables appearing in type signatures of polymorphic functions and arbitrary fixed types. The former have their usual meaning, abstract over types in the language itself, and we denote them using lower case letters. The latter are variables on the meta-level, i.e., when applying the results presented here, all such variables have to be replaced by proper types. Arbitrary fixed types are denoted using upper case letters.

Occasionally, the $\forall a.\tau$ syntax is used to stress that a type is polymorphic. In Haskell this is syntactically correct, though it requires the `ExplicitForall` or `RankNTypes` extension. The same syntax will occasionally be used for Curry, even though it is not actually part of the language.

2.1 Parametric Polymorphism and Free Theorems

In this section we provide a basic understanding of *free theorems* [Wadler, 1989] and what they can be used for. We do not go into details on how to prove free theorems, though. A more elaborate discussion and formal background can be found in [Wadler, 1989].

The short version is this: Free theorems are statements connecting different instantiations of a parametrically polymorphic function and are derived from the function's type alone. The polymorphism being *parametric* is the essential requirement here because we need the different instantiations to rely on the same code.

As an example, consider the function *filter*, which takes a predicate (i.e., a Boolean-valued function) and a list and returns a list containing only those entries satisfying the predicate:

$$
\begin{aligned}
&\mathit{filter} :: \forall a.(a \to \mathsf{Bool}) \to [\,a\,] \to [\,a\,]\\
&\mathit{filter}\ p\ [\,] \qquad\quad = [\,]\\
&\mathit{filter}\ p\ (x : \mathit{xs}) = \mathbf{if}\ p\ x\ \mathbf{then}\ x : \mathit{filter}\ p\ \mathit{xs}\ \mathbf{else}\ \mathit{filter}\ p\ \mathit{xs}
\end{aligned}
$$

[1] `http://www.andres-loeh.de/lhs2tex`

The function *filter* is polymorphic, i.e., it can be used for different types (represented by the type variable a) and always uses the above code.
For example, *filter even* $[6,3,4,5,7,8,9]$ equals $[6,4,8]$ and *filter* ($\leqslant$ `'k'`) `"haskell"` equals `"hake"`.
The free theorem for *filter* (or rather for the type of *filter*) states that the equation

$$map\ g\ (filter\ (p \circ g)\ xs) = filter\ p\ (map\ g\ xs) \tag{2.1}$$

holds for all types A and B, functions $g :: \mathsf{A} \to \mathsf{B}$ and $p :: \mathsf{B} \to \mathsf{Bool}$ and lists $xs :: [\mathsf{A}]$. Note that on the left hand side *filter* is instantiated at A, while on the right hand side it is instantiated at B.
Equation (2.1) can be proved using equational reasoning and without relying on free theorems. This is done by induction and distinguishing three cases regarding xs: The list can be empty, it can be $x : xs$ where $p\ (g\ x)$ holds or it can be $x : xs$ where $p\ (g\ x)$ does not hold. All three cases are proved by using the definitions of *filter* and *map*.

The real power of free theorems is that equation (2.1) can be derived from the type of *filter* alone. Thus the same equation is true for *any* function of that type, i.e., $f :: \forall a.(a \to \mathsf{Bool}) \to [a] \to [a]$. The function f could be a slight variation of *filter* keeping all entries for which the predicate does not hold. It could also check whether all list entries satisfy the predicate and if so, reverse the list's order. Of course the function could also do something silly like always return the second argument or always return an empty list. We just do not know, but we still always know the equation to hold.
There are things the function f can certainly not do. For example f cannot generate any new list entries that are not already present in the argument list. This is due to f having a rather restricted interface to the entry type. The only way to get something of the unknown entry type is by taking it from the list given as the second argument. The predicate can be applied to some of the entries and further decisions may be based on the outcome of such tests. Apart from that, entries can only be passed on in the result list, possibly duplicating, reordering or dropping some of them.
Consider the two instantiations of f at A resp. B and the function arguments $p \circ g$ and xs, resp. p and $map\ g\ xs$. The two lists xs and $map\ g\ xs$ have the same length, so if the behavior of f somehow depends on the length of the list, no difference is discernible. Also, f can try to apply the predicate to list entries. So let x be some entry of the list xs, which means the respective entry in $map\ g\ xs$ is $g\ x$. Then $p \circ g$ applied to x is $p\ (g\ x)$, which is the same as p applied to $g\ x$. Thus the resulting truth value is the same in either instantiation and decisions based on this truth value are made in analogous manner on either side.
Both instantiations of f behave "the same", but $f\ (p \circ g)\ xs$ produces a list of type $[\mathsf{A}]$, while $f\ p\ (map\ g\ xs)$ produces a list of type $[\mathsf{B}]$. Yet the entries of the latter list are the results of applying g to the entries of the former list. So, g can as well be applied afterwards by mapping it over the result of $f\ (p \circ g)\ xs$, which is the left hand side of equation (2.1) (for arbitrary f).

So what are free theorems good for? If two different expressions are provably semantically equal, we can exchange one for the other in code without changing the meaning. Thus we can always use the one which can be evaluated faster. For example the left hand side of equation (2.1) will in general take more time than the right because g has to be computed twice for some list entries.

There is a third expression that is also semantically equivalent to either side of equation (2.1):

$$\mathit{foldr}\ (\lambda x\ \mathit{ys} \rightarrow \mathbf{let}\ y = g\ x\ \mathbf{in}\ \mathbf{if}\ p\ y\ \mathbf{then}\ y : \mathit{ys}\ \mathbf{else}\ \mathit{ys})\ [\,]\ \mathit{xs}$$

This equivalence is an example of short cut fusion [Gill et al., 1993] and can also be derived from free theorems though in a different way. The advantage of the third expression is that the list only has to be traversed once. Building intermediate list structures and then consuming them again right away takes up time and the last version saves this time by doing the mapping and filtering in one go. We will take a closer look at short cut fusion in section 4.9, when we prove similar results for Curry.

Free theorems can also help us to better understand a given polymorphic function. By observing the behavior of the function at one type we also learn something about how the function would behave at another type. In many cases there is some specific type that already determines the function's behavior completely, which for example is very useful for testing [Bernardy et al., 2010]. Other applications include automatic bidirectionalization [Voigtländer, 2009].

Finally, two warnings. Additional side conditions become necessary, as soon as we take partiality and other side-effects into account [Wadler, 1989]. In actual Haskell, the function f can create values of any type by using *undefined*. To account for this, g has to be strict, i.e., send *undefined* to *undefined*. We will discuss this more thoroughly in chapter 4.

The other warning is about different kinds of polymorphism. In the above example, *filter* is polymorphic, i.e., can be used for different types, but independently of the type the same code is used. We call this kind of polymorphism *parametric* [Strachey, 2000]. There is also ad-hoc polymorphism, which actually overloads a function name.

For example, the equality test (==) can also be used for many different types, but depending on the type of the arguments, different pieces of code are executed. This is reflected in the type, which is $\forall a.\mathsf{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \mathsf{Bool}$ and contains the type-class [Wadler and Blott, 1989] constraint $\mathsf{Eq}\ a$. The same type without the constraint, i.e., $\forall a.a \rightarrow a \rightarrow \mathsf{Bool}$, does not contain (==) and the free theorem for this type does not hold for (==).

2.2 Effects and Monads

2.2.1 Simulating State

Haskell functions are functions in the mathematical sense, i.e., the result is uniquely determined by the arguments. In particular, the result can only depend on the arguments and not on any global machine state like global variables or objects being allocated in a heap.

The following example is from an unpublished paper [Hutton and Fulger, 2008] and will play an important role throughout chapter 3. The task is simple: For the Tree type

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

we want to write a function

```
relabel :: Tree a → Tree Integer
```

replacing the entries at the leaves with different integers.

While *relabel* (Leaf x) = Leaf 0 is straightforward, the Node case is problematic: After *relabel* has been applied to the left subtree, we do not know which labels have been used. Since we cannot store the information in a counter variable, the recursive calls need to return the next free label along with the labeled subtree. Also, in order for the extra information to be of any use, the next recursive call has to take the next label as an additional argument. Therefore we need a helper function doing the actual work, while *relabel* will only call the helper. We define *label*:

```
label :: Tree a → Integer → (Tree Integer, Integer)
label (Leaf _)    n = (Leaf n, n + 1)
label (Node l r) n =
  let
    (l' , n' ) = label l n
    (r', n'') = label r n'
  in (Node l' r', n'')
```

The first rule uses the current label and returns the next label. The second rule passes the initial label n to the first recursive call, passes the now smallest unused label n' on to the second recursive call and then returns the final state n''. Using *label* we can define *relabel*:

```
relabel :: Tree a → Tree Integer
relabel t = fst (label t 0)
```

This solution does the job, but passing on the state by hand all the time is tedious and error-prone.

A better solution is to abstract stateful computations. We define a wrapper State, where s represents the type of whatever information we want to propagate and a is the computation's result type[2]:

```
newtype State s a = State {runState :: s → (a, s)}
```

[2] This usually is part of Control.Monad.State, though the library implementation is more complicated. The library does not provide the value level constructor State, so *state* should be used instead.

Computations of this kind can be combined:

```
(≫=) :: State s a → (a → State s b) → State s b
u ≫= v = State (λs →
  let
    (a, s′ ) = runState u s
    (b, s′′) = runState (v a) s′
  in (b, s′′))
```

The first argument of the *bind* operator is a stateful computation producing an a. The second argument is a function that takes an a (and implicitly also a state) and returns a b (and a state). Together they form a computation that produces a result of type b. We also define the following functions:

```
(≫) :: State s a → State s b → State s b
u ≫ v = u ≫= λ_ → v

return :: a → State s a
return a = State (λs → (a, s))

get :: State s s
get = State (λs → (s, s))

put :: s → State s ()
put s = State (λ_ → ((), s))

evalState :: State s a → s → a
evalState u s = fst (runState u s)
```

The (≫) operator allows to combine computations without using the result of the first one in the second one. The *return* function produces a trivial computation that simply returns a fixed value without touching the state. To access the state and manipulate it, *get* and *put* can be used, where *put* has the dummy return type (). Finally *evalState* is an alternative to *runState* that forgets the final state.
Using the State abstraction, we can give *label* and *relabel* in a cleaner form:

```
fresh :: State Int Int
fresh =
  get ≫= λn →
  put (n + 1) ≫
  return n

label :: Tree a → State Int (Tree Int)
label (Leaf x) = fresh ≫= λn → return (Leaf n)
label (Node l r) =
  label l ≫= λl′ →
  label r ≫= λr′ →
  return (Node l′ r′)
```

```
relabel :: Tree a → Tree Int
relabel t = evalState (label t) 0
```

Note how we can read *fresh* as an imperative program: read the state, call it n, set the state to be $n + 1$, return n.

The state abstraction is also how Haskell interacts with the outside world [Peyton Jones, 2002]. There is a special type constructor IO for input/output, which we can think of as State World, where World is some opaque primitive type representing the state of the outside world. A function like *putStrLn* :: String → IO () therefore can be thought of as taking a string and a world and producing a new world (and an empty tuple). The new world is much like the old one, only that the given string has been written to the standard output. The abstraction makes sure that whatever step is performed next is applied to the new world.
A World can never be accessed directly, i.e., one cannot look into it, manipulate it, duplicate it, dispose of it or generate a new one from scratch. Parts of the world can be accessed or changed, but only using a given set of predefined IO operations.

2.2.2 Monads

Propagation of a state is a kind of side-effect because in order to know what a function does we also have to take into account how the state changes and influences the result. A similar trick allows to perform nondeterministic computations [Wadler, 1985]: Instead of functions from A to B we use functions from A to [B]. Here, [ ] is the list type constructor, so we allow several results collected in a list but also none. Such list-valued functions can be combined in a similar manner:

```
concatBy :: [a] → (a → [b]) → [b]
xs `concatBy` f = concat (map f xs)
(!≫) :: [a] → [b] → [b]
xs !≫ ys = xs `concatBy` λ_ → ys
```

The function *concatBy* is like building unions: The lists $f\ x$ for all entries x of *xs* are concatenated. The (!≫) combinator may seem surprising at first because it only concatenates copies of *ys*, but is especially useful in combination with the *guard* function:

```
guard :: Bool → [()]
guard True  = [()]
guard False = []
```

Since failing computations are represented by empty lists, *guard* enforces some condition to be true. A successful computation needs to have some value, so we use a dummy entry (). Then *guard* b !≫ *ys* checks whether b holds and, if so, returns *ys*.
Using the above combinators, we can define the list of all Pythagorean triples (with entries at most 100):

```
[1 . . 100] ‘concatBy‘ λx →
[1 . . 100] ‘concatBy‘ λy →
[1 . . 100] ‘concatBy‘ λz →
guard (x^2 + y^2 == z^2) !≫
[(x, y, z)]
```

The same can of course be achieved using list comprehensions.

There are many more types of side-effect that can be simulated by similar tricks. A common abstraction covering all of them in homogeneous manner are monads [Wadler, 1992]:

```
class Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
(≫) :: (Monad m) ⇒ m a → m b → m b
u ≫ v = u ≫= λ_ → v
```

The *return* function converts a value to a computation that does nothing but yield the given result. The (≫=) operator, pronounced *bind*, takes an initial computation and a so called continuation, i.e., a computation depending on the result of the first computation, and combines both. The instance for State *s* uses the definitions of *return* and (≫=) we have already seen. The instance for lists is given by *return* $x = [x]$ and (≫=) = *concatBy*. Another example is computations that can fail, i.e., partial functions:

```
data Maybe a = Nothing | Just a
instance Monad Maybe where
  return         = Just
  Nothing ≫= v = Nothing
  Just x   ≫= v = v x
```

Failure is represented by Nothing and success by Just, where the field of Just is the result. When trying to pass the result of a failed computation to another computation, the overall computation fails as well. Otherwise, if there is an actual result, the continuation is applied to this result.

Monads have a special syntax in Haskell, the **do**-block. The definition

```
fresh :: State Int Int
fresh =
  get ≫= λn →
  put (n + 1) ≫
  return n
```

can be given equivalently as

```
fresh :: State Int Int
fresh = do
  n ← get
  put (n + 1)
  return n
```

thus hiding the ($\gg\!=$) and ($\gg$) operators and binding variables with a left arrow. The **do**-notation can be used for all monads, including user-defined type constructors with custom instances of Monad.

Every monad instance is required to satisfy three monad laws, which we only give for the sake of completeness:

$$\begin{aligned} \mathit{return}\ x \gg\!= v &= v\ x \\ u \gg\!= \mathit{return} &= u \\ u \gg\!= (\lambda x \to v\ x \gg\!= w) &= (u \gg\!= v) \gg\!= w \end{aligned}$$

The last law makes sure composition of computations is associative, which justifies the linear structure of **do**-blocks. The instances we have seen so far are all correct, but we will not carry out the proofs. We will encounter these laws and laws for other type classes a lot throughout this thesis.

2.3 Introduction to Functional-Logic Programming

Curry [Hanus, 2013] is a functional-logic language, combining concepts from both functional and logic programming. It is very similar to Haskell in many respects and as we assume the reader to know at least some Haskell, we will introduce Curry by explaining the differences.

2.3.1 Logic Features

Curry is nondeterministic, i.e., a program can have several results instead of just (or rather at most) one. There are several ways to introduce nondeterminism and one of them is the binary choice operator (?), which can return either of its arguments. This allows to define a function *coin*:

```
coin :: Int
coin = 0 ? 1
```

Unlike Haskell's *mplus* the binary choice operator does not need any effect wrapper (i.e., a monad) and simply has type (?) :: $a \to a \to a$.
Another way to introduce nondeterminism is by giving overlapping rules for a function. Consider the following example function computing all suffixes of a list:

$$\begin{array}{lll} \mathit{suffixes} :: [a] & & \to [a] \\ \mathit{suffixes} & l & = l \\ \mathit{suffixes} & (x : \mathit{xs}) & = \mathit{suffixes}\ \mathit{xs} \end{array}$$

The first rule always matches and states that any list is a suffix of itself. The second rule only applies to non-empty lists. In this situation the suffixes of xs are also suffixes of the longer list $x : \mathit{xs}$.
Thus *suffixes* $[1, 2]$ can evaluate to $[1, 2]$ using the first rule and also to *suffixes* $[2]$ using the second. In the latter case the next evaluation step can produce $[2]$ or *suffixes* $[\,]$. Since the empty list $[\,]$ does not match the pattern $(x : \mathit{xs})$, the call *suffixes* $[\,]$ can only be evaluated using the first rule, thus resulting in $[\,]$. The three possible results $[1, 2]$, $[2]$ and $[\,]$ are in fact all the suffixes of $[1, 2]$.
The same function definition is also valid in Haskell. Since the first rule always applies, the second rule is irrelevant and *suffixes* is just the identity function restricted to lists. If we wanted to compute all suffixes of a given list in Haskell, we would have to make the nondeterminism explicit by returning a list of suffixes.
Curry's choice operator can be implemented using overlapping rules as well:

$$\begin{array}{l} (?) :: a \to a \to a \\ x \mathbin{?} y = x \\ x \mathbin{?} y = y \end{array}$$

Conversely, overlapping rules can be represented by case splitting and binary choice, but this direction is more complicated (cf. the section on definitional trees in [Hanus, 2006]).

In Curry, variables can be *logic*, i.e., not bound to an expression a priori, but only determined by *constraints*. Consider the following alternative definition of the *suffixes* function:

$$\begin{array}{l} \mathit{suffixes}' :: [a] \to [a] \\ \mathit{suffixes}'\ l = \mathbf{let}\ p, s\ \mathbf{free\ in}\ p \mathbin{+\!\!+} s =:= l\ \&\!>\ s \end{array}$$

On the right hand side p and s are logic variables, which in our example represent lists. The $(+\!\!+)$ operator concatenates these lists and the result is compared to the given list l using *constraint equality*, written $(=:=)$. Finally the (&>) operator checks whether the constraint is satisfied and, if so, returns s as a result.
Writing the function like this reflects the definition of being a suffix: s is a suffix of l if there is a list p, such that the concatenation of p and s equals l. This is very close to how one would define suffixes in Prolog [Colmerauer and Roussel, 1996], i.e., `suffix(L,S) :- append(P,S,L)`. The two implementations *suffixes* and *suffixes′* are not really equal, but we defer the discussion of the (somewhat subtle) difference to the section about laziness.

In Curry, patterns do not have to be linear, i.e., the same variable may appear multiple times in a pattern. Consider for example the following function performing a lookup in an association list:

```
lookup :: a → [(a, b)]      → b
lookup    k   ((k, v) : _ ) = v
lookup    k   (_      : ps) = lookup k ps
```

The variable k appears twice in the pattern of the first rule. This is a shorthand notation for $lookup\ k\ ((k', v) : \_) = k =:= k'\ \&{>}\ v$, making the pattern linear by introducing a constraint. Like in Haskell, the underscore is a wild-card pattern, i.e., represents parts of patterns that do not appear on the right hand side of the rule and thus do not need to have a name.

2.3.2 Failure and Non-Strictness

Like Haskell, Curry is non-strict, i.e., an expression can have one or more results even if it contains a subexpression that does not. For example **let** $x =$ **failed in** True results in True even though the primitive **failed** by its definition admits no result at all. In both languages this is implemented by lazy evaluation, i.e., if a variable is bound to an expression, the expression is stored in a heap and remains unevaluated until evaluation is forced or the garbage collector disposes of it. In Curry, failing computations do not cause the program to abort but to backtrack, making **failed** a natural counterpart to choice.

Constraint equality (=:=) however is strict, i.e., two expressions are only equal if they can be reduced to equal constructor terms. An infinite list cannot be written out using finitely many constructors and so does not equal any list (not even itself). Thus if we call *suffixes′* (the latter version using concatenation and constraint equality) for an infinite list l, there is no way to satisfy the constraint $p +\!\!+\, s =:= l$ and no result will be returned. On the other hand, *suffixes* (the former, recursive version) works well for infinite lists because it can return whatever is left of the given list after having cut off some entries.

2.3.3 Sharing

Sometimes intermediate results are supposed to be shared between different parts of a program. In Haskell this is only a question of runtime: Evaluating the same expression twice will result in the same value and is thus unnecessary but not harmful. In a nondeterministic language, evaluating the same expression twice may yield two different results and therefore sharing becomes a question of semantic relevance.

In Curry, variables and functions are two separate concepts and knowing the difference is important because it influences sharing. In a nutshell, variables represent one value throughout their scope and every lookup will produce the same result (this is referred to as *call-time choice nondeterminism*). Functions on the other hand are evaluated independently every time they are invoked.

Whether an identifier is a variable or a function is not determined by its type, but by where and how the identifier is introduced. Functions are introduced by rules either on the top-level or locally in a **let** or **where** clause. Functions can also be imported from other modules or be predefined by the language. Variables are always local and can be

introduced in various different places: as function arguments (both in rules and lambda abstractions), in **case** statements or in **let** or **where** clauses.
Unfortunately, there is a small syntactic overlap, where, according to the above, an identifier could be either: The local binding **let** $f = \ldots$ **in**... could be interpreted as a nullary local function or as a trivial pattern only consisting of the variable f. This actually makes a difference in some cases, so the ambiguity has to be resolved somehow and is decided in favor of the identifier being a variable. The same problem does not occur on the top-level, because variables are never in the global scope. Even the mathematical constant π therefore counts as a function, if it is defined on the top-level.
There are different *case modes*, i.e., conventions for how functions, constructors and variables should be named. Coming from a Haskell background, we use Haskell mode – uppercase for constructors and lowercase for functions and variables. In order to still be able to tell the difference between variables and functions, variable names will be letters and function names will be words. Here, we take letters to also include x_n, xs and x'.
Functions have an arity, which is the number of arguments given in the function's rules (this number has to coincide for all rules). A partial application of a function can be passed around and shared. As soon as the application is saturated, i.e., the number of given arguments is (at least) the arity, not the expression will be shared but its result.

Let us discuss some examples. Since *coin* is defined on the top-level, it is a function. Therefore $\mathit{coin} + \mathit{coin}$ contains two independent calls, both of which can yield 0 or 1 and the sum has three possible values: 0, 1 and 2.
If we want to use the same value twice, we have to bind it to a name. The expression **let** $c = \mathit{coin}$ **in** $c + c$ only has two possible values (0 and 2), because c is a variable and both occurrences of c refer to the same value (either 0 or 1). Results are also shared whenever they are passed to a function as an argument. If we define a function *double*

$$
\begin{array}{l}
\mathit{double} :: \mathsf{Int} \to \mathsf{Int} \\
\mathit{double}\ n = n + n
\end{array}
$$

then the expression *double coin* also only has 0 and 2 as results. This behavior is called *call-time choice* because one should think of the choice being made when the function is called (actually, the choice is only made when the argument is needed).
If we want to inline the *double* function we have to make sure the argument is still shared. Thus *double coin* may be replaced by **let** $c = \mathit{coin}$ **in** $c + c$, but not by $\mathit{coin} + \mathit{coin}$.
Conversely, we sometimes want to disable sharing explicitly, but without moving the nondeterministic computation to the top-level. This can be done using dummy arguments as in **let** $\mathit{delayed}\ () = \mathit{coin}$ **in** $\mathit{delayed}\ () + \mathit{delayed}\ ()$. Here, we have a unary local function *delayed*, which can only be evaluated after an argument has been given, even though the argument does not appear in the body. Here, 1 is a possible result because the function *delayed* is evaluated twice and every copy has its own result of *coin*.

To get a feeling for the interaction between higher-order features and nondeterminism, let us take the example a bit further. The expression $\mathit{map}\ ((+)\ \mathit{coin})\ [0, 0]$ only produces

$[0, 0]$ and $[1, 1]$, but not $[0, 1]$ or $[1, 0]$ because *coin* is only called once and produces either 0 or 1. The same is true for **let** $f = (+)$ *coin* **in** *map* f $[0, 0]$, where f is a local function-typed variable.
What if we made the partial application $(+)$ *coin* a function? Consider two possible ways to do so:

$$\begin{aligned}&\mathit{mayInc0} :: \mathsf{Int} \to \mathsf{Int}\\&\mathit{mayInc0} = (+)\ \mathit{coin}\\&\mathit{mayInc1} :: \mathsf{Int} \to \mathsf{Int}\\&\mathit{mayInc1}\ x = \mathit{coin} + x\end{aligned}$$

While *map mayInc0* $[0, 0]$ still only produces lists with equal entries, *map mayInc1* $[0, 0]$ suddenly also yields $[0, 1]$ and $[1, 0]$ as possible results. The former function is nullary, so every call is a saturated application, and in particular, *map mayInc0* $[0, 0]$ is the same as *map* $((+)$ *coin*$)$ $[0, 0]$. The function *maynInc1* is unary, and since no argument is given to it in *map mayInc1* $[0, 0]$, it is passed to the higher order function *map* unevaluated. The function *map* then applies *mayInc1* twice and both applications are evaluated independently, which leads to additional results.
The two functions *mayInc0* and *mayInc1* seem to be the same because of eta equivalence. Yet to the contrary, they form a counter-example to eta equivalence in Curry. The example also shows the importance of the concept of arity and that arity is not encoded in the type.

Both functions and variables can be defined recursively, but due to the different ways of handling nondeterminism, the two kinds of recursion behave differently. If we define a (nullary) top-level function *zeros* by

$$\begin{aligned}&\mathit{zeros} :: [\mathsf{Int}]\\&\mathit{zeros} = [\,]\ ?\ 0 : \mathit{zeros}\end{aligned}$$

it will produce all lists only having entries 0. Since every call is evaluated independently, the stack of recursive calls may end at any level if the left alternative is chosen.
If on the other hand we define a variable **let** $zs = [\,]\ ?\ 0 : zs$ **in**... it can evaluate to two head normal forms $zs = [\,]$ and $zs = 0 : zs$. Once either head normal form has been chosen, all subsequent lookups result in the same head normal form. In particular, if the latter has been chosen, the recursive lookup in the tail refers back to the list zs itself. Viewed from the outside, two lists are produced: An empty list and an infinite list, but no nonempty finite list.

2.3.4 Type System

There are some features of Haskell that are not inherited by Curry, especially when it comes to the type system. Most prominently, Curry – at least the current mainstream version – has no typeclasses [Wadler and Blott, 1989]. The absence of this feature leads to various small changes. Numeric literals and operators cannot be overloaded and have to be disambiguated by hand. For example, addition for floats has to be written as (+.) to distinguish it from integer addition and the monadic bind ($\gg=$) can only be used for the IO monad.

On the other hand, the equality test (==) is overloaded in an ad-hoc manner and can be used for any data type. It checks whether the constructors match and descends recursively to the fields like the generic instance in Haskell would. If one wants to use some kind of equivalence relation, it has to have a different name. Unfortunately, the type system does not prohibit testing functions for equality, but programs may fail at runtime if the equality test is executed.

This brings us back to the two functions *suffixes* and $\mathit{suffixes}'$. Another difference lies in the fact that the more functional version *suffixes* never inspects list entries but just passes them on, while $\mathit{suffixes}'$ checks them for equality. Thus when trying to use $\mathit{suffixes}'$ for a list of functions, it might not give any result or cause the program to abort.

There are some more features around Haskell's type system that could be inherited by Curry but are not: Curry does not have higher rank types, i.e., functions can be polymorphic but all type variables are (implicitly) all-quantified outside the type expression. Also, there are no type variables of kind other than $*$ and no generalized algebraic data types.

Chapter 3

Understanding Idiomatic Traversals

Contents

This chapter is a study of idiomatic traversals, which are the functional programming counterpart of iterating over a data structure. In Haskell this is modeled by the typeclass Traversable, which provides a function

$$\mathit{traverse} :: (\mathsf{Traversable}\ t, \mathsf{Applicative}\ a) \Rightarrow (x \to a\ y) \to t\ x \to a\ (t\ y),$$

that allows to apply an effectful function to every entry of some structure, collecting the results in a structure of the same shape and combining the effects into a single operation. Typical examples for traversable structures are lists, trees of all sorts, maps and arrays. The Traversable interface abstracts both over the type of effect (represented by a for Applicative in the above signature) and the data structure at the same time, making it a valuable tool in producing well-structured and reusable code. Since version 4.8.0.0 of GHC's base package, Traversable is part of the Prelude.[1]

The Traversable typeclass has been introduced in [McBride and Paterson, 2008] along with the typeclass Applicative (appearing as a constraint in the above signature). Applicatives are also called *idioms* and we will use both terms with the understanding that Applicative is the implementation and idiom is the concept. Idioms are similar to monads [Moggi, 1991] in that they allow to combine effectful computations. The ways to do so are restricted in comparison to monads, but on the other hand idioms are a real superclass of monads, i.e., there are additional instances.

Even though monads are ubiquitous in Haskell, the natural way to handle effects in the context of traversals are idioms. Traversals generally do not need the full power of monads, so it would be an arbitrary restriction to only allow those with monadic effects. Even worse, this restriction would exclude traversals in the Const idiom, which allows to understand monoids as idioms and makes folds a special case of traversals. Later we will see more reasons why idioms and traversals make such a good match. The Applicative class has also gained a lot of importance in version 4.8.0.0 of GHC's base package, as it is now defined in the Prelude, where it is a superclass of Monad. We will take a deeper look at applicatives in section 3.1.

The point of typeclasses is having a joint interface to analogous functionalities of different types. The implementation of the methods should be irrelevant to whoever writes code calling the methods. When we *reason* about code using typeclasses, we also want to have a 'joint interface'. Equational reasoning about functions abstracted over typeclasses should not rely on the implementations of the methods for any particular instance. Thus, the methods need a counterpart for arguing, which are laws.

A proof relying only on the laws of some typeclass interface is guaranteed to work equally well for all (lawful) instances. Even if there only is one instance we are interested in, there still is good reason to conduct the proof in an abstract manner: Should the implementation ever be changed, an abstract proof remains valid.

Still another reason applies to the IO monad handling the connection between programs and the outside world. This monad cannot be implemented in the language itself, so we

[1] http://hackage.haskell.org/package/base-4.8.0.0

are left with the choice between a black box and Pandora's box: Either we treat IO as an opaque primitive and learn to reason about it in an appropriate manner or we unfold the definition and give up our privilege of arguing about functional programs rather than imperative programs.

Gibbons and Hinze [2011] give a number of example proofs for simple monadic programs. These proofs are conducted on the interface level and do not rely on implementations. Can something similar be achieved for the Traversable class? What are suitable laws and how can we reason about traversals without relying on their implementations?

McBride and Paterson [2008] give laws for the Applicative class ((3.3) – (3.6)) and though these laws take getting used to, there is no serious discussion about whether this set of laws is appropriate. When it comes to the Traversable class, the same paper gives a good intuition what a traversal should do, but no clear characterization. The paper does give lists and trees as examples and offers a recipe for defining a suitable traversal for all tree-like data types. The paper does however not specify whether there might be other equally good traversals or more generally what constitutes a good traversal. For example the recipe always gives depth-first traversals (just like GHC's automatic deriving mechanism does), but in some situations a breadth-first traversal might be better suited. On the negative side, McBride and Paterson [2008] discuss the partially applied function arrow $(\rightarrow)$ X (where X is some fixed concrete type) and reason that it is not traversable. This somehow points to only finite structures being traversable, i.e., the number of entries has to be a finite number.

A more detailed discussion can be found in [Gibbons and Oliveira, 2009]. There some properties are listed which seem reasonable as laws and which apply to traversals deemed correct in an intuitive sense. Using a dummy applicative that does not allow any real effects, mapping a pure function over the structure can be formulated as a traversal. One of the laws (which later was dubbed *identity-* or *unitarity law*) basically forces this 'mapping by traversing' to coincide with the predefined functor instance. This law rules out all traversals that change the shape of the structure in any way, i.e., traversals that drop, reorder or duplicate parts of the structure. Also, if a traversal does not visit all entries, the unvisited entries inevitably are lost, which violates the identity law.

The only type of unwanted behavior that seems hard to rule out is duplication of effects. A traversal might apply the effectful function to some entry more than once and thus trigger unnecessary effects, but then only use one of the results. Such traversals are called *duplicating* or *duplicitous* and Gibbons and Oliveira [2009] wanted to rule them out, but it remained a challenge to find a good formulation to do so.

The answer turned out to be a law later called *composition-* or *linearity law*. This law is due to Ross Paterson and was published in [Gibbons and Oliveira, 2009] already, but neither of them realized the full potential. Jaskelioff and Rypacek [2012] show that a particular minimal example of a duplicating traversal violates the composition law. They suggest this law might be enough to detect duplicating behavior in general. Thus Gibbons and Oliveira [2009] seem to have given all the laws for traversals already and Jaskelioff and Rypacek [2012] propose to change the class definition to actually include these laws. They give a number of justifications for their proposal, one of which is the aforementioned insight about the surprising connection between the composition law and duplicating traversals. Another point the paper makes is that a reasonably big class of types called *finitary containers* [Abbott et al., 2003] are lawfully traversable w.r.t. these laws. Finally the laws have a good motivation in category theory. We will discuss the Traversable class and its laws (plus some easy consequences) in section 3.2.

The laws having surprising and far reaching consequences seems to be a good thing, but there is a downside. Given a question about some concrete traversal, it is rather unclear how to use them to answer the question. Gibbons and Oliveira [2009] themselves did not immediately realize the full implications of their laws, so how can the average Haskell programmer be expected to do so? One concrete question concerning a traversal appears in [Hutton and Fulger, 2008]. For a given tree type, they define a traversal that replaces all entries with consecutive numbers. They then prove that collecting all used labels produces a list with no duplicates.
The obvious generalization is to ask the same question for arbitrary traversable types: Does a lawful traversal behave the same way, i.e., will labeling and then collecting the labels lead to a list without duplicates? This question has been answered positive in our paper [Bird et al., 2013] for a slightly different labeling and collecting strategy and using a lot of machinery. In this thesis I first take a different path and prove the claim using the laws only (section 3.3). Also I prove the claim in (a direct generalization of) the original formulation and do not alter the labeling process in any way as we did in the paper.
As often, being restricted to primitive means makes things harder and the proof is neither straightforward nor pretty. Yet it serves to show something can be done using the laws only. It however also serves to show that this is not a good way to approach the matter, as auxiliary concepts have to be introduced which are hard to guess.

The problem regarding the laws seems to be that they do not provide a good intuition for what traversing actually does. We would like to reason about traversals like we reason about iterators, using notions of 'shape', 'contents', 'entry' and entries having some order. Such notions exist in the context of finitary containers, which are traversable as Jaskelioff and Rypacek [2012] have shown.
A finitary container is given by a set of *shapes*, each of which has an *arity*. An n-ary shape can be filled with n objects of some common, arbitrary type, which thus become the *contents*. Conversely, given a container we can dissect it into its shape and its contents. The obvious way to traverse such a structure is to first dissect it, traverse the list of contents and use the list of results to then fill the shape again. Finitary containers will be discussed in section 3.4.

The proof for the generalized tree labeling claim we gave in [Bird et al., 2013] consists of two steps. The first step is proving a general statement about lawful traversals called *representation theorem*. The theorem states that every lawfully traversable type is indeed a finitary container, so the converse of the theorem by Jaskelioff and Rypacek [2012] is also true and both concepts are actually the same. Moreover, every lawful traversal

corresponds to the standard traversal for finitary containers. This allows us to switch back and forth between the two different viewpoints.
The theorem uses a somewhat nonstandard notion of shape, which is *make functions*. A make function is a polymorphic function of any arity returning a traversable (subject to some laws). The make function therefore intuitively is a shape with a number of holes. In section 3.4 we will give a more formal correspondence between make functions and usual notions of shape. This section is more theoretic than the rest of this chapter and uses some concepts from category theory. We will not need any results from this section later – it merely serves to shed some light on the notion of make function and its connection to finitary containers.
The proof of the representation theorem is constructive, i.e., the proof reveals how to actually find the shape/contents decomposition of a traversable. This is done by a special traversal using the so called *batch idiom*, which is an example of a *free idiom* [Capriotti and Kaposi, 2014]. The result of the special batch traversal determines the result of all other traversals, in particular shape and contents can be extracted. Section 3.5 contains the representation theorem together with its proof.

In [Bird et al., 2013] the second step in proving the tree labeling claim is deriving a new law about monadic traversals called *inversion law*: If one effectful function cancels the effects of another, i.e., they satisfy

$$u\ x \gg\!\!= v = \mathit{return}\ x \tag{3.1}$$

then also

$$\mathit{traverse}\ u\ t \gg\!\!= \mathit{treverse}\ v = \mathit{return}\ t \tag{3.2}$$

holds where *treverse* is a backwards traversal of the data structure. Applying this inversion law to a particular pair of functions directly leads to the desired statement about labeling. The statement is not completely the same though, as the inversion law can only prove results about invertible functions. Thus, a slight reformulation of the labeling procedure is necessary to satisfy the invertibility requirement. Both the reformulation of the claim and the inversion law are by Richard Bird and Jeremy Gibbons.
Even though the representation theorem has initially been developed to prove the inversion law, once the theorem is at hand there is an easier proof of the labeling claim in its original formulation. So in this thesis the inversion law only appears as an example application of the representation theorem (the direct proof being another one).

The equivalence between traversable types and finitary containers has been proved independently by Jaskelioff and O'Connor [2015]. The statement given therein is more general – for example it also covers containers with several entry types or traversals where the effect wrapper is something weaker than an idiom. The proof relies on similar ideas, but is presented in a very different manner – most notably, a lot of category theory is used. They also provide a machine-checkable version of the proof written in Coq.
The category theory approach by Jaskelioff and O'Connor [2015] and my approach by equational reasoning are two complementary views of the same phenomenon. The batch

idiom has a very clear, category theoretical motivation, which is connected to being a (particular) free idiom. On the other hand, the batch idiom can be given in a very concrete form in Haskell. Taking the latter viewpoint, all the properties of the batch idiom can be checked by hand without knowing anything about the theoretical background. Another advantage of the concrete approach is, that it is by its very nature easier to apply. The representation theorem is formulated in Haskell and ready to be used for equational reasoning. We will see some examples in section 3.6 and further examples can be found in [Matsuda and Wang, 2015].

In this chapter we take the somewhat artificial viewpoint of our language being total. We do not allow general recursion, but only structural recursion like in figure 3.1. Therefore we do not have undefined or partially defined values. Most of all, the list type does not contain infinite lists. If we insisted on having them, we could always add a constructor with type $(\mathsf{Nat} \to a) \to [\,a\,]$ to include infinite lists.
There are three main reasons for this alteration. First of all, some effects do not interact well with infinite structures. If a traversal runs a potentially failing computation for every entry of some structure, the overall computation succeeds if and only if every single computation does. If the number of entries is infinite and all computations succeed, the program does not terminate (while failure can be detected after a finite number of iterations). Instead of having to repeat the restriction of equations only holding for total values, we incorporate the restriction into the global setting. Second, not having infinite data structures enables proofs by induction, which we will do a lot. And finally, one of the steps in the proof of the representation theorem crucially relies on every value being given by a finite number of constructors.

Finally, some conventions regarding the names of types. Even though it is common practice to use a, b, c and so on for type variables, we use x, y and z for arbitrary types. The letters a, b and c are reserved for applicatives.

3.1 Idioms

In the introduction of this chapter we have argued that the natural effect system to consider in the context of traversals is idioms, which we shall thus take a closer look at. Idioms have only recently claimed their place as a superclass of monad and remain the less known effect system. Therefore we start with monads and define Applicative as a restricted interface. Later we will encounter additional instances for this restricted interface.

3.1.1 Monadic Idioms

Consider **do**-blocks of the special form

do

$x_0 \quad \leftarrow u_0$

$\dots$
$x_{n-1} \leftarrow u_{n-1}$
$\mathit{return}\ (f\ x_0 \dots x_{n-1})$

where no x_i appears in any of the u_i. Any **do**-block can be adjusted to end in a *return* statement using the monad laws. Also if a line does not contain a binding $x_i \leftarrow$ we can always add a dummy variable and then ignore the corresponding argument in the function f. The only crucial restriction is none of the u_i depending on any of the x_i. Take stateful computations as an example. The state will be passed on from one computation to the next, but no computation can depend on the *result* of an earlier one. In the final expression, all results can be used, but no further changes can be made to the state. If the effect is nondeterminism, no computation u_i can depend on any previous choice, making the search space an n-fold Cartesian product.
This gives us a good initial understanding of what idioms can do. When it comes to the syntax McBride and Paterson [2008] propose

$$[\![f\ u_0 \dots u_{n-1}]\!],$$

which is shorter than the **do**-block as it omits naming all variables. The standard way to write the above **do**-block in idiomatic syntax however is:

$$\mathit{pure}\ f \circledast u_0 \circledast \dots \circledast u_{n-1}$$

The $(\circledast)$ operator is called *idiomatic application* and should be thought of as a variant of function application that is able to deal with effects. It is thus application in a figurative or *idiomatic* sense. In particular, idiomatic application – like actual application – associates to the left. If the u_i are atomic (i.e., not the result of subcomputations combined with $(\circledast)$) the above is sometimes referred to as *canonical form* of an idiomatic computation. For monadic idioms we will switch back and forth freely between the **do**-block notation and idiomatic syntax.
The class declaration is given by:

```
infixl 4 ⊛
class Applicative a where
  pure :: x → a x
  (⊛)  :: a (x → y) → a x → a y
liftA :: Applicative a ⇒ (x → y) → a x → a y
liftA f u = pure f ⊛ u
```

In the GHC library[2] the Applicative class has Functor as a superclass constraint. Here, we will however ignore this constraint and use *liftA* (from Control.Applicative) instead of *fmap* (which is a method of the Functor class).

[2] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Applicative

```
data Tree x = Leaf x | Node (Tree x) (Tree x)
fresh :: State Int Int
fresh = do
  n ← get
  put (n + 1)
  return n
label :: Tree x       → State Int (Tree Int)
label    (Leaf x)     = do
  n ← fresh
  return (Leaf n)
label    (Node l r)   = do
  l′ ← label l
  r′ ← label r
  return (Node l′ r′)
relabel :: Tree x → Tree Int
relabel   t          = evalState (label t) 0
```

Figure 3.1: Tree labeling using monadic syntax

Figure 3.1 shows the labeling procedure defined by Hutton and Fulger [2008]. They also give an equivalent way to define *label* using the applicative interface:

```
label :: Tree x      → State Int (Tree Int)
label   (Leaf x)     = pure Leaf ⊛ fresh
label   (Node l r)   = pure Node ⊛ label l ⊛ label r
```

We cannot translate the **do**-block in the definition of *fresh* into applicative syntax though, as the *put* $(n+1)$ statement uses the variable n bound by the previous statement. Here, the Applicative instance for State s is:[3]

```
instance Applicative (State s) where
  pure x = State (λs → (x, s))
  State u ⊛ State v = State (λs →
    let (f   , s′)  = u s
        (x   , s″) = v s′
    in  (f x, s″))
```

Actually there is no need to give this instance in detail, as for all monads $pure\ x = return\ x$ and $u \circledast v = ap\ u\ v$ gives the corresponding applicative instance. Here, *ap* is defined like this:

[3] We assume the definition **newtype** State s x = State $\{runState :: \lambda s \to (x, s)\}$ from the introduction (section 2.2).

```
ap :: Monad m ⇒ m (x → y) → m x → m y
ap u v = do
  f ← u
  x ← v
  return (f x)
```

Every monad is an idiom and every function written in applicative syntax with effects in that monad can be translated back to monadic syntax. We will take the viewpoint of the applicative instance being uniquely determined by the monad instance as above and thus we do not give any further applicative instances for monads explicitly.

3.1.2 Other Idioms

Even though there are other reasons for restricting the interface, it would still seem kind of pointless to introduce a superclass of Monad that does not have any additional instances. The number of (atomic) non-monadic idioms is small, but there are some. In addition there are some constructions that produce new idioms out of given ones. This section only describes the constructions we will use at some point, but many more can be found in [Paterson, 2012] and [Capriotti and Kaposi, 2014]. These constructions can of course be applied to monads, but then the result only is an idiom in general.

Atomic Non-monadic Idioms

Lists are an instance of Monad and represent nondeterministic computations. Therefore they also have an Applicative instance. There is another possible idiom structure for lists with no corresponding monad structure, which McBride and Paterson [2008] give as one of their motivating examples for the Applicative class. To not get confused with the standard instance, the wrapper ZipList is used (cf. Control.Applicative[4]). Ziplists should be thought of as (finite or infinite) streams of values that can be combined at every point in time.

```
newtype ZipList x = ZipList [x]
instance Applicative ZipList where
  pure x                 = ZipList (repeat x)
  ZipList fs ⊛ ZipList xs = ZipList (zipWith (λf x → f x) fs xs)
```

The *pure* method produces an infinite list of copies of the same element. The i-th entry of *zipWith* $(\lambda f\ x \to f\ x)$ *fs xs* is the i-th entry of *fs* applied to the i-th entry of *xs*. That is, unless one of the argument lists is too short to have an i-th entry, in which case the result also ends.
The functional reactive language Elm[5] uses a similar concept. There the Signal type constructor represents values depending on the actual time and has an interface similar to Haskell's Applicative interface.

[4] `http://hackage.haskell.org/package/base/docs/Control-Applicative.html#t:ZipList`

[5] `http://elm-lang.org/`

The second important idiom that is not a monad is the Const type constructor, which is introduced as Accy (for accumulating) in [McBride and Paterson, 2008]. The following definition can be found in Control.Applicative[6]:

```
newtype Const x y = Const {getConst :: x}
instance Monoid n ⇒ Applicative (Const n) where
  pure _                  = Const mempty
  Const n₁ ⊛ Const n₂ = Const (mappend n₁ n₂)
```

The constraint Monoid N makes sure values of type N can be combined using some associative function *mappend* :: N → N → N, where *mempty* :: N is the neutral element. This idiom can be used to reformulate the collecting procedure of the labeling example. First, we give the original formulation used by Hutton and Fulger [2008] (We call the function *contents* instead of *labels* to avoid confusion with *label*):

```
data Tree x = Leaf x | Node (Tree x) (Tree x)
contents :: Tree x       → [x]
contents    (Leaf x)    = [x]
contents    (Node l r) = contents l ⧺ contents r
```

The monoid instance for lists is given by list concatenation (i.e., *mempty* = [] and *mappend* = (⧺)). Thus we can rewrite *contents* to also use applicative syntax:

```
note :: x  → Const [x] y
note    x  =  Const [x]
collect :: Tree x       → Const [x] (Tree y)
collect    (Leaf x)    = pure Leaf ⊛ note x
collect    (Node l r) = pure Node ⊛ collect l ⊛ collect r
contents :: Tree [x] → [x]
contents    t          = getConst (collect t)
```

To not change the type of *contents* we introduce a function *collect* doing the actual work (like *label* does for the labeling part). The new implementation of *contents* then simply removes the wrapper (like *relabel* does). The function *note* is only for convenience, as $(\lambda x \rightarrow$ Const $[x])$ is long and Const ∘ (:[]) is cryptic.

Constructions on Idioms

Type constructors can be composed (cf. Data.Functor.Compose[7])

```
newtype Compose f g x = Compose {runCompose :: f (g x)}
```

and if both *f* and *g* are idioms, then so is their composite [McBride and Paterson, 2008]:

[6] http://hackage.haskell.org/package/base/docs/Control-Applicative.html#t:Const

[7] http://hackage.haskell.org/package/transformers/docs/Data-Functor-Compose.html

```
instance (Applicative a, Applicative b) ⇒ Applicative (Compose a b) where
  pure x                  = Compose (pure (pure x))
  Compose f ⊛ Compose x = Compose (pure (⊛) ⊛ f ⊛ x)
```

The idiomatic application written as a section belongs to the inner idiom (i.e., b), while all other applicative methods on the right hand side of the second rule belong to the outer idiom (i.e., a).

Note that the same is not true for monads – even if M and M′ are monads, Compose M M′ does not have to be a monad. One way to still be able to combine monads is using monad transformers, but this construction uses additional input (the "transformer version" of one of the monads) and is more complicated [Liang et al., 1995].

Composition of idioms is associative in some sense we will make precise in section 3.1.5. There is also a neutral element, which is an idiom that does not allow for any actual effects but is just a wrapper allowing to treat pure functions as effectful.

```
newtype Identity x = Identity {runIdentity :: x}
instance Applicative Identity where
  pure x                    = Identity x
  Identity f ⊛ Identity x = Identity (f x)
```

In fact Identity is even a monad, though we will never use this.

In section 3.6 we will need another construction on idioms, which is reversing the order of effects. Gibbons and Oliveira [2009] define a wrapper Backwards and give an applicative instance for Backwards A where A is any idiom:

```
newtype Backwards a x = Backwards {forwards :: a x}
instance Applicative a ⇒ Applicative (Backwards a) where
  pure x                      = Backwards (pure x)
  Backwards u ⊛ Backwards v = Backwards (pure (λx f → f x) ⊛ v ⊛ u)
```

In [Gibbons and Oliveira, 2009] the inverse of the value constructor Backwards is called *runBackwards*, but here, we stick to Control.Applicative.Backwards[8] and call it *forwards*. Note that idiomatic application in the backwards idiom relies on idiomatic application in the original (forwards) idiom with the order of the arguments switched.

Backwards idioms are a curious phenomenon and their existence teaches us something about idioms in general. IO is a monad, therefore it is also an idiom and thus there has to be a Backwards IO idiom. Using this idiom one can write programs that perform IO actions in reversed order. So can you write a reversed echo program, that writes a character to the output before reading the character from the input? Obviously you cannot, but neither can you combine both IO actions in the other direction using only applicative syntax. Passing the result of the reading action to the writing action requires the monadic bind operator (≫=), which idioms do not provide.

[8] `http://hackage.haskell.org/package/transformers/docs/Control-Applicative-Backwards.html`

3.1.3 Idiom Laws

The Applicative class is governed by the following laws, also given by McBride and Paterson [2008]:

$$\mathit{pure}\ \mathit{id} \circledast v = v \tag{3.3}$$

$$\mathit{pure}\ (\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w) \tag{3.4}$$

$$\mathit{pure}\ f \circledast \mathit{pure}\ x = \mathit{pure}\ (f\ x) \tag{3.5}$$

$$u \circledast \mathit{pure}\ x = \mathit{pure}\ (\lambda f \to f\ x) \circledast u \tag{3.6}$$

Law (3.3) is called *identity law* and makes sure $\mathit{liftA}\ \mathit{id} = \mathit{id}$. The second law is called *composition law* and makes sure combining effects is associative. On the left an additional $\mathit{pure}\ (\circ)$ is needed to make the types match. The *homomorphism law* (3.5) makes sure idiomatic application extends usual application. Finally the *interchange law* (3.6) allows to switch the order of pure and effectful computations (while two effectful computations may not switch their order).

One way to understand the laws is that they allow to bring any computation into its canonical form [McBride and Paterson, 2008]

$$\mathit{pure}\ f \circledast u_0 \circledast \ldots \circledast u_{n-1}$$

where the u_i are not nested idiomatic applications. The composition law (3.4) can be used from right to left to avoid computations that nest to the right. All pure computations can be swapped to the left by the interchange law (3.6) and then combined into one by the homomorphism law (3.5). The identity law (3.3) is only needed for the special case of no computation being pure. In this case a 'dummy' *pure id* can be added without changing the meaning.

The laws imply that *liftA* satisfies the functor laws. While $\mathit{liftA}\ \mathit{id} = \mathit{id}$ is immediate from the identity law (3.3), showing

$$\mathit{liftA}\ (g \circ f) = \mathit{liftA}\ g \circ \mathit{liftA}\ f$$

needs some more work:

$$\begin{array}{ll}
 & \mathit{liftA}\ (g \circ f)\ u \\
= & \langle \text{ definition of } \mathit{liftA} \ \rangle \\
 & \mathit{pure}\ (g \circ f) \circledast u \\
= & \langle \text{ homomorphism law (3.5) } \rangle \\
 & \mathit{pure}\ ((\circ)\ g) \circledast \mathit{pure}\ f \circledast u \\
= & \langle \text{ homomorphism law (3.5) } \rangle \\
 & \mathit{pure}\ (\circ) \circledast \mathit{pure}\ g \circledast \mathit{pure}\ f \circledast u \\
= & \langle \text{ composition law (3.4) } \rangle \\
 & \mathit{pure}\ g \circledast (\mathit{pure}\ f \circledast u) \\
= & \langle \text{ definition of } \mathit{liftA} \ \rangle \\
 & \mathit{liftA}\ g\ (\mathit{liftA}\ f\ u)
\end{array}$$

For all monadic idioms the idiom laws are consequences of the monad laws. For the Const idiom, the laws rely on the monoid laws (i.e., associativity and left and right neutrality).

3.1.4 Flattening Formula

Besides the laws themselves, another useful equation for dealing with applicative syntax is the so called *flattening formula*. Given two computations in canonical form, it gives – also in canonical form – the result of (idiomatically) applying one computation to the other. The formula uses some shorthand notations. The first simply gives a shorter way to write canonical forms:

$$\mathit{pure}\ f \circledast_{i=0}^{n-1} u_i = \mathit{pure}\ f \circledast u_0 \circledast \ldots \circledast u_{n-1}$$

This operator also associates to the left and has the same precedence as a single ($\circledast$). The second shorthand notation is a generalized application operation:

$$(g \circ_{m,n} f)\ x_0 \ldots x_{n-1} = g\ x_0 \ldots x_{m-1}\ (f\ x_m \ldots x_{n-1})$$

Some noteworthy special cases are:

$$\begin{array}{lllll}
\circ_{0,0} = (\$) :: & (y \to z) \to & & y \to & z \\
\circ_{0,1} = (\circ) :: & (y \to z) \to & (x \to y) \to & x \to & z \\
\circ_{1,1} = \mathit{flip} :: & (x \to y \to z) \to & & y \to x \to & z
\end{array}$$

Both shorthand notations are not function definitions in Haskell in a strict sense. They can possibly be made actual code using Template Haskell or some of GHC's more advanced type level features. Here, we will treat them as metasyntactic abbreviations: For every concrete choice of m and n the meaning is clear from the above equations.

Lemma 3.1.1. The flattening formula

$$(\mathit{pure}\ g \circledast_{i=0}^{m-1} u_i) \circledast (\mathit{pure}\ f \circledast_{i=m}^{n-1} u_i) = \mathit{pure}\ (g \circ_{m,n} f) \circledast_{i=0}^{n-1} u_i \tag{3.7}$$

holds for all $0 \leq m \leq n$, $u_i :: \mathsf{A}\ \mathsf{X}$, $f :: \underbrace{\mathsf{X} \to \ldots \to \mathsf{X}}_{n-m} \to \mathsf{Y}$ and $g :: \underbrace{\mathsf{X} \to \ldots \to \mathsf{X}}_{m} \to \mathsf{Y} \to \mathsf{Z}$.

Proof. First we prove the special case $m = 0$ by induction on n. The base case is $n = 0$:

$$\begin{array}{ll}
& \mathit{pure}\ g \circledast \mathit{pure}\ f \\
= & \langle \text{ homomorphism rule (3.5) } \rangle \\
& \mathit{pure}\ (g\ f) \\
= & \langle \text{ definition of } \circ_{\cdot,\cdot} \rangle \\
& \mathit{pure}\ (g \circ_{0,0} f)
\end{array}$$

The induction step is justified like this:

$$\begin{array}{ll}
& \mathit{pure}\ g \circledast (\mathit{pure}\ f \circledast_{i=0}^{n} u_i) \\
= & \langle \text{ splitting off last application } \rangle \\
& \mathit{pure}\ g \circledast (\mathit{pure}\ f \circledast_{i=0}^{n-1} u_i \circledast u_n) \\
= & \langle \text{ composition rule (3.4) } \rangle \\
& \mathit{pure}\ (\circ) \circledast \mathit{pure}\ g \circledast (\mathit{pure}\ f \circledast_{i=0}^{n-1} u_i) \circledast u_n
\end{array}$$

$$\begin{array}{ll}
= & \langle \text{ homomorphism rule (3.5) } \rangle \\
 & \mathit{pure}\ ((\circ)\ g) \circledast (\mathit{pure}\ f \circledast_{i=0}^{n-1}\ u_i) \circledast u_n \\
= & \langle \text{ flattening formula for } 0, n\ \rangle \\
 & \mathit{pure}\ (((\circ)\ g) \circ_{0,n} f) \circledast_{i=0}^{n-1}\ u_i \circledast u_n \\
= & \langle \text{ see below } \rangle \\
 & \mathit{pure}\ (g \circ_{0,n+1} f) \circledast_{i=0}^{n-1}\ u_i \circledast u_n \\
= & \langle \text{ absorbing last application } \rangle \\
 & \mathit{pure}\ (g \circ_{0,n+1} f) \circledast_{i=0}^{n}\ u_i
\end{array}$$

The missing transformation can easily be checked by applying the expression $n+1$ times:

$$\begin{array}{ll}
 & (((\circ)\ g) \circ_{0,n} f)\ x_0 \ldots x_n \\
= & \langle \text{ definition of } \circ_{\cdot,\cdot}\ \rangle \\
 & (\circ)\ g\ (f\ x_0 \ldots x_{n-1})\ x_n \\
= & \langle \text{ definition of } (\circ)\ \rangle \\
 & g\ (f\ x_0 \ldots x_{n-1}\ x_n) \\
= & \langle \text{ definition of } \circ_{\cdot,\cdot}\ \rangle \\
 & g \circ_{0,n+1} f\ x_0 \ldots x_n
\end{array}$$

This concludes the proof of the special case $m = 0$. The general case, too, is proved by induction, this time starting with the base case $0 \leq m = n$:

$$\begin{array}{ll}
 & (\mathit{pure}\ g \circledast_{i=0}^{m-1}\ u_i) \circledast \mathit{pure}\ f \\
= & \langle \text{ interchange rule (3.6) } \rangle \\
 & \mathit{pure}\ (\lambda x \to x\ f) \circledast (\mathit{pure}\ g \circledast_{i=0}^{m-1}\ u_i) \\
= & \langle \text{ flattening formula for } 0, m\ \rangle \\
 & \mathit{pure}\ ((\lambda x \to x\ f) \circ_{0,m} g) \circledast_{i=0}^{m-1}\ u_i \\
= & \langle \text{ see below } \rangle \\
 & \mathit{pure}\ (g \circ_{m,m} f) \circledast_{i=0}^{m-1}\ u_i
\end{array}$$

The missing transformation can easily be checked by applying the expression m times:

$$\begin{array}{ll}
 & ((\lambda x \to x\ f) \circ_{0,m} g)\ x_0 \ldots x_{m-1} \\
= & \langle \text{ definition of } \circ_{\cdot,\cdot}\ \rangle \\
 & (\lambda x \to x\ f)\ (g\ x_0 \ldots x_{m-1}) \\
= & \langle \text{ beta equivalence } \rangle \\
 & g\ x_0 \ldots x_{m-1}\ f \\
= & \langle \text{ definition of } \circ_{\cdot,\cdot}\ \rangle \\
 & (g \circ_{m,m} f)\ x_0 \ldots x_{m-1}
\end{array}$$

The induction step is justified like this:

$$\begin{array}{ll}
 & (\mathit{pure}\ g \circledast_{i=0}^{m-1}\ u_i) \circledast (\mathit{pure}\ f \circledast_{i=m}^{n}\ u_i) \\
= & \langle \text{ splitting off last application } \rangle \\
 & (\mathit{pure}\ g \circledast_{i=0}^{m-1}\ u_i) \circledast (\mathit{pure}\ f \circledast_{i=m}^{n-1}\ u_i \circledast u_n)
\end{array}$$

$$
\begin{aligned}
= \quad & \langle \text{ composition rule (3.4) } \rangle \\
& \mathit{pure}\ (\circ) \circledast (\mathit{pure}\ g \circledast_{i=0}^{m-1}\ u_i) \circledast (\mathit{pure}\ f \circledast_{i=m}^{n-1}\ u_i) \circledast u_n \\
= \quad & \langle \text{ flattening formula for } 0, m \ \rangle \\
& \mathit{pure}\ ((\circ) \circ_{0,m}\ g) \circledast_{i=0}^{m-1}\ u_i \circledast (\mathit{pure}\ f \circledast_{i=m}^{n-1}\ u_i) \circledast u_n \\
= \quad & \langle \text{ flattening formula for } m, n \ \rangle \\
& \mathit{pure}\ (((\circ) \circ_{0,m}\ g) \circ_{m,n} f) \circledast_{i=0}^{n-1}\ u_i \circledast u_n \\
= \quad & \langle \text{ see below } \rangle \\
& \mathit{pure}\ (g \circ_{m,n+1} f) \circledast_{i=0}^{n-1}\ u_i \circledast u_n \\
= \quad & \langle \text{ absorbing last application } \rangle \\
& \mathit{pure}\ (g \circ_{m,n+1} f) \circledast_{i=0}^{n}\ u_i
\end{aligned}
$$

The missing transformation can easily be checked by applying the expression $n+1$ times:

$$
\begin{aligned}
& (((\circ) \circ_{0,m}\ g) \circ_{m,n} f)\ x_0 \ldots x_n \\
= \quad & \langle \text{ definition of } \circ_{\cdot,\cdot} \ \rangle \\
& ((\circ) \circ_{0,m}\ g)\ x_0 \ldots x_{m-1}\ (f\ x_m \ldots x_{n-1})\ x_n \\
= \quad & \langle \text{ definition of } \circ_{\cdot,\cdot} \ \rangle \\
& (\circ)\ (g\ x_0 \ldots x_{m-1})\ (f\ x_m \ldots x_{n-1})\ x_n \\
= \quad & \langle \text{ section } \rangle \\
& (g\ x_0 \ldots x_{m-1} \circ f\ x_m \ldots x_{n-1})\ x_n \\
= \quad & \langle \text{ definition of } (\circ) \ \rangle \\
& g\ x_0 \ldots x_{m-1}\ (f\ x_m \ldots x_{n-1}\ x_n) \\
= \quad & \langle \text{ definition of } \circ_{\cdot,\cdot} \ \rangle \\
& (g \circ_{m,n+1} f)\ x_0 \ldots x_n
\end{aligned}
$$

□

Using the flattening formula we can show the following equation about composite idioms:

$$
\begin{gathered}
\mathit{pure}\ f \circledast \mathsf{Compose}\ u_0 \circledast \ldots \circledast \mathsf{Compose}\ u_{n-1} \\
= \\
\mathsf{Compose}\ (\mathit{pure}\ (\lambda v_0 \ldots v_{n-1} \to \mathit{pure}\ f \circledast v_0 \circledast \ldots \circledast v_{n-1}) \circledast u_0 \circledast \ldots \circledast u_{n-1})
\end{gathered}
\tag{3.8}
$$

The proof is by induction, where the base case is the definition of *pure* in the Compose idiom and the induction step is shown as follows:

$$
\begin{aligned}
& \mathit{pure}\ f \circledast_{i=0}^{n} \mathsf{Compose}\ u_i \\
= \quad & \langle \text{ splitting off last } (\circledast) \ \rangle \\
& \mathit{pure}\ f \circledast_{i=0}^{n-1} \mathsf{Compose}\ u_i \circledast \mathsf{Compose}\ u_n \\
= \quad & \langle \text{ induction hypothesis } \rangle \\
& \mathsf{Compose}\ (\mathit{pure}\ (\lambda v_0 \ldots v_{n-1} \to \mathit{pure}\ f \circledast_{i=0}^{n-1}\ v_i) \circledast_{i=0}^{n-1}\ u_i) \circledast \mathsf{Compose}\ u_n \\
= \quad & \langle\ (\circledast) \text{ in } \mathsf{Compose} \text{ idiom } \rangle \\
& \mathsf{Compose}\ (\mathit{pure}\ (\circledast) \circledast (\mathit{pure}\ (\lambda v_0 \ldots v_{n-1} \to \mathit{pure}\ f \circledast_{i=0}^{n-1}\ v_i) \circledast_{i=0}^{n-1}\ u_i) \circledast u_n) \\
= \quad & \langle \text{ flattening formula (3.7) } \rangle
\end{aligned}
$$

$$\begin{array}{ll}
& \mathsf{Compose}\ ((\mathit{pure}\ ((\circledast) \circ_{0,n} (\lambda v_0 \ldots v_{n-1} \to \mathit{pure}\ f \circledast_{i=0}^{n-1} v_i)) \circledast_{i=0}^{n-1} u_i) \circledast u_n) \\
= & \langle \text{ definition of } \circ_{\cdot,\cdot} \ \rangle \\
& \mathsf{Compose}\ ((\mathit{pure}\ (\lambda v_0 \ldots v_{n-1} \to (\circledast)\ (\mathit{pure}\ f \circledast_{i=0}^{n-1} v_i)) \circledast_{i=0}^{n-1} u_i) \circledast u_n) \\
= & \langle \text{ eta equivalence } \rangle \\
& \mathsf{Compose}\ ((\mathit{pure}\ (\lambda v_0 \ldots v_{n-1}\ v_n \to (\mathit{pure}\ f \circledast_{i=0}^{n-1} v_i) \circledast v_n) \circledast_{i=0}^{n-1} u_i) \circledast u_n) \\
= & \langle \text{ absorbing last } (\circledast) \ \rangle \\
& \mathsf{Compose}\ (\mathit{pure}\ (\lambda v_0 \ldots v_{n-1}\ v_n \to \mathit{pure}\ f \circledast_{i=0}^{n} v_i) \circledast_{i=0}^{n} u_i)
\end{array}$$

We also find a useful equation for dealing with backwards idioms:

$$\begin{array}{c}
\mathit{pure}\ f \circledast \mathsf{Backwards}\ u_0 \circledast \ldots \circledast \mathsf{Backwards}\ u_{n-1} \\
= \\
\mathsf{Backwards}\ (\mathit{pure}\ (\lambda x_{n-1} \ldots x_0 \to f\ x_0 \ldots x_{n-1}) \circledast u_{n-1} \circledast \ldots \circledast u_0)
\end{array} \tag{3.9}$$

In the proof we use an additional shorthand notation for series of idiomatic applications with falling indexes. We again proceed by induction, the induction step being:

$$\begin{array}{ll}
& \mathit{pure}\ f \circledast_{i=0}^{n} \mathsf{Backwards}\ u_i \\
= & \langle \text{ splitting off last } (\circledast) \ \rangle \\
& \mathit{pure}\ f \circledast_{i=0}^{n-1} \mathsf{Backwards}\ u_i \circledast \mathsf{Backwards}\ u_n \\
= & \langle \text{ induction hypothesis } \rangle \\
& \mathsf{Backwards}\ (\mathit{pure}\ (\lambda x_{n-1} \ldots x_0 \to f\ x_0 \ldots x_{n-1}) \circledast_{i=n-1}^{0} u_i) \circledast \mathsf{Backwards}\ u_n \\
= & \langle \ (\circledast) \text{ in } \mathsf{Backwards} \text{ idiom } \rangle \\
& \mathsf{Backwards}\ (\mathit{pure}\ (\lambda x\ g \to g\ x) \circledast u_n \circledast (\mathit{pure}\ (\lambda x_{n-1} \ldots x_0 \to f\ x_0 \ldots x_{n-1}) \\
& \quad \circledast_{i=n-1}^{0} u_i)) \\
= & \langle \text{ flattening formula (3.7) } \rangle \\
& \mathsf{Backwards}\ (\mathit{pure}\ ((\lambda x\ g \to g\ x) \circ_{1,n+1} (\lambda x_{n-1} \ldots x_0 \to f\ x_0 \ldots x_{n-1})) \circledast u_n \\
& \quad \circledast_{i=n-1}^{0} u_i) \\
= & \langle \text{ definition of } \circ_{\cdot,\cdot} \ \rangle \\
& \mathsf{Backwards}\ (\mathit{pure}\ (\lambda x_n\ x_{n-1} \ldots x_0 \to (\lambda g \to g\ x_n)\ (f\ x_0 \ldots x_{n-1})) \circledast u_n \\
& \quad \circledast_{i=n-1}^{0} u_i) \\
= & \langle \text{ beta equivalence } \rangle \\
& \mathsf{Backwards}\ (\mathit{pure}\ (\lambda x_n\ x_{n-1} \ldots x_0 \to f\ x_0 \ldots x_{n-1}\ x_n) \circledast u_n \circledast_{i=n-1}^{0} u_i) \\
= & \langle \text{ absorbing first } (\circledast) \ \rangle \\
& \mathsf{Backwards}\ (\mathit{pure}\ (\lambda x_n\ x_{n-1} \ldots x_0 \to f\ x_0 \ldots x_{n-1}\ x_n) \circledast_{i=n}^{0} u_i)
\end{array}$$

3.1.5 Idiom Morphisms

In order to compare computations in different idioms, we need some way of relating them. This is done by so called *idiom morphisms*, i.e., functions $\varphi :: \mathsf{A}\ x \to \mathsf{B}\ x$ satisfying the laws

$$\varphi\ (\mathit{pure}\ x) = \mathit{pure}\ x \tag{3.10}$$

$$\varphi\ (u \circledast v) = \varphi\ u \circledast \varphi\ v \tag{3.11}$$

where A and B are idioms and the *pure* and $(\circledast)$ methods on the left (resp. right) hand side rely on the instance for A (resp. B).
An intuitive description is this: Applying an idiom morphism to an effectful computation is the same as applying the idiom morphism to every subcomputation. To get a feeling for what that actually means, let us discuss some examples, starting with the function *listToMaybe*:

$$\begin{aligned}
&\mathit{listToMaybe} :: [x] && \to \mathsf{Maybe}\ x\\
&\mathit{listToMaybe}\quad [\,] && = \mathsf{Nothing}\\
&\mathit{listToMaybe}\quad (x : \_) && = \mathsf{Just}\ x
\end{aligned}$$

This function connects nondeterminism to partiality by preserving failure and sending computations with multiple results to computations with only one result (i.e., the first one in the list). In both idioms a composite computation succeeds if every subcomputation succeeds. In the list idiom the first result of a successful composite computation stems from combining the first results of the (necessarily also successful) subcomputations. The equations (3.10) and (3.11) can be checked easily and give a formal meaning to computations in both idioms being related.
Another example is $\mathit{pure} \circ \mathit{runIdentity}$ which takes a computation with trivial effect (i.e., in the $\mathsf{Identity}$ idiom) into any idiom. The resulting computation still does not have any real effects, as it is the result of applying *pure*. This idiom morphism can be used to prove that some computation u does not have any real effects by finding a pre-image x with $u = \mathit{pure}\ (\mathit{runIdentity}\ x)$.

Lemma 3.1.2. The function

$$\mathit{pure} \circ \mathit{runIdentity} :: \mathsf{Applicative}\ a \Rightarrow \mathsf{Identity}\ x \to a\ x$$

is an idiom morphism from the $\mathsf{Identity}$ idiom to any idiom.

Proof. First we check (3.10):

$$\begin{aligned}
&\quad \mathit{pure}\ (\mathit{runIdentity}\ (\mathit{pure}\ x))\\
=&\quad \langle\ \mathit{pure} \text{ in } \mathsf{Identity} \text{ idiom}\ \rangle\\
&\quad \mathit{pure}\ (\mathit{runIdentity}\ (\mathsf{Identity}\ x))\\
=&\quad \langle \text{ inverse isomorphisms } \rangle\\
&\quad \mathit{pure}\ x
\end{aligned}$$

Since every $u :: \mathsf{Identity}\ \mathsf{X}$ is $\mathsf{Identity}\ x$ for $x = \mathit{runIdentity}\ u$ it suffices to check the following case in order to check (3.11):

$$\begin{aligned}
&\quad \mathit{pure}\ (\mathit{runIdentity}\ (\mathsf{Identity}\ f \circledast \mathsf{Identity}\ x))\\
=&\quad \langle\ (\circledast) \text{ in } \mathsf{Identity} \text{ idiom}\ \rangle\\
&\quad \mathit{pure}\ (\mathit{runIdentity}\ (\mathsf{Identity}\ (f\ x)))\\
=&\quad \langle \text{ inverse isomorphisms } \rangle\\
&\quad \mathit{pure}\ (f\ x)
\end{aligned}$$

$$\begin{aligned}
&= \quad \langle \text{ homomorphism law (3.5) } \rangle \\
&\quad \mathit{pure}\ f \circledast \mathit{pure}\ x
\end{aligned}$$

□

In section 3.1.2 we have claimed composition of idioms to be associative in some sense. Of course none of the equations

$$\begin{aligned}
\mathsf{Compose\ A\ Identity} &\cong \mathsf{A} \\
\mathsf{Compose\ Identity\ A} &\cong \mathsf{A} \\
\mathsf{Compose\ A\ (Compose\ B\ C)} &\cong \mathsf{Compose\ (Compose\ A\ B)\ C}
\end{aligned}$$

holds as equations of types. They are true as isomorphisms of idioms though: There is a pair of mutually inverse idiom morphisms connecting both sides. We exemplarily show the first one:

Lemma 3.1.3. The function

$$\mathit{liftA}\ \mathit{runIdentity} \circ \mathit{runCompose} :: \mathsf{Applicative}\ a \Rightarrow \mathsf{Compose}\ a\ \mathsf{Identity}\ x \to a\ x$$

is an invertible idiom morphism from $\mathsf{Compose\ A\ Identity}$ to A for any idiom A.

Proof. Invertibility is easily checked, where the inverse is $\mathsf{Compose} \circ \mathit{liftA}\ \mathsf{Identity}$. On to checking (3.10):

$$\begin{aligned}
&\quad \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{runCompose}\ (\mathit{pure}\ x)) \\
&= \quad \langle\ \mathit{pure} \text{ in } \mathsf{Compose} \text{ idiom } \rangle \\
&\quad \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{runCompose}\ (\mathsf{Compose}\ (\mathit{pure}\ (\mathit{pure}\ x)))) \\
&= \quad \langle \text{ inverse isomorphisms } \rangle \\
&\quad \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{pure}\ (\mathit{pure}\ x)) \\
&= \quad \langle\ \mathit{pure} \text{ in } \mathsf{Identity} \text{ idiom } \rangle \\
&\quad \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{pure}\ (\mathsf{Identity}\ x)) \\
&= \quad \langle \text{ definition of } \mathit{liftA}\ \rangle \\
&\quad \mathit{pure}\ \mathit{runIdentity} \circledast \mathit{pure}\ (\mathsf{Identity}\ x) \\
&= \quad \langle \text{ homomorphism law (3.5) } \rangle \\
&\quad \mathit{pure}\ (\mathit{runIdentity}\ (\mathsf{Identity}\ x)) \\
&= \quad \langle \text{ inverse isomorphisms } \rangle \\
&\quad \mathit{pure}\ x
\end{aligned}$$

The proof for (3.11) relies on the flattening formula:

$$\begin{aligned}
&\quad \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{runCompose}\ (\mathsf{Compose}\ u \circledast \mathsf{Compose}\ v)) \\
&= \quad \langle\ (\circledast) \text{ in } \mathsf{Compose} \text{ idiom } \rangle \\
&\quad \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{runCompose}\ (\mathsf{Compose}\ (\mathit{pure}\ (\circledast) \circledast u \circledast v))) \\
&= \quad \langle \text{ inverse isomorphisms } \rangle
\end{aligned}$$

$$\begin{array}{ll}
& \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{pure}\ (\circledast) \circledast u \circledast v) \\
= & \langle \text{ definition of } \mathit{liftA} \ \rangle \\
& \mathit{pure}\ \mathit{runIdentity} \circledast (\mathit{pure}\ (\circledast) \circledast u \circledast v) \\
= & \langle \text{ eta equivalence } \rangle \\
& \mathit{pure}\ \mathit{runIdentity} \circledast (\mathit{pure}\ (\lambda x\ y \to x \circledast y) \circledast u \circledast v) \\
= & \langle \text{ flattening formula (3.7) } \rangle \\
& \mathit{pure}\ (\lambda x\ y \to \mathit{runIdentity}\ (x \circledast y)) \circledast u \circledast v \\
= & \langle\ (\circledast) \text{ in } \mathsf{Identity} \text{ idiom } \rangle \\
& \mathit{pure}\ (\lambda x\ y \to \mathit{runIdentity}\ x\ (\mathit{runIdentity}\ y)) \circledast u \circledast v \\
= & \langle \text{ flattening formula (3.7) } \rangle \\
& \mathit{pure}\ (\lambda x \to \mathit{runIdentity}\ x) \circledast u \circledast (\mathit{pure}\ (\lambda y \to \mathit{runIdentity}\ y) \circledast v) \\
= & \langle \text{ eta equivalence } \rangle \\
& \mathit{pure}\ \mathit{runIdentity} \circledast u \circledast (\mathit{pure}\ \mathit{runIdentity} \circledast v) \\
= & \langle \text{ definition of } \mathit{liftA} \ \rangle \\
& \mathit{liftA}\ \mathit{runIdentity}\ u \circledast \mathit{liftA}\ \mathit{runIdentity}\ v \\
= & \langle \text{ inverse isomorphisms } \rangle \\
& \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{runCompose}\ (\mathsf{Compose}\ u)) \\
& \quad \circledast\ \mathit{liftA}\ \mathit{runIdentity}\ (\mathit{runCompose}\ (\mathsf{Compose}\ v))
\end{array}$$

We also have to check that the inverse is an idiom morphism. Here, we can use the corresponding properties for the direction we have already established.

$$\begin{array}{ll}
& \mathsf{Compose}\ (\mathit{liftA}\ \mathsf{Identity}\ (\mathit{pure}\ x)) \\
= & \langle \text{ see above } \rangle \\
& \mathsf{Compose}\ (\mathit{liftA}\ \mathsf{Identity}\ (\mathit{liftA}\ \mathit{runIdentity}\ (\mathit{runCompose}\ (\mathit{pure}\ x)))) \\
= & \langle \text{ inverse isomorphisms } \rangle \\
& \mathit{pure}\ x
\end{array}$$

The other derivation is analogous. □

Finally, it is worth pointing out that an idiom morphism between two monads does not necessarily have to be a monad morphism. Here, a monad morphism is defined in much the same way as an idiom morphism as some function $\varphi :: \mathsf{M}\ x \to \mathsf{M}'\ x$ satisfying two rules:

$$\begin{aligned}
\varphi\ (\mathit{return}\ x) &= \mathit{return}\ x \\
\varphi\ (u \mathbin{>\!\!>\!\!=} v) &= \varphi\ u \mathbin{>\!\!>\!\!=} \varphi \circ v
\end{aligned}$$

The list type constructor and the Maybe type constructor are monads and thus also idioms. The function *listToMaybe* is an idiom morphism but it is not a monad morphism, as the following counter-example shows.

$$\begin{array}{ll}
& [0,1] \mathbin{>\!\!>\!\!=} \lambda n \to [1\,..\,n] \\
= & \langle\ (\mathbin{>\!\!>\!\!=}) \text{ in list monad } \rangle \\
& [1\,..\,0] \mathbin{+\!\!+} [1\,..\,1]
\end{array}$$

$$\begin{array}{ll}
= & \langle \text{ calculating the arithmetic sequences } \rangle \\
 & [\,] +\!\!+ [1] \\
= & \langle \text{ list concatenation } \rangle \\
 & [1]
\end{array}$$

So we have $\mathit{listToMaybe}\ ([0,1] \gg\!= \lambda n \rightarrow [1\mathinner{..}n]) = \mathit{listToMaybe}\ [1] = \mathsf{Just}\ 1$. On the other hand:

$$\begin{array}{ll}
 & \mathit{listToMaybe}\ [0,1] \gg\!= \lambda n \rightarrow \mathit{listToMaybe}\ [1\mathinner{..}n] \\
= & \langle \text{ definition of } \mathit{listToMaybe} \ \rangle \\
 & \mathsf{Just}\ 0 \gg\!= \lambda n \rightarrow \mathit{listToMaybe}\ [1\mathinner{..}n] \\
= & \langle \ (\gg\!=) \text{ in } \mathsf{Maybe} \text{ monad } \rangle \\
 & \mathit{listToMaybe}\ [1\mathinner{..}0] \\
= & \langle \text{ calculating the arithmetic sequence } \rangle \\
 & \mathit{listToMaybe}\ [\,] \\
= & \langle \text{ definition of } \mathit{listToMaybe} \ \rangle \\
 & \mathsf{Nothing}
\end{array}$$

3.2 Traversable Functors

3.2.1 The Interface

Now we turn to the actual protagonists of this chapter – traversable functors. In the last section we have written *label* and *collect* in applicative syntax as:

$$\begin{array}{llll}
\mathit{label} & :: \mathsf{Tree}\ x & \rightarrow & \mathsf{State\ Int}\ (\mathsf{Tree\ Int}) \\
\mathit{label} & (\mathsf{Leaf}\ x) & = & \mathit{pure}\ \mathsf{Leaf} \circledast \mathit{fresh} \\
\mathit{label} & (\mathsf{Node}\ l\ r) & = & \mathit{pure}\ \mathsf{Node} \circledast \mathit{label}\ l \circledast \mathit{label}\ r \\
\mathit{collect} & :: \mathsf{Tree}\ x & \rightarrow & \mathsf{Const}\ [x]\ (\mathsf{Tree}\ y) \\
\mathit{collect} & (\mathsf{Leaf}\ x) & = & \mathit{pure}\ \mathsf{Leaf} \circledast \mathit{note}\ x \\
\mathit{collect} & (\mathsf{Node}\ l\ r) & = & \mathit{pure}\ \mathsf{Node} \circledast \mathit{collect}\ l \circledast \mathit{collect}\ r
\end{array}$$

Both functions are quite similar, in particular the recursion scheme coincides. The only part that actually differs is *fresh* vs. $\mathit{note}\ x$. To compensate for the missing dependence of *fresh* on x, we can replace *fresh* by the equivalent $\mathit{const}\ \mathit{fresh}\ x$. We can then introduce a common abstraction of *label* and *collect*:

$$\begin{array}{l}
\mathit{traverseTree} :: \mathsf{Applicative}\ a \Rightarrow (x \rightarrow a\ y) \rightarrow \mathsf{Tree}\ x \rightarrow a\ (\mathsf{Tree}\ y) \\
\mathit{traverseTree}\ u\ (\mathsf{Leaf}\ x) \quad\ = \mathit{pure}\ \mathsf{Leaf} \circledast u\ x \\
\mathit{traverseTree}\ u\ (\mathsf{Node}\ l\ r) = \mathit{pure}\ \mathsf{Node} \circledast \mathit{traverseTree}\ u\ l \circledast \mathit{traverseTree}\ u\ r \\
\mathit{label} :: \ \mathsf{Tree}\ x \rightarrow \mathsf{State\ Int}\ (\mathsf{Tree\ Int}) \\
\mathit{label} = \mathit{traverseTree}\ (\mathit{const}\ \mathit{fresh}) \\
\mathit{collect} :: \ \mathsf{Tree}\ x \rightarrow \mathsf{Const}\ [x]\ (\mathsf{Tree}\ y) \\
\mathit{collect} = \mathit{traverseTree}\ \mathit{note}
\end{array}$$

```
class Functor t where
  fmap :: (x → y) → t x → t y
class Foldable t where
  foldMap :: Monoid n ⇒ (x → n) → t x → n
class (Functor t, Foldable t) ⇒ Traversable t where
  traverse :: Applicative a ⇒ (x → a y) → t x → a (t y)
  -- with fmap = fmapDefault a suitable Functor instance can be defined
fmapDefault :: Traversable t ⇒ (x → y) → t x → t y
fmapDefault f = runIdentity ∘ traverse (Identity ∘ f)
  -- with foldMap = foldMapDefault a suitable Foldable instance can be defined
foldMapDefault :: (Traversable t, Monoid n) ⇒ (x → n) → t x → n
foldMapDefault f = getConst ∘ traverse (Const ∘ f)
```

Figure 3.2: The Traversable class and its superclasses Functor and Foldable

Specializing the argument *u* of *traverseTree* to *const fresh* gives the former implementation of *label* and specializing *u* to *note* gives the former implementation of *collect*. Using the common abstraction *traverseTree* saves us from spelling out the pattern matching twice and makes both *label* and *collect* very short.

We can give a similar function for lists instead of trees

```
traverseList :: Applicative a ⇒ (x → a y) → [x] → a [y]
traverseList u []       = pure []
traverseList u (x : xs) = pure (:) ⊛ u x ⊛ traverseList u xs
```

and similar functions exist for many other types. This motivates the definition of the typeclass Traversable [McBride and Paterson, 2008]. The class declaration together with declarations for the superclasses Functor and Foldable can be found in figure 3.2. All three classes are defined in the Prelude[9].
The *traverseTree* function given for the Tree type above now becomes part of a class instance:

```
instance Traversable Tree where
  traverse u (Leaf x)   = pure Leaf ⊛ u x
  traverse u (Node l r) = pure Node ⊛ traverse u l ⊛ traverse u r
```

And similarly for lists:

[9] Functor: `http://hackage.haskell.org/package/base/docs/Prelude.html#t:Functor`
Foldable: `http://hackage.haskell.org/package/base/docs/Prelude.html#t:Foldable`
Traversable: `http://hackage.haskell.org/package/base/docs/Prelude.html#t:Traversable`

```
instance Traversable [] where
  traverse u []       = pure []
  traverse u (x : xs) = pure (:) ⊛ u x ⊛ traverse u xs
```

In general every tree-like data type can be made an instance by visiting the fields of a constructor from left to right, either using *traverse* recursively or applying u, and combining the results using the same constructor again. This is in fact what the automatically derived instances look like. There are however other reasonable instances as we will see.

The superclass Functor provides a method *fmap*, which is similar to *traverse* but can only deal with pure functions. As we can interpret pure functions as effectful functions with a trivial effect, *fmap* should be uniquely determined by *traverse*. This in fact holds and the superclass constraint Functor t can always be satisfied by using

```
instance Functor T where
  fmap = fmapDefault
```

as a Functor instance. So the Functor instance relies on the Traversable instance, while at the same time T being a functor is necessary for having a Traversable instance in the first place. This seemingly circular dependence is fine however, as *fmap* relies on *traverse* on the value level and Traversable relies on Functor on the type level. Later we will show that *fmapDefault* always coincides with *fmap* for any lawful Traversable instance.

The Foldable class allows to combine all entries of a container if they can be mapped into a monoid somehow. This can be used to define the *contents* function, but also to get the number of entries of a container or the minimal entry (if an ordering is defined). Note that *foldMapDefault* relies on the Const applicative, which is one of the reasons for having applicative effects instead of monadic ones.

The interdependence between Foldable and Traversable is much the same as between Functor and Traversable: The *foldMap* method is uniquely determined by *traverse* and can be defined using *foldMapDefault*.

There is also a connection between Functor and Foldable. Whenever a type constructor is an instance of both classes, the equation

$$\mathit{foldMap}\ g \circ \mathit{fmap}\ f = \mathit{foldMap}\ (g \circ f)$$

is required to hold. This is true for *foldMapDefault* and *fmapDefault* as we will see.

3.2.2 Expected Behavior

Figure 3.3 shows the definitions by Hutton and Fulger [2008], but generalized in a straightforward manner to work with arbitrary traversals. For the Tree type and the *traverseTree* function, Hutton and Fulger [2008] show in their Theorem 1 that

$$\mathit{contents}\ (\mathit{relabel}\ t)$$

```
$\mathit{fresh} :: \mathsf{State\ Int\ Int}$
$\mathit{fresh} = \mathbf{do}$
    $n \leftarrow \mathit{get}$
    $\mathit{put}\ (n+1)$
    $\mathit{return}\ n$
$\mathit{label} :: \mathsf{Traversable}\ t \Rightarrow t\ x \rightarrow \mathsf{State\ Int}\ (t\ \mathsf{Int})$
$\mathit{label} = \mathit{traverse}\ (\mathit{const}\ \mathit{fresh})$
$\mathit{relabel} :: \mathsf{Traversable}\ t \Rightarrow t\ x \rightarrow t\ \mathsf{Int}$
$\mathit{relabel}\ t = \mathit{evalState}\ (\mathit{label}\ t)\ 0$
$\mathit{note} :: x \rightarrow \mathsf{Const}\ [x]\ y$
$\mathit{note}\ x = \mathsf{Const}\ [x]$
$\mathit{collect} :: \mathsf{Traversable}\ t \Rightarrow t\ x \rightarrow \mathsf{Const}\ [x]\ (t\ y)$
$\mathit{collect} = \mathit{traverse}\ \mathit{note}$
$\mathit{contents} :: \mathsf{Traversable}\ t \Rightarrow t\ x \rightarrow [x]$
$\mathit{contents}\ t = \mathit{getConst}\ (\mathit{collect}\ t)$
```

Figure 3.3: Code base for the generalized labeling example

does not contain any duplicates. In fact they show that

$$\mathit{runState}\ (\mathit{label}\ t)\ n = (t', n')$$

implies

$$\mathit{contents}\ t' = [n \mathinner{..} (n' - 1)]$$

(which is their lemma 2). The latter claim is stronger, in that it also requires all labels to be used, i.e., no label is wasted. The proof is done by checking the property by using the definitions and doing induction over recursive calls.
Now we ask exactly the same question in the more general context of an arbitrary traversal: Is it true that *contents* (*relabel* *t*) does not contain any duplicates?

The answer is no, as there is the following counter-example. We define a wrapper Twice with a ‘bad’ Traversable instance.[10]

```
$\mathbf{newtype}\ \mathsf{Twice}\ x = \mathsf{Twice}\ x$
$\mathbf{instance}\ \mathsf{Traversable\ Twice}\ \mathbf{where}$
    $\mathit{traverse}\ u\ (\mathsf{Twice}\ x) = \mathit{pure}\ (\lambda\_\ y \rightarrow \mathsf{Twice}\ y) \circledast u\ x \circledast u\ x$
```

Now all we have to do is calculate.

[10] This is a reformulation of the flawed traversal Jaskelioff and Rypacek [2012] used to exemplify how the composition law (3.13) rules out duplicating traversals.

$$
\begin{array}{ll}
& \mathit{label}\ (\mathsf{Twice}\ x) \\
= & \langle \text{ definition of } \mathit{label} \rangle \\
& \mathit{traverse}\ (\mathit{const\ fresh})\ (\mathsf{Twice}\ x) \\
= & \langle \text{ definition of } \mathit{traverse} \rangle \\
& \mathit{pure}\ (\lambda\_\ y \to \mathsf{Twice}\ y) \circledast \mathit{const\ fresh}\ x \circledast \mathit{const\ fresh}\ x \\
= & \langle \text{ definition of } \mathit{const} \rangle \\
& \mathit{pure}\ (\lambda\_\ y \to \mathsf{Twice}\ y) \circledast \mathit{fresh} \circledast \mathit{fresh} \\
= & \langle \text{ translating into a } \textbf{do}\text{-block} \rangle \\
& \textbf{do}\ \{\_ \leftarrow \mathit{fresh}; y \leftarrow \mathit{fresh}; \mathit{return}\ (\mathsf{Twice}\ y)\}
\end{array}
$$

Running this stateful computation with the initial value 0 will result in Twice 1, as the first call of *fresh* gives 0 and the second call gives 1. So we have

$$\mathit{relabel}\ (\mathsf{Twice}\ x) = \mathsf{Twice}\ 1.$$

On to collecting labels:

$$
\begin{array}{ll}
& \mathit{collect}\ (\mathsf{Twice}\ 1) \\
= & \langle \text{ definition of } \mathit{collect} \rangle \\
& \mathit{traverse\ note}\ (\mathsf{Twice}\ 1) \\
= & \langle \text{ definition of } \mathit{traverse} \rangle \\
& \mathit{pure}\ (\lambda\_\ y \to \mathsf{Twice}\ y) \circledast \mathit{note}\ 1 \circledast \mathit{note}\ 1 \\
= & \langle \text{ definition of } \mathit{note} \rangle \\
& \mathit{pure}\ (\lambda\_\ y \to \mathsf{Twice}\ y) \circledast \mathsf{Const}\ [1] \circledast \mathsf{Const}\ [1] \\
= & \langle \text{ calculation in the } \mathsf{Const} \text{ idiom } \rangle \\
& \mathsf{Const}\ [1,1]
\end{array}
$$

So *contents* (Twice 1) = *getConst* (Const $[1,1]$) = $[1,1]$ which clearly does contain a duplicate entry.

What happened? The given *traverse* function visits the single entry inside the Twice wrapper twice, even though only one number can be stored. Then the single entry is written into the *contents* list twice.

This is not the behaviour we expect when using *traverse*. On an intuitive level it is quite clear that we want to rule out such duplicitous traversals. In order to do so, we need to formulate laws for the Traversable class. All lawful instances will make (the generalization of) Hutton and Fulger's claim true.

Figure 3.4 shows some flawed traversal functions for the Tree type constructor. The *notMuchLeft* function ignores the left subtree altogether, so most of the contents will be lost after traversing. In *mixedUp* both subtrees are used, but they switch places, which will change the shape of the tree. Finally, *duplicitous* visits the left subtree twice but only uses the result of the first visit. This preserves the shape, but triggers unnecessary (and unwanted) effects.

In contrast, the following is a valid traversal and actually useful, as it allows to traverse a tree in breadth first order[11]:

[11] This function is not structurally recursive, but can still be shown to be total.

```
notMuchLeft :: Applicative a ⇒ (x → a y) → Tree x → a (Tree y)
notMuchLeft u (Leaf x)    = pure Leaf ⊛ u x
notMuchLeft u (Node l r) = notMuchLeft u r

mixedUp :: Applicative a ⇒ (x → a y) → Tree x → a (Tree y)
mixedUp u (Leaf x)    = pure Leaf ⊛ u x
mixedUp u (Node l r) = pure Node ⊛ mixedUp u r ⊛ mixedUp u l

duplicitous :: Applicative a ⇒ (x → a y) → Tree x → a (Tree y)
duplicitous u (Leaf x)      = pure Leaf ⊛ u x
duplicitous u (Node l r)   = pure (λl′ _ r′ → Node l′ r′)
  ⊛ duplicitous u l ⊛ duplicitous u l ⊛ duplicitous u r
```

Figure 3.4: Unlawful traversal functions

```
bftraverse :: Applicative a ⇒ (x → a y) → Tree x → a (Tree y)
bftraverse u t = pure head ⊛ go [t] where
  go []                 = pure []
  go (Leaf x    : ts) = pure (λy ts′          → ts′ ++ [Leaf y]     ) ⊛ u x ⊛ go ts
  go (Node l r : ts) = pure (λ(r′ : l′ : ts′) → ts′ ++ [Node l′ r′]) ⊛ go (ts ++ [l, r])
```

The function *go* takes a list of trees, traverses all of them in parallel and collects the resulting trees in reversed order. Nodes are handled by enqueuing both subtrees, calling *go* recursively and extracting the subtrees from the front of the (now reversed) list. Using *head* is safe in the above, as the result of *go* always has the same length as the argument *go* is called with.
This example shows that correct instances are not unique in general; often there are different reasonable choices.

3.2.3 Laws for the Traversable Class

Before we give the laws for the Traversable class, we quickly review the laws concerning the superclasses Functor and Foldable. We expect *fmap* to satisfy the two laws

$$\begin{aligned} \mathit{fmap}\ \mathit{id} &= \mathit{id} \\ \mathit{fmap}\ (g \circ f) &= \mathit{fmap}\ g \circ \mathit{fmap}\ f \end{aligned}$$

where $f :: \mathsf{X} \to \mathsf{Y}$ and $g :: \mathsf{Y} \to \mathsf{Z}$. If a type constructor is an instance of both Functor and Foldable, we additionally require

$$\mathit{foldMap}\ (g \circ f) = \mathit{foldMap}\ g \circ \mathit{fmap}\ f$$

where $f :: \mathsf{X} \to \mathsf{Y}$ and $g :: \mathsf{Y} \to \mathsf{N}$ and N is an instance of Monoid.
Jaskelioff and Rypacek [2012] proposed the following set of laws for the Traversable class. All the individual laws were published in [Gibbons and Oliveira, 2009] already.

The *identity law* states that

$$\mathit{traverse}\ \mathsf{Identity} = \mathsf{Identity} \tag{3.12}$$

where $\mathsf{Identity}$ is the wrapper of the dummy idiom $\mathsf{Identity}$ we have seen in section 3.1.2. In [Jaskelioff and Rypacek, 2012] this law is called *unitarity law*.
Idioms can be composed as we have also seen in section 3.1.2 and there is a law to make sure idiom composition interacts well with composing (effectful) functions. Given computations $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$ and $v :: \mathsf{Y} \to \mathsf{B}\ \mathsf{Z}$ we require the *composition law*

$$\mathit{traverse}\ (\mathsf{Compose} \circ \mathit{liftA}\ v \circ u) = \mathsf{Compose} \circ \mathit{liftA}\ (\mathit{traverse}\ v) \circ \mathit{traverse}\ u \tag{3.13}$$

to hold. In [Jaskelioff and Rypacek, 2012] this law is called *linearity law*.
Also we require some naturality properties, the first of which is compatibility with idiom morphisms. If $\varphi :: \mathsf{A}\ x \to \mathsf{B}\ x$ is an idiom morphism and $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$ an effectful function,

$$\mathit{traverse}\ (\varphi \circ u) = \varphi \circ \mathit{traverse}\ u \tag{3.14}$$

has to hold.
Regarding a Functor instance as given we have to make sure the Traversable instance is correct with respect to this Functor instance. In this case we expect the following naturality properties for all applicatives A and B and all functions $f :: \mathsf{X} \to \mathsf{Y}$, $g :: \mathsf{Y} \to \mathsf{Z}$, $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$ and $v :: \mathsf{Y} \to \mathsf{B}\ \mathsf{Z}$:

$$\mathit{traverse}\ (v \circ f) = \mathit{traverse}\ v \circ \mathit{fmap}\ f \tag{3.15}$$

$$\mathit{traverse}\ (\mathit{liftA}\ g \circ u) = \mathit{liftA}\ (\mathit{fmap}\ g) \circ \mathit{traverse}\ u \tag{3.16}$$

These laws imply that *fmap* coincides with *fmapDefault*, as we will see, so we do not need to require any further laws connecting Traversable and Functor. Since none of the laws mention *foldMap*, we ensure compatibility with Foldable by requiring *foldMap* to coincide with *foldMapDefault*.

The rules do indeed detect flawed instances. The identity law (3.12) does not hold for *notMuchLeft* and *mixedUp* from figure 3.4. To show this, we define

$$t_0 = \mathsf{Node}\ (\mathsf{Leaf}\ 0)\ (\mathsf{Leaf}\ 1)$$

and compute

$$\begin{aligned} \mathit{notMuchLeft}\ \mathsf{Identity}\ t_0 &= \mathsf{Identity}\ (\mathsf{Leaf}\ 1) \\ \mathit{mixedUp}\ \mathsf{Identity}\ t_0 &= \mathsf{Identity}\ (\mathsf{Node}\ (\mathsf{Leaf}\ 1)\ (\mathsf{Leaf}\ 0)) \end{aligned}$$

so neither gives the correct (according to (3.12)) result $\mathsf{Identity}\ t_0$.
The flaw in the *duplicitous* function is harder to detect because the identity law (3.12) is ignorant of the additional effects. The composition law (3.13) however does detect the flawed instance:

$$\begin{aligned} & \mathit{duplicitous}\ (\lambda\_ \to \mathsf{Compose}\ [[\,], [0]])\ t_0 \\ = & \\ & \mathsf{Compose}\ [[\,], [\,], [\,], [\,], [\,], [\,], [\,], [\mathsf{Node}\ (\mathsf{Leaf}\ 0)\ (\mathsf{Leaf}\ 0)]] \end{aligned}$$

This is not the same as:

$$\begin{aligned} & (\mathsf{Compose} \circ \mathit{liftA}\ (\mathit{duplicitous}\ \mathit{id}) \circ \mathit{duplicitous}\ (\lambda\_ \to [[\,], [0]]))\ t_0 \\ = & \\ & \mathsf{Compose}\ [[\,], [\,], [\,], [\,], [\,], [\mathsf{Node}\ (\mathsf{Leaf}\ 0)\ (\mathsf{Leaf}\ 0)], [\,], [\mathsf{Node}\ (\mathsf{Leaf}\ 0)\ (\mathsf{Leaf}\ 0)]] \end{aligned}$$

On the other hand, the composition law does not rule out *notMuchLeft*. Because the alteration of the shape is idempotent, applying it twice gives the same result as applying it once.

Note that the composition law (3.13) can only be formulated because of applicative effects instead of monadic effects. If we defined the Traversable class to only work with monadic effects, the composition of two monads would not be an allowed type of effect for traversing anymore.
We might try to use a monadic composition law instead. For example using Kleisli composition

$$\begin{aligned} & (\mathrel{>\!\!=\!\!\!>}) :: \mathsf{Monad}\ m \Rightarrow (x \to m\ y) \to (y \to m\ z) \to x \to m\ z \\ & (u \mathrel{>\!\!=\!\!\!>} v)\ x = u\ x \mathbin{>\!\!\!>\!\!=} v \end{aligned}$$

we could require

$$\mathit{traverse}\ u \mathrel{>\!\!=\!\!\!>} \mathit{traverse}\ v = \mathit{traverse}\ (u \mathrel{>\!\!=\!\!\!>} v) \tag{3.17}$$

to hold. This equation is however wrong for most instances we consider appropriate: The left side applies u to all entries first and only then starts applying v, while the right side alternates between u and v in one traversal. Gibbons and Oliveira [2009] show that for commutative monads equation (3.17) is a consequence of the laws. We will return to this issue when we discuss applications of the representation theorem in 3.6.
The above considerations are further examples for Applicative and Traversable being a good match. Monadic effects may constitute the majority of traversals that actually appear in real code, but they should still be considered a special case of idiomatic traversals. As soon as we start using truly monadic features (like Kleisli composition), things behave strangely.

The set of laws proposed by Jaskelioff and Rypacek [2012] seems natural from a category theory point of view, but it is not at all clear how to put these laws to use. The properties we expect on an intuitive level, like ‘no duplicitous traversals’, are not immediate from the laws. Thus when it comes to arguing about given code, the artificial nature of the laws makes it hard to gain any insights. The next subsection contains some easy and still very general consequences and the next section shows how the (generalized) claim about labeling and collecting labels can be shown. The representation theorem 3.5.2 will give an equivalent characterization of what constitutes a lawful instance, that is ready to be used for arguing about given code.

3.2.4 Some Consequences of the Laws

To get a first feeling for how to work with the laws just presented, we show some easy consequences of the laws. From now on we require all traversals to be lawful and take "traversable" to actually mean "lawfully traversable". The first consequence is that *fmapDefault* defined via *traverse* coincides with *fmap*.

$$
\begin{array}{ll}
 & \mathit{fmapDefault}\ f \\
= & \langle \text{ definition of } \mathit{fmapDefault} \ \rangle \\
 & \mathit{runIdentity} \circ \mathit{traverse}\ (\mathsf{Identity} \circ f) \\
= & \langle \text{ naturality law (3.15) } \rangle \\
 & \mathit{runIdentity} \circ \mathit{traverse}\ \mathsf{Identity} \circ \mathit{fmap}\ f \\
= & \langle \text{ identity law (3.12) } \rangle \\
 & \mathit{runIdentity} \circ \mathsf{Identity} \circ \mathit{fmap}\ f \\
= & \langle \text{ inverse isomorphisms } \rangle \\
 & \mathit{fmap}\ f
\end{array}
$$

Note that we did not use the naturality law (3.16) here. In fact, it is unnecessary to require this law at all as it is a consequence of the other laws:

$$
\begin{array}{ll}
 & \mathit{liftA}\ (\mathit{fmap}\ g) \circ \mathit{traverse}\ u \\
= & \langle \ \mathit{fmap} = \mathit{fmapDefault} \ \rangle \\
 & \mathit{liftA}\ (\mathit{fmapDefault}\ g) \circ \mathit{traverse}\ u \\
= & \langle \text{ definition of } \mathit{fmapDefault} \ \rangle \\
 & \mathit{liftA}\ (\mathit{runIdentity} \circ \mathit{traverse}\ (\mathsf{Identity} \circ g)) \circ \mathit{traverse}\ u \\
= & \langle \ \mathit{liftA} \text{ and function composition } \rangle \\
 & \mathit{liftA}\ \mathit{runIdentity} \circ \mathit{liftA}\ (\mathit{traverse}\ (\mathsf{Identity} \circ g)) \circ \mathit{traverse}\ u \\
= & \langle \text{ inverse isomorphisms } \rangle \\
 & \mathit{liftA}\ \mathit{runIdentity} \circ \mathit{runCompose} \circ \mathsf{Compose} \circ \mathit{liftA}\ (\mathit{traverse}\ (\mathsf{Identity} \circ g)) \\
 & \quad \circ\ \mathit{traverse}\ u \\
= & \langle \text{ composition law (3.13) } \rangle \\
 & \mathit{liftA}\ \mathit{runIdentity} \circ \mathit{runCompose} \circ \mathit{traverse}\ (\mathsf{Compose} \circ \mathit{liftA}\ (\mathsf{Identity} \circ g) \circ u) \\
= & \langle \text{ idiom morphism law (3.14) and lemma 3.1.3 } \rangle \\
 & \mathit{traverse}\ (\mathit{liftA}\ \mathit{runIdentity} \circ \mathit{runCompose} \circ \mathsf{Compose} \circ \mathit{liftA}\ (\mathsf{Identity} \circ g) \circ u) \\
= & \langle \text{ inverse isomorphisms } \rangle \\
 & \mathit{traverse}\ (\mathit{liftA}\ \mathit{runIdentity} \circ \mathit{liftA}\ (\mathsf{Identity} \circ g) \circ u) \\
= & \langle \ \mathit{liftA} \text{ and function composition } \rangle \\
 & \mathit{traverse}\ (\mathit{liftA}\ (\mathit{runIdentity} \circ \mathsf{Identity} \circ g) \circ u) \\
= & \langle \text{ inverse isomorphisms } \rangle \\
 & \mathit{traverse}\ (\mathit{liftA}\ g \circ u)
\end{array}
$$

A similar proof shows that the other naturality property

$$\mathit{traverse}\ (v \circ f) = \mathit{traverse}\ v \circ \mathit{fmap}\ f \tag{3.15}$$

also follows from *fmap* = *fmapDefault*, the composition law (3.13) and naturality in the idiom (3.14). Thus we could have required *fmap* = *fmapDefault*, identity (3.12), composition (3.13) and the idiom morphism law (3.14) as an alternative set of laws (this is what the GHC Prelude does). Going one step further we could even decide to forget about Functor altogether and use *fmapDefault* right away. In this case the first functor law follows from identity (3.12) and the second functor law follows from composition (3.13) and naturality in the idiom (3.14).

The compatibility requirement between Functor and Foldable is satisfied if both are compatible with the Traversable instance:

$$\begin{aligned}
&\quad \mathit{foldMap}\ g \circ \mathit{fmap}\ f\\
=&\quad \langle\ \mathit{foldMap} \text{ in terms of } \mathit{traverse}\ \rangle\\
&\quad \mathit{getConst} \circ \mathit{traverse}\ (\mathsf{Const} \circ g) \circ \mathit{fmap}\ f\\
=&\quad \langle\ \text{naturality law (3.15)}\ \rangle\\
&\quad \mathit{getConst} \circ \mathit{traverse}\ (\mathsf{Const} \circ g \circ f)\\
=&\quad \langle\ \mathit{foldMap} \text{ in terms of } \mathit{traverse}\ \rangle\\
&\quad \mathit{foldMap}\ (g \circ f)
\end{aligned}$$

Another easy consequence is sometimes referred to as the *purity law* and states that

$$\mathit{traverse}\ \mathit{pure} = \mathit{pure}.$$

Indeed:

$$\begin{aligned}
&\quad \mathit{traverse}\ \mathit{pure}\\
=&\quad \langle\ \text{inverse isomorphisms}\ \rangle\\
&\quad \mathit{traverse}\ (\mathit{pure} \circ \mathit{runIdentity} \circ \mathsf{Identity})\\
=&\quad \langle\ \text{idiom morphism law (3.14) and lemma 3.1.2}\ \rangle\\
&\quad \mathit{pure} \circ \mathit{runIdentity} \circ \mathit{traverse}\ \mathsf{Identity}\\
=&\quad \langle\ \text{identity law (3.12)}\ \rangle\\
&\quad \mathit{pure} \circ \mathit{runIdentity} \circ \mathsf{Identity}\\
=&\quad \langle\ \text{inverse isomorphisms}\ \rangle\\
&\quad \mathit{pure}
\end{aligned}$$

3.3 Proving the Labeling Claim only using the Laws

In this section we show how Hutton and Fulger's relabeling statement can be proved for all Traversable types. The final forms of the relevant functions have been given in figure 3.3 already.

The claim can now be formulated in various ways. One would be *contents* (*relabel t*) not having any duplicate entries. Like Hutton and Fulger we not only claim all list entries to be different, but that the list in fact contains subsequent numbers (up to, but excluding the next label):

$$\mathit{runState}\ (\mathit{label}\ t)\ n = (t', n') \implies \mathit{contents}\ t' = [n \mathinner{..} n' - 1] \tag{3.18}$$

The key to the direct proof is the following lemma:

Lemma 3.3.1. Let A, B and C be idioms and $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$, $v :: \mathsf{Y} \to \mathsf{B}\ \mathsf{Z}$ and $w :: \mathsf{X} \to \mathsf{C}\ \mathsf{Z}$ be functions with effects in these idioms. Furthermore, let $\psi :: \mathsf{C}\ x \to \mathsf{A}\ (\mathsf{B}\ x)$ be a polymorphic function connecting the three idioms that satisfies:

$$\psi\ (\mathit{pure}\ x) = \mathit{pure}\ (\mathit{pure}\ x) \tag{3.19}$$

$$\psi\ (f \circledast x) = \mathit{pure}\ (\circledast) \circledast \psi\ f \circledast \psi\ x \tag{3.20}$$

If

$$\psi \circ w = \mathit{liftA}\ v \circ u \tag{3.21}$$

then also:

$$\psi \circ \mathit{traverse}\ w = \mathit{liftA}\ (\mathit{traverse}\ v) \circ \mathit{traverse}\ u \tag{3.22}$$

Proof. The function $\psi :: \mathsf{C}\ x \to \mathsf{A}\ (\mathsf{B}\ x)$ satisfies equations (3.19) and (3.20) if and only if $\mathsf{Compose} \circ \psi$ is an idiom morphism:

$$\begin{array}{ll}
& \mathsf{Compose}\ (\psi\ (\mathit{pure}\ x)) \\
= & \langle \text{ property (3.19) of } \psi\ \rangle \\
& \mathsf{Compose}\ (\mathit{pure}\ (\mathit{pure}\ x)) \\
= & \langle \text{ definition of } \mathit{pure} \text{ in } \mathsf{Compose}\ \rangle \\
& \mathit{pure}\ x
\end{array}$$

$$\begin{array}{ll}
& \mathsf{Compose}\ (\psi\ (u \circledast v)) \\
= & \langle \text{ property (3.20) of } \psi\ \rangle \\
& \mathsf{Compose}\ (\mathit{pure}\ (\circledast) \circledast \psi\ u \circledast \psi\ v) \\
= & \langle \text{ definition of } (\circledast) \text{ in } \mathsf{Compose}\ \rangle \\
& \mathsf{Compose}\ (\psi\ u) \circledast \mathsf{Compose}\ (\psi\ v)
\end{array}$$

Now we combine the composition and idiom morphism law:

$$\begin{array}{ll}
& \psi \circ \mathit{traverse}\ w \\
= & \langle \text{ inverse isomorphisms } \rangle \\
& \mathit{runCompose} \circ \mathsf{Compose} \circ \psi \circ \mathit{traverse}\ w \\
= & \langle \text{ idiom morphism law (3.14) } \rangle \\
& \mathit{runCompose} \circ \mathit{traverse}\ (\mathsf{Compose} \circ \psi \circ w) \\
= & \langle \text{ condition (3.21) } \rangle \\
& \mathit{runCompose} \circ \mathit{traverse}\ (\mathsf{Compose} \circ \mathit{liftA}\ v \circ u) \\
= & \langle \text{ composition law (3.13) } \rangle \\
& \mathit{runCompose} \circ \mathsf{Compose} \circ \mathit{liftA}\ (\mathit{traverse}\ v) \circ \mathit{traverse}\ u \\
= & \langle \text{ inverse isomorphisms } \rangle \\
& \mathit{liftA}\ (\mathit{traverse}\ v) \circ \mathit{traverse}\ u
\end{array}$$

□

Using the lemma we can now show Hutton and Fulger's claim in general.

Proof of (3.18). The claim can be reformulated in the following equivalent way: For every t there is some number s (the size of t), such that

$$\begin{aligned}&\textbf{do}\ \{\,t' \leftarrow \mathit{label}\ t; \mathit{return}\ (\mathit{collect}\ t')\,\}\\ =&\\ &\textbf{do}\ \{\,n \leftarrow \mathit{get}; \mathit{put}\ (n+s); \mathit{return}\ (\mathsf{Const}\ [\,n\mathinner{\ldotp\ldotp}n+s-1])\,\}\end{aligned}$$

holds. This means labeling and collecting the labels is the same as increasing the counter by s and returning the corresponding s labels as a list.
We use

```
newtype Sum x = Sum x
instance Num x ⇒ Monoid (Sum x) where
  mempty                 = Sum 0
  Sum s `mappend` Sum s' = Sum (s + s')
tick :: Const (Sum Nat) x
tick = Const (Sum 1)
```

where we assume Nat to be the type of nonnegative integers. We define s by

$$\mathsf{Const}\ (\mathsf{Sum}\ s) = \mathit{traverse}\ (\mathit{const}\ \mathit{tick})\ t.$$

Thus s is found by traversing t and counting the ticks.
The key step of course is applying lemma 3.3.1. To this end choose $\mathsf{A} = \mathsf{State\ Int}$, $\mathsf{B} = \mathsf{Const}\ [\mathsf{Int}]$ and $\mathsf{C} = \mathsf{Const}\ (\mathsf{Sum\ Nat})$. The function u is $\mathit{const\ fresh} :: x \to \mathsf{State\ Int\ Int}$, for v we use $\mathit{note} :: y \to \mathsf{Const}\ [y]\ z$ and w is $\mathit{const\ tick}$. Finally, the role of ψ will be played by ι:

```
ι :: Const (Sum Nat) x → State Int (Const [Int] x)
ι (Const (Sum s)) = do
  n ← get
  put (n + s)
  return (Const [n .. n + s − 1])
```

Here, we use $(+)$ to add integers and naturals without any prior conversion.
The conclusion (3.22) of the lemma is:

$$\iota \circ \mathit{traverse}\ (\mathit{const\ tick}) = \mathit{liftA}\ (\mathit{traverse\ note}) \circ \mathit{traverse}\ (\mathit{const\ fresh})$$

We apply both sides to some t, starting with the right hand side. Since *traverse note* is *collect* and *traverse* (*const fresh*) is *label*, we get *liftA collect* (*label* t) or equivalently

$$\textbf{do}\ \{\,t' \leftarrow \mathit{label}\ t; \mathit{return}\ (\mathit{collect}\ t')\,\}.$$

Applying the left hand side gives $\iota\ (\mathit{traverse}\ (\mathit{const}\ \mathit{tick})\ t)$, which is $\iota\ (\mathsf{Const}\ (\mathsf{Sum}\ s))$. Using the definition of ι we get

$$\mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+s); \mathit{return}\ (\mathsf{Const}\ [n\mathinner{\ldotp\ldotp}n+s-1])\}.$$

Thus the claim is shown once we know the lemma to be applicable. So we check the conditions, starting with the conditions (3.19) and (3.20) for ι:

$$\begin{array}{ll}
 & \iota\ (\mathit{pure}\ x) \\
= & \langle\ \mathit{pure} \text{ in the } \mathsf{Const} \text{ idiom}\ \rangle \\
 & \iota\ (\mathsf{Const}\ \mathit{mempty}) \\
= & \langle\ \mathit{mempty} \text{ in the } \mathsf{Sum} \text{ monoid}\ \rangle \\
 & \iota\ (\mathsf{Const}\ (\mathsf{Sum}\ 0)) \\
= & \langle\ \text{definition of } \iota\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+0); \mathit{return}\ (\mathsf{Const}\ [n\mathinner{\ldotp\ldotp}n+0-1])\} \\
= & \langle\ \text{simplifications}\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ n; \mathit{return}\ (\mathsf{Const}\ [\,])\} \\
= & \langle\ \text{effects are trivial}\ \rangle \\
 & \mathit{return}\ (\mathsf{Const}\ [\,]) \\
= & \langle\ \mathit{pure} = \mathit{return} \text{ for monads and } \mathit{pure} \text{ in } \mathsf{Const} \text{ idiom}\ \rangle \\
 & \mathit{pure}\ (\mathit{pure}\ x)
\end{array}$$

$$\begin{array}{ll}
 & \iota\ (\mathsf{Const}\ (\mathsf{Sum}\ s) \circledast \mathsf{Const}\ (\mathsf{Sum}\ s')) \\
= & \langle\ (\circledast) \text{ in } \mathsf{Const} \text{ idiom}\ \rangle \\
 & \iota\ (\mathsf{Const}\ (\mathsf{Sum}\ s \ \text{`}\mathit{mappend}\text{`}\ \mathsf{Sum}\ s')) \\
= & \langle\ \mathit{mappend} \text{ in } \mathsf{Sum} \text{ monoid}\ \rangle \\
 & \iota\ (\mathsf{Const}\ (\mathsf{Sum}\ (s+s'))) \\
= & \langle\ \text{definition of } \iota\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+s+s'); \mathit{return}\ (\mathsf{Const}\ [n\mathinner{\ldotp\ldotp}n+s+s'-1])\} \\
= & \langle\ \text{list concatenation}\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+s+s'); \\
 & \quad \mathit{return}\ (\mathsf{Const}\ ([n\mathinner{\ldotp\ldotp}n+s-1] \mathbin{+\!\!+} [n+s\mathinner{\ldotp\ldotp}n+s+s'-1]))\} \\
= & \langle\ (\circledast) \text{ in } \mathsf{Const} \text{ idiom}\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+s+s'); \\
 & \quad \mathit{return}\ (\mathsf{Const}\ [n\mathinner{\ldotp\ldotp}n+s-1] \circledast \mathsf{Const}\ [n+s\mathinner{\ldotp\ldotp}n+s+s'-1])\} \\
= & \langle\ \text{additional } \mathit{put}\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+s); \mathit{put}\ (n+s+s'); \\
 & \quad \mathit{return}\ (\mathsf{Const}\ [n\mathinner{\ldotp\ldotp}n+s-1] \circledast \mathsf{Const}\ [n+s\mathinner{\ldotp\ldotp}n+s+s'-1])\} \\
= & \langle\ \text{additional } \mathit{get}\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+s); n' \leftarrow \mathit{get}; \mathit{put}\ (n'+s'); \\
 & \quad \mathit{return}\ (\mathsf{Const}\ [n\mathinner{\ldotp\ldotp}n+s-1] \circledast \mathsf{Const}\ [n'\mathinner{\ldotp\ldotp}n'+s'-1])\} \\
= & \langle\ (\circledast) \text{ in monadic idioms}\ \rangle \\
 & \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+s); \mathit{return}\ ((\circledast)\ (\mathsf{Const}\ [n\mathinner{\ldotp\ldotp}n+s-1]))\} \circledast \\
 & \quad \mathbf{do}\ \{n' \leftarrow \mathit{get}; \mathit{put}\ (n'+s'); \mathit{return}\ (\mathsf{Const}\ [n'\mathinner{\ldotp\ldotp}n'+s'-1])\}
\end{array}$$

$$
\begin{array}{l}
= \quad \langle\ \mathit{liftA} \text{ in monadic idioms } \rangle \\
\quad \mathit{pure}\ (\circledast) \circledast \mathbf{do}\ \{ n \leftarrow \mathit{get}; \mathit{put}\ (n + s); \mathit{return}\ (\mathsf{Const}\ [n\,..\,n + s - 1])\} \circledast \\
\qquad \mathbf{do}\ \{ n' \leftarrow \mathit{get}; \mathit{put}\ (n' + s'); \mathit{return}\ (\mathsf{Const}\ [n'\,..\,n' + s' - 1])\} \\
= \quad \langle\ \text{definition of } \iota\ \rangle \\
\quad \mathit{pure}\ (\circledast) \circledast \iota\ (\mathsf{Const}\ (\mathsf{Sum}\ s)) \circledast \iota\ (\mathsf{Const}\ (\mathsf{Sum}\ s'))
\end{array}
$$

Also we have to make sure (3.21), i.e., $\iota \circ \mathit{const}\ \mathit{tick} = \mathit{liftA}\ \mathit{note} \circ \mathit{const}\ \mathit{fresh}$ holds:

$$
\begin{array}{l}
\quad \iota\ \mathit{tick} \\
= \quad \langle\ \text{definition of } \mathit{tick}\ \rangle \\
\quad \iota\ (\mathsf{Const}\ (\mathsf{Sum}\ 1)) \\
= \quad \langle\ \text{definition of } \iota\ \rangle \\
\quad \mathbf{do}\ \{ n \leftarrow \mathit{get}; \mathit{put}\ (n + 1); \mathit{return}\ (\mathsf{Const}\ [n\,..\,n + 1 - 1])\} \\
= \quad \langle\ \text{simplifications}\ \rangle \\
\quad \mathbf{do}\ \{ n \leftarrow \mathit{get}; \mathit{put}\ (n + 1); \mathit{return}\ (\mathsf{Const}\ [n])\} \\
= \quad \langle\ \text{definition of } \mathit{fresh} \text{ and } \mathit{note}\ \rangle \\
\quad \mathbf{do}\ \{ n \leftarrow \mathit{fresh}; \mathit{return}\ (\mathit{note}\ n)\} \\
= \quad \langle\ \mathit{liftA} \text{ in monadic idioms } \rangle \\
\quad \mathit{liftA}\ \mathit{note}\ \mathit{fresh}
\end{array}
$$

□

What have we learned? First of all Hutton and Fulger's claim is true for all lawful instances of Traversable. This exemplifies how the laws of the Traversable class indeed allow to prove nontrivial things. Yet, while the above proof is comprehensible, it is not straightforward. Even knowing the lemma, several choices have to be made in a consistent way in order to use it. This requires a good intuition for how the various effects interact.

3.4 Finitary Containers

Jaskelioff and Rypacek [2012] show that a certain class of functors called *finitary containers* [Abbott et al., 2003], *dependent polynomial functors* [Gambino and Hyland, 2003] or *functors shapely over lists* [Moggi et al., 1999] are traversable in a natural way. The representation theorem 3.5.2 proves the converse: Every lawful instance of Traversable is (isomorphic to) a finitary container type.

Before discussing the theorem and its proof, it is useful to introduce the notion of finitary containers and give some motivation for the (unusual) concept of shape we will use in the theorem, i.e., make functions. At the beginning of the next section, we will define make functions again in a way that is suitable for the rest of the discussion. No results from this section will be used in the subsequent development.

Due to the nature of the concepts we are going to discuss, the formulas in this section cannot be read as Haskell code. They have to be understood as mathematical formalism or formalization in a dependently typed language. For consistency, we still try to stick to the Haskell notation as much as possible.

3.4.1 Definition

A finitary container [Abbott et al., 2003] is given by a set of shapes S and a function $\mathit{arity} :: \mathsf{S} \to \mathsf{Nat}$ assigning an arity to each shape. A good example to keep in mind is taking S to be the set of binary trees (as a graph, i.e., without labels) and *arity* to be the function that assigns the number of leaves to a tree.
Such a pair defines an extension functor $\mathsf{Ext}_{\mathit{arity}}$[12] given by

$$\mathsf{X} \longmapsto \mathsf{Ext}_{\mathit{arity}}\,\mathsf{X} = \sum_{s \in \mathsf{S}} \mathsf{X}^{\mathit{arity}\ s}$$

i.e., an element of the extension functor applied to X is given by a dependent pair consisting of a shape s and a sequence of *arity* s elements of type X. We will denote the elements as tuples $(s, [x_0, \ldots, x_{n-1}])$, where the list part is also called the *contents* of the extension. We will only use such tuples when the length of the list coincides with the shape's arity and keep track of this condition by hand.
In the example where S is the set of binary trees and *arity* s the number of leaves of s, we get a functor isomorphic to our Tree type. An element of the extension functor at some type X is given by a tree and a sequence of elements of X, while an element of $\mathsf{Tree}\ \mathsf{X}$ is a tree-shaped data structure with elements of X at the leaves.
As a second example, taking $\mathsf{S} = \mathsf{Nat}$ and $\mathit{arity}\ n = n$ leads to vectors of arbitrary length: The shape of a vector is given by its length n and the vector contains n elements of some entry type. In this case the shape is determined by the contents because the shape is the length of the contents list.
In general, the functor structure of $\mathsf{Ext}_{\mathit{arity}}$ is given by

$$\mathit{fmap}\ f\ (s, [x_0, \ldots, x_{n-1}]) = (s, [f\ x_0, \ldots, f\ x_{n-1}])$$

where the list on the right has length *arity* s if and only if the list on the left does.
Jaskelioff and Rypacek [2012] observe, that a $\mathsf{Traversable}$ instance can be given in much the same way:

$$\mathit{traverse}\ u\ (s, [x_0, \ldots, x_{n-1}]) = \mathit{pure}\ (\lambda xs \to (s, xs)) \circledast \mathit{traverse}\ u\ [x_0, \ldots, x_{n-1}].$$

The *traverse* function on the right is the standard instance for lists given in section 3.2. Thus proving correctness of the above instance amounts to proving correctness of the standard instance for lists. The claim of theorem 3.5.1 is that all traversals basically have the above form.

We have seen what one might call the "synthetic approach" to finitary containers: Given a set of shapes and an arity function we have constructed a functor that is traversable in a reasonable way. Now we take a slightly different viewpoint, start with a given functor F and ask whether it is isomorphic to the extension functor of a finitary container.
So let an isomorphism between $\mathsf{F}\ \mathsf{X}$ and $\mathsf{Ext}_{\mathit{arity}}\,\mathsf{X}$ be given that is natural in X. In our context, 'being natural in X' means the isomorphism is given by polymorphic functions

[12] We take S to be implicitly determined as the domain of *arity*.

$\mathsf{F}\ x \to \mathsf{Ext}_{\mathit{arity}}\ x$ and $\mathsf{Ext}_{\mathit{arity}}\ x \to \mathsf{F}\ x$. Starting with $t :: \mathsf{F\ X}$ and applying the isomorphism, we find a shape and a list of contents, such that the shape's arity is the list's length. Thus we can dissect t into its shape and contents, which is expressed by the functions

$$\begin{aligned} \mathit{shape} &:: \mathsf{F}\ x \to \mathsf{S} \\ \mathit{contents} &:: \mathsf{F}\ x \to [x] \end{aligned}$$

and the equation

$$\mathit{arity} \circ \mathit{shape} = \mathit{length} \circ \mathit{contents}.$$

Using the isomorphism in the other direction, we can start with a shape $s :: \mathsf{S}$ and a list of contents $\mathit{xs} :: [\mathsf{X}]$ and combine them into some $t :: \mathsf{F\ X}$. This is only possible if (s, xs) is a valid pair, i.e., $\mathit{arity}\ s = \mathit{length}\ \mathit{xs}$. The process of combining is expressed by the partial function

$$\mathit{fill} :: \mathsf{S} \to [x] \to \mathsf{F}\ x$$

which as stated only works for compatible arguments in the above sense. We refrain from making the partiality explicit in the type by adding Maybe, and confine ourselves to only apply the partial function to suitable arguments.
Finally, we cannot only apply the isomorphism in either direction, but we know these processes to be inverses of each other. That is, starting with some $t :: \mathsf{F\ X}$ we can dissect it into $\mathit{shape}\ t$ and $\mathit{contents}\ t$ and since we know that $\mathit{arity}\ (\mathit{shape}\ t) = \mathit{length}\ (\mathit{contents}\ t)$ we can apply *fill* and

$$\mathit{fill}\ (\mathit{shape}\ t)\ (\mathit{contents}\ t) = t$$

has to hold. Going the other way around, starting with (compatible) $s :: \mathsf{S}$ and $\mathit{xs} :: [\mathsf{X}]$ the two equations

$$\begin{aligned} \mathit{shape}\ (\mathit{fill}\ s\ \mathit{xs}) &= s \\ \mathit{contents}\ (\mathit{fill}\ s\ \mathit{xs}) &= \mathit{xs} \end{aligned}$$

have to hold.
Using the isomorphisms, we can also give a traversal for F using *contents*, *shape* and *fill*:

$$\mathit{traverse}\ u\ t = \mathit{pure}\ (\mathit{fill}\ (\mathit{shape}\ t)) \circledast \mathit{traverse}\ u\ (\mathit{contents}\ t)$$

The application of *fill* is allowed because traversing the list *contents* t with the standard instance does not change the length.

Once the representation theorem is proved, we will know that all traversals are essentially the one given above for some suitable *shape*, *contents* and *fill*. This allows to answer questions about traversables by translating them into questions about finitary containers. For example, given two different traversal functions, how do they relate to each other? Translating this into a question about finitary containers, can a functor be isomorphic to two different finitary containers? Yet easier, can two different finitary containers be isomorphic to each other? Apart from the trivial variations like replacing S by a set of equal size, the only degree of freedom is reordering the factors in $\mathsf{X}^{arity\ s}$. This corresponds to visiting the entries in a traversable in a different order, so that is the only possible difference between two traversing functions for the same type constructor.

3.4.2 Finding a Set of Shapes

The above conditions describe what it means for a given functor to be isomorphic to the extension functor of a given finitary container. Now we go one step further and ask how the finitary container structure can be *found* for a given functor.
Yet, if we only fixed the functor, the result would not be unique in general. Given one finitary container structure, we can find another one by post-composing the *contents* function with some permutation and pre-composing the *fill* function with the inverse permutation. So, in addition to the functor, we assume the function $contents :: \mathsf{F}\ x \to [x]$ to be given. This fixes the order of the entries.

Both the set of shapes and the *arity* function are uniquely determined by the functor F together with a given *contents* function. To this end, assume we already knew a solution S, *arity*.
Consider a special entry type – the unit type (), which is only inhabited by the empty tuple (). This choice is reasonable, as a tree with trivial labels at the leafs is just as good as an unlabel tree. For every given length there is only one list of type [()] (lists containing empty tuples). So for this special type the *length* function is invertible and the inverse is

$$\lambda n \to replicate\ n\ () :: \mathsf{Int} \to [()].$$

Thus given a shape $s :: \mathsf{S}$ there is only one compatible list of contents $xs :: [()]$, which is

$$replicate\ (arity\ s)\ ().$$

Applying *fill* to s and this list, we find some $t = fill\ s\ (replicate\ (arity\ s)\ ())$ and $shape\ t = s$ holds. If on the other hand some $t :: \mathsf{F}\ ()$ is given, we can dissect it into its shape and contents. But since the contents are determined by its shape via the arity, we can reconstruct t from $shape\ t$ alone as

$$t = fill\ (shape\ t)\ (replicate\ (arity\ (shape\ t))\ ()).$$

From the above we conclude that S and $\mathsf{F}\ ()$ are isomorphic via the function *shape*. We also claimed that *arity* is determined by the functor F. Indeed, since $arity \circ shape = length \circ contents$ holds and *shape* is an isomorphism for $\mathsf{X} = ()$, the function *arity* is also determined by F. So, if there is a suitable choice for S and *arity* at all, then $\mathsf{S} = \mathsf{F}\ ()$ and $arity\ s = length\ (contents\ s)$ is one.

Since we now know how to find the set of shapes and the arity function for a given functor, we can give an alternative description of finitary containers. There is no more need for a *shape* function to be given, as we can simply define it:

$$\mathit{shape} = \mathit{fmap}\ (\mathit{const}\ ()) :: \mathsf{F}\ x \to \mathsf{F}\ ()$$

We do still need the function $\mathit{contents} :: \mathsf{F}\ x \to [x]$ to be given though. Also, $\mathit{fill} :: \mathsf{F}\ () \to [x] \to \mathsf{F}\ x$ has to be given (which still is a partial function). In the new setting, $\mathit{fill}\ s\ \mathit{xs}$ exists if $\mathit{length}\ (\mathit{contents}\ s) = \mathit{length}\ \mathit{xs}$ or equivalently $\mathit{contents}\ s = \mathit{map}\ (\mathit{const}\ ())\ \mathit{xs}$. The properties we require are the same as before:

$$\begin{aligned} \mathit{shape}\ (\mathit{fill}\ s\ \mathit{xs}) &= s \\ \mathit{contents}\ (\mathit{fill}\ s\ \mathit{xs}) &= \mathit{xs} \\ \mathit{fill}\ (\mathit{shape}\ t)\ (\mathit{contents}\ t) &= t \end{aligned}$$

We can define S and *arity* without using *fill*. Yet to prove that F is isomorphic to $\mathsf{Ext}_{\mathit{arity}}$, the function *fill* and the above laws are required.
This description is usually referred to as *functors shapely over lists* [Moggi et al., 1999]. The *contents* function is a natural transformation between the functor F and the list functor (hence ‘over lists’). The equations amount to the naturality squares

$$\begin{tikzcd} \mathsf{F}\,\mathsf{X} \arrow[r, "\mathit{shape}="', "\mathit{fmap}\ (\mathit{const}\ ())" '] \arrow[d, "\mathit{contents}"'] & \mathsf{F}\ () \arrow[d, "\mathit{contents}"] \\ {[\mathsf{X}]} \arrow[r, "\mathit{map}\ (\mathit{const}\ ())"] & {[()]} \end{tikzcd}$$

being pullback squares.

3.4.3 Finding Another Set of Shapes

The problem with what has been discussed so far is that it cannot be used in Haskell directly. Dealing with the partial function *fill* properly would require spelling out the partiality using Maybe. There however is a different choice for the set of shapes that also arises from the functor F together with a *contents* function and leads to a description better suited for Haskell.
If *fill* is partially applied to a shape s (for any notion of shape), the result is a polymorphic (partial) function of type $[x] \to \mathsf{F}\ x$, expecting a list of length $\mathit{arity}\ s$. So instead of using lists, we can make the length requirement explicit by changing the type to

$$\underbrace{x \to \ldots \to x \to}_{\mathit{arity}\ s} \mathsf{F}\ x,$$

where the arity of the function is the arity of the shape. We call functions arising in this manner *make functions*.
Can all polymorphic functions with a type $x \to \ldots \to x \to \mathsf{F}\ x$ arise like this? No they cannot because make functions are (essentially) partial applications of *fill* and *fill* has

certain properties. Since *shape* (*fill* s *xs*) $= s$, we have *shape* (*make* $x_0 \ldots x_{n-1}) = s$ for the function *make* that arises from *fill* s, independently of the choice of the x_i. That is good news as it tells us how to get back the shape s we started with from its make function: Any saturated application of the make function will result in something having the shape s.

Another thing we know about (partial) applications of *fill* is that *contents* (*fill* s *xs*) $=$ *xs*. Therefore all make functions have to satisfy

$$\mathit{contents}\ (\mathit{make}\ x_0 \ldots x_{n-1}) = [x_0, \ldots, x_{n-1}]. \tag{3.23}$$

Because of naturality

$$\mathit{fmap}\ f\ (\mathit{make}\ x_0 \ldots x_{n-1}) = \mathit{make}\ (f\ x_0) \ldots (f\ x_{n-1}) \tag{3.24}$$

has to hold. This latter property is automatic in Haskell due to free theorems [Wadler, 1989].

What can we conclude from *fill* (*shape* t) (*contents* t) $= t$? If we start with $t :: \mathsf{F\ X}$ and take the make function arising from *shape* t, then

$$t = \mathit{make}\ x_0 \ldots x_{n-1}$$

where $[x_0, \ldots, x_{n-1}] =$ *contents* t. So t can be reconstructed from its shape and contents by applying the make function to the contents.

We defined make functions as functions arising as partial applications of *fill* and derived properties (3.23) and (3.24). We can also go the other way around and start with a polymorphic function *make* satisfying (3.23) and (3.24). Does it define a shape in the original sense? Every $\mathsf{F\ X}$ that results from an application of a given *make* does indeed have the same shape because of (3.24):

$$\begin{array}{ll}
& \mathit{shape}\ (\mathit{make}\ x_0 \ldots x_{n-1}) \\
= & \langle \text{ property of (!!) } \rangle \\
& \mathit{shape}\ (\mathit{make}\ ([x_0, \ldots, x_{n-1}]\ !!\ 0) \ldots ([x_0, \ldots, x_{n-1}]\ !!\ (n-1))) \\
= & \langle \text{ (3.24) } \rangle \\
& \mathit{shape}\ (\mathit{fmap}\ ([x_0, \ldots, x_{n-1}]!!)\ (\mathit{make}\ 0 \ldots (n-1))) \\
= & \langle \text{ functoriality of } \mathit{shape} \ \rangle \\
& \mathit{shape}\ (\mathit{make}\ 0 \ldots (n-1))
\end{array}$$

If we partially apply *fill* to this shape, can we recover the function *make*?

$$\begin{array}{ll}
& \mathit{fill}\ (\mathit{shape}\ (\mathit{make}\ 0 \ldots (n-1)))\ [x_0, \ldots, x_{n-1}] \\
= & \langle \text{ (3.23) } \rangle \\
& \mathit{fill}\ (\mathit{shape}\ (\mathit{make}\ 0 \ldots (n-1)))\ (\mathit{contents}\ (\mathit{make}\ x_0 \ldots x_{n-1})) \\
= & \langle \text{ see above } \rangle \\
& \mathit{fill}\ (\mathit{shape}\ (\mathit{make}\ x_0 \ldots x_{n-1}))\ (\mathit{contents}\ (\mathit{make}\ x_0 \ldots x_{n-1}))
\end{array}$$

$$
\begin{array}{ll}
= & \langle \text{ pullback property } \rangle \\
 & \mathit{make}\ x_0 \ldots x_{n-1}
\end{array}
$$

Thus make functions and shapes are in one-to-one correspondence.
The drawback of this representation is that make functions are rather implicit as a description of shape because they are functions. Also, extracting the make function from a given container is somewhat difficult. On the other hand when dealing with code, we rarely strip a tree of all its contents explicitly and refill the naked shape later. Instead we use *fmap* and *traverse* in the code and only *reason* about the shape to prove properties of a given function. So there actually is no need to have a type that represents all shapes and therefore we are content with the implicit description of 'a polymorphic function of some arity'. For any given shape we can write down the function and that is good enough to reason about shapes in a piece of code that probably never mentions shapes explicitly.

3.5 The Representation Theorem

Before we come to the main result of this chapter, let us recapitulate. We have introduced the Traversable class and seen examples of how one might want to use its method *traverse*. We have also discussed the laws which the class is usually required to satisfy (section 3.2). We have seen how using these laws, proofs about programs using *traverse* can be carried out. Such proofs are comprehensible, but not easily found (section 3.3). On the other hand we have seen finitary containers – a class of functors which are traversable in a straightforward manner (section 3.4). Moreover proofs about finitary containers are equally straightforward as shape and contents can be dealt with independently.
In this section, we prove a result called *representation theorem* which basically shows that all (lawfully) traversable functors are in fact finitary containers. This allows to reason about any (lawful) traversal the same way we think about finitary containers, i.e., shape and contents can be separated and *traverse* is only concerned with the contents. Thus, short and elegant proofs become possible.

The starting point are traversable functors and therefore *contents* is again defined using *traverse* (as opposed to being some primitive as in the previous section):

$$
\begin{array}{l}
\mathit{note} :: x \to \mathsf{Const}\ [x]\ y \\
\mathit{note}\quad x = \mathsf{Const}\ [x] \\
\mathit{contents} :: \mathsf{Traversable}\ t \Rightarrow t\ x \to [x] \\
\mathit{contents} = \mathit{getConst} \circ \mathit{traverse}\ \mathit{note}
\end{array}
$$

We use the notion of make function from the previous section with the following definition. A make function (for some traversable type T) is a polymorphic function $\mathit{make} :: x \to \ldots \to x \to \mathsf{T}\ x$ of any arity satisfying:

$$\mathit{contents}\ (\mathit{make}\ x_0 \ldots x_{n-1}) = [x_0, \ldots, x_{n-1}] \tag{3.23}$$

$$\mathit{fmap}\ f\ (\mathit{make}\ x_0 \ldots x_{n-1}) = \mathit{make}\ (f\ x_0) \ldots (f\ x_{n-1}) \tag{3.24}$$

Thus, being a make function determines the result of some applications of *traverse*. Condition (3.23) implies

$$\mathit{traverse}\ \mathit{note}\ (\mathit{make}\ x_0 \ldots x_{n-1}) = \mathsf{Const}\ [x_0, \ldots, x_{n-1}]$$

and condition (3.24) implies

$$\mathit{traverse}\ (\mathsf{Identity} \circ f)\ (\mathit{make}\ x_0 \ldots x_{n-1}) = \mathsf{Identity}\ (\mathit{make}\ (f\ x_0) \ldots (f\ x_{n-1})).$$

The first theorem claims that indeed all applications of *traverse* to *make* $x_0 \ldots x_{n-1}$ follow a general pattern that contains the above two as special cases.

Theorem 3.5.1. Let X and Y be arbitrary types, A an idiom and T a (lawfully) traversable type constructor. Let $x_i :: \mathsf{X}$ be a series of values, $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$ some computation with effects in A and *make* a make function for T. Then the following equation holds in $\mathsf{A}\ (\mathsf{T}\ \mathsf{Y})$:

$$\mathit{traverse}\ u\ (\mathit{make}\ x_0 \ldots x_{n-1}) = \mathit{pure}\ \mathit{make} \circledast u\ x_0 \circledast \ldots \circledast u\ x_{n-1} \tag{3.25}$$

If A is a monad, equivalently:

$$\begin{gathered}\mathit{traverse}\ u\ (\mathit{make}\ x_0 \ldots x_{n-1}) \\ = \\ \mathbf{do}\ \{y_0 \leftarrow u\ x_0; \ldots; y_{n-1} \leftarrow u\ x_{n-1}; \mathit{return}\ (\mathit{make}\ y_0 \ldots y_{n-1})\}\end{gathered} \tag{3.26}$$

The second theorem states that every traversable object can be split into its shape and contents in a unique way:

Theorem 3.5.2. Let some $t :: \mathsf{T}\ \mathsf{X}$ be given where X is an arbitrary type and T a (lawfully) traversable type constructor. There is a unique n, a unique n-ary make function *make* and unique values $x_0, \ldots, x_{n-1} :: \mathsf{X}$, such that

$$t = \mathit{make}\ x_0 \ldots x_{n-1}. \tag{3.27}$$

So, every traversable can be split into shape and contents in a unique way. Once we know the decomposition, we can compute the result of any traversal easily using theorem 3.5.1. In [Bird et al., 2013] both theorems are presented together as ‘every t can be represented as in (3.27) and the function *make* “so obtained” satisfies (3.25).’ There is no actual restriction in *make* being obtained via 3.5.2: Any application *make* $x_0 \ldots x_{n-1}$ gives a t whose representation is *make* $x_0 \ldots x_{n-1}$ because of the representation being unique. While there is no new insight in the formulation given here, we find the claim to be clearer. The remainder of this section is dedicated to proving both theorems and closely follows the proof given in the paper.

3.5.1 The Batch Idiom

The function $traverse :: (\mathsf{Applicative}\ a, \mathsf{Traversable}\ t) \Rightarrow (x \to a\ y) \to t\ x \to a\ (t\ y)$ is polymorphic in x, y, a and t. Since it is a method of the Traversable class, it is ad-hoc polymorphic in t. That is to say *traverse* for any concrete type constructor T is aware of the term constructors of $\mathsf{T\ X}$. For example the *traverse* function for our Tree type can do pattern matching on the Leaf and Node constructors. On the other hand the polymorphism in the idiom is parametric, i.e., the behavior is the same for all idioms. This is formally expressed by the idiom morphism law (3.14) we will use in the proof.
Here, we stick to an informal description and ask: What can a general traversal function look like? Consider an application of *traverse*, say $traverse\ u\ t$ for $t :: \mathsf{T\ X}$ and $u :: \mathsf{X} \to \mathsf{A\ Y}$. The type of $traverse\ u\ t$ is $\mathsf{A\ (T\ Y)}$ so the outermost layer is the idiom, which is opaque for the traversal. The *traverse* function can generate values of the unknown idiomatic type only by applying u to some argument of type X or by using the methods provided by the Applicative class.
Every idiomatic computation can be given in its canonical form, so in particular the result of $traverse\ u\ t$ has a canonical form. This canonical form has to be essentially the same for all applicatives because *traverse* is parametrically polymorphic in a. Thus, we want to build a special idiom that allows us to capture and inspect this canonical form. There will be one special effectful function, which when traversed with produces the canonical form. Once we have obtained this blueprint, we can specialize it to any effectful traversal of the same object.
So how will the canonical form look like in general? The only atomic effectful function *traverse* can use is the one it receives as its higher order argument. We thus expect a canonical form like

$$pure\ f \circledast u\ x_0 \circledast \ldots \circledast u\ x_{n-1}$$

where f is some pure function and the x_i are entries of the traversed object. We will represent this canonical form as

$$\mathsf{P}\ f \boxast x_0 \boxast \ldots \boxast x_{n-1}$$

where P and $\boxast$ are constructors and $\boxast$ associates to the left. We do not need to mark where the u is supposed to be inserted, as this is clear from the structure of the computation. We will also use the shorthand notation

$$\mathsf{P}\ f \boxast_{i=0}^{n-1} x_i$$

analogously to the shorthand notation for canonical forms.

The idiom we will be using is given in figure 3.5. Batch needs three type arguments: The first represents the entry type of the traversable object we want to study. The second type argument represents the entry type of the resulting traversable, which might be different. Finally, the third type argument is the result type of the $\mathsf{Batch}\ x\ y$ idiom itself.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Batch}\ x\ y\ z = \mathsf{P}\ z \mid (\boxast)\ (\mathsf{Batch}\ x\ y\ (y \rightarrow z))\ x\\
&\textbf{instance}\ \mathsf{Applicative}\ (\mathsf{Batch}\ x\ y)\ \textbf{where}\\
&\quad \mathit{pure}\ z && = \mathsf{P}\ z\\
&\quad \mathsf{P}\ f \circledast \mathsf{P}\ z && = \mathsf{P}\ (f\ z)\\
&\quad (v \boxast x) \circledast \mathsf{P}\ z && = (\mathsf{P}\ ((\circ)\ (\lambda f \rightarrow f\ z)) \circledast v) \boxast x\\
&\quad v \circledast (w \boxast x) && = (\mathsf{P}\ (\circ) \circledast v \circledast w) \boxast x\\
&\mathit{batch} :: x \rightarrow \mathsf{Batch}\ x\ y\ y\\
&\mathit{batch}\quad x\ =\ \mathsf{P}\ \mathit{id} \boxast x\\
&\mathit{runWith} :: \mathsf{Applicative}\ a \Rightarrow (x \rightarrow a\ y) \rightarrow \mathsf{Batch}\ x\ y\ z \rightarrow a\ z\\
&\mathit{runWith}\ u\ (\mathsf{P}\ z) \quad = \mathit{pure}\ z\\
&\mathit{runWith}\ u\ (v \boxast x) = \mathit{runWith}\ u\ v \circledast u\ x
\end{aligned}
$$

Figure 3.5: Definition of the Batch idiom and its interface

The type of the P constructor is $z \rightarrow \mathsf{Batch}\ x\ y\ z$ and it is indeed used as the *pure* method in the Applicative instance. The type of $(\boxast)$ is

$$(\boxast) :: \mathsf{Batch}\ x\ y\ (y \rightarrow z) \rightarrow x \rightarrow \mathsf{Batch}\ x\ y\ z,$$

so the right field is an entry of the traversed object and the left field is another computation. The result type of the nested computation expects an additional argument of type y, which will be the result of applying an actually effectful function to the right field later.

The *runWith* function is straightforward from the above discussion. When actually running the blueprint computation the constructors are turned into idiomatic primitives and some given effectful function u is inserted at the appropriate places. The *batch* function is made such that $\mathit{batch}\ x$ turns into $u\ x$ once the computation is run. The normal form of $u\ x$ is $\mathit{pure}\ \mathit{id} \circledast u\ x$, and so $\mathit{batch}\ x$ has to be $\mathsf{P}\ \mathit{id} \boxast x$.

The only tricky part of the definition are the rules for $(\circledast)$ in the Batch idiom itself. The first rule has to hold because of the homomorphism law (3.5) and the fact that *pure* is P. The guiding principle behind the other two rules is that in order for Batch to faithfully represent the general effectful computation $\mathit{traverse}\ u\ t$ for arbitrary t, we want $\mathit{runWith}\ u$ to be an idiom morphism. Thus,

$$\mathit{runWith}\ u\ ((v \boxast x) \circledast \mathsf{P}\ z) = \mathit{runWith}\ u\ (v \boxast x) \circledast \mathit{runWith}\ u\ (\mathsf{P}\ z)$$

has to hold. The right hand side can be transformed as follows:

$$
\begin{aligned}
&\ \ \mathit{runWith}\ u\ (v \boxast x) \circledast \mathit{runWith}\ u\ (\mathsf{P}\ z)\\
=&\quad \langle\ \text{definition of } \mathit{runWith}\ \rangle\\
&\ \ (\mathit{runWith}\ u\ v \circledast u\ x) \circledast \mathit{pure}\ z\\
=&\quad \langle\ \text{interchange law (3.6)}\ \rangle
\end{aligned}
$$

$$\begin{array}{ll}
& \mathit{pure}\ (\lambda f \to f\ z) \circledast (\mathit{runWith}\ u\ v \circledast u\ x) \\
= & \langle\ \text{composition law (3.4)}\ \rangle \\
& \mathit{pure}\ (\circ) \circledast \mathit{pure}\ (\lambda f \to f\ z) \circledast \mathit{runWith}\ u\ v \circledast u\ x \\
= & \langle\ \text{homomorphism law (3.5)}\ \rangle \\
& \mathit{pure}\ ((\circ)\ (\lambda f \to f\ z)) \circledast \mathit{runWith}\ u\ v \circledast u\ x \\
= & \langle\ \text{assuming } \mathit{runWith} \text{ is an idiom morphism}\ \rangle \\
& \mathit{runWith}\ u\ (\mathsf{P}\ ((\circ)\ (\lambda f \to f\ z)) \circledast v) \circledast u\ x \\
= & \langle\ \text{definition of } \mathit{runWith}\ \rangle \\
& \mathit{runWith}\ u\ ((\mathsf{P}\ ((\circ)\ (\lambda f \to f\ z)) \circledast v) \boxast x)
\end{array}$$

So by defining $(v \boxast x) \circledast \mathsf{P}\ z = (\mathsf{P}\ ((\circ)\ (\lambda f \to f\ z)) \circledast v) \boxast x$, we make sure the above equation holds. This justifies the second rule for $(\circledast)$ and by a similar calculation we can justify the third rule:

$$\begin{array}{ll}
& \mathit{runWith}\ u\ v \circledast \mathit{runWith}\ u\ (w \boxast x) \\
= & \langle\ \text{definition of } \mathit{runWith}\ \rangle \\
& \mathit{runWith}\ u\ v \circledast (\mathit{runWith}\ u\ w \circledast u\ x) \\
= & \langle\ \text{composition law (3.4)}\ \rangle \\
& \mathit{pure}\ (\circ) \circledast \mathit{runWith}\ u\ v \circledast \mathit{runWith}\ u\ w \circledast u\ x \\
= & \langle\ \text{assuming } \mathit{runWith} \text{ is an idiom morphism}\ \rangle \\
& \mathit{runWith}\ u\ (\mathsf{P}\ (\circ) \circledast v \circledast w) \circledast u\ x \\
= & \langle\ \text{definition of } \mathit{runWith}\ \rangle \\
& \mathit{runWith}\ u\ ((\mathsf{P}\ (\circ) \circledast v \circledast w) \boxast x)
\end{array}$$

Now that we have seen all definitions and the ideas behind those definitions, we can start to prove some helpful statements about Batch.

Lemma 3.5.3. The equation

$$\mathit{runWith}\ u\ (\mathsf{P}\ f \boxast_{i=0}^{n-1} x_i) = \mathit{pure}\ f \circledast_{i=0}^{n-1} u\ x_i$$

holds for all idioms A, series of $x_i :: \mathsf{X}$ and functions $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$ and $f :: \mathsf{Y} \to \ldots \to \mathsf{Y} \to \mathsf{Z}$.

Proof. The proof is by induction, where the base case is obvious from the first rule of $\mathit{runWith}$. For the induction step:

$$\begin{array}{ll}
& \mathit{runWith}\ u\ (\mathsf{P}\ f \boxast_{i=0}^{n} x_i) \\
= & \langle\ \text{splitting of last } (\boxast)\ \rangle \\
& \mathit{runWith}\ u\ ((\mathsf{P}\ f \boxast_{i=0}^{n-1} x_i) \boxast x_n) \\
= & \langle\ \text{definition of } \mathit{runWith}\ \rangle \\
& \mathit{runWith}\ u\ (\mathsf{P}\ f \boxast_{i=0}^{n-1} x_i) \circledast u\ x_n \\
= & \langle\ \text{induction hypothesis}\ \rangle \\
& (\mathit{pure}\ f \circledast_{i=0}^{n-1} u\ x_i) \circledast u\ x_n \\
= & \langle\ \text{absorbing last } (\circledast)\ \rangle \\
& \mathit{pure}\ f \circledast_{i=0}^{n} u\ x_i
\end{array}$$

□

Lemma 3.5.4. For all idioms A and functions $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$:

$$\mathit{runWith}\ u \circ \mathit{batch} = u$$

Proof. By applying both to some x:

$$\begin{array}{ll}
 & \mathit{runWith}\ u\ (\mathit{batch}\ x) \\
= & \quad \langle \text{ definition of } \mathit{batch} \ \rangle \\
 & \mathit{runWith}\ u\ (\mathsf{P}\ \mathit{id} \boxast x) \\
= & \quad \langle \text{ definition of } \mathit{runWith} \ \rangle \\
 & \mathit{pure}\ \mathit{id} \circledast u\ x \\
= & \quad \langle \text{ identity law (3.3) } \rangle \\
 & u\ x
\end{array}$$

□

Lemma 3.5.5. The $\mathsf{Applicative}$ instance for $\mathsf{Batch}\ x\ y$ is correct, i.e., satisfies the idiom laws (3.3) – (3.6).

Proof. Define a function $\mathit{weight} :: \mathsf{Batch}\ x\ y\ z \to \mathsf{Int}$ counting the $(\boxast)$ constructors. An easy joint induction proof shows that $u \circledast v$ is well-defined[13] and $\mathit{weight}\ (u \circledast v) = \mathit{weight}\ u + \mathit{weight}\ v$.
The identity law (3.3) is proved by induction, where the base case is:

$$\begin{array}{ll}
 & \mathsf{P}\ \mathit{id} \circledast \mathsf{P}\ z \\
= & \quad \langle \text{ first rule for } (\circledast) \ \rangle \\
 & \mathsf{P}\ (\mathit{id}\ z) \\
= & \quad \langle \text{ definition of } \mathit{id} \ \rangle \\
 & \mathsf{P}\ z
\end{array}$$

The induction step is as follows:

$$\begin{array}{ll}
 & \mathsf{P}\ \mathit{id} \circledast (v \boxast x) \\
= & \quad \langle \text{ third rule for } (\circledast) \ \rangle \\
 & (\mathsf{P}\ (\circ) \circledast \mathsf{P}\ \mathit{id} \circledast v) \boxast x \\
= & \quad \langle \text{ first rule for } (\circledast) \ \rangle \\
 & (\mathsf{P}\ ((\circ)\ \mathit{id}) \circledast v) \boxast x \\
= & \quad \langle \ \mathit{id} \circ f = f, \text{ so } (\circ)\ \mathit{id} = \mathit{id} \ \rangle \\
 & (\mathsf{P}\ \mathit{id} \circledast v) \boxast x \\
= & \quad \langle \text{ induction hypothesis } \rangle \\
 & v \boxast x
\end{array}$$

The homomorphism law (3.5) is obvious from the first rule of $(\circledast)$. The interchange law (3.6) requires to distinguish two cases, but no induction. Pure case:

[13] That means using the same definition in a language that is not total, evaluation of $(\circledast)$ terminates for total arguments.

$$\begin{aligned}
&\mathsf{P}\ f \circledast \mathsf{P}\ z\\
=\quad&\langle \text{ first rule for } (\circledast)\ \rangle\\
&\mathsf{P}\ (f\ z)\\
=\quad&\langle \text{ beta equivalence } \rangle\\
&\mathsf{P}\ ((\lambda g \to g\ z)\ f)\\
=\quad&\langle \text{ first rule for } (\circledast)\ \rangle\\
&\mathsf{P}\ (\lambda g \to g\ z) \circledast \mathsf{P}\ f
\end{aligned}$$

Application case:

$$\begin{aligned}
&(v \boxast x) \circledast \mathsf{P}\ z\\
=\quad&\langle \text{ second rule for } (\circledast)\ \rangle\\
&(\mathsf{P}\ ((\circ)\ (\lambda g \to g\ z)) \circledast v) \boxast x\\
=\quad&\langle \text{ first rule for } (\circledast)\ \rangle\\
&(\mathsf{P}\ (\circ) \circledast \mathsf{P}\ (\lambda g \to g\ z) \circledast v) \boxast x\\
=\quad&\langle \text{ third rule for } (\circledast)\ \rangle\\
&\mathsf{P}\ (\lambda g \to g\ z) \circledast (v \boxast x)
\end{aligned}$$

The composition law (3.4) can be checked in a similar manner, but we leave out the details here. □

The next lemma provides means to work with the canonical forms without having to descend to the recursive definition of $(\circledast)$ in the Batch idiom.

Lemma 3.5.6. For natural $0 \le m \le n$, a sequence $x_i :: \mathsf{X}$ and functions $f :: \mathsf{Y} \to \ldots \to \mathsf{Y} \to \mathsf{Z}$ and $g :: \mathsf{Y} \to \ldots \to \mathsf{Y} \to \mathsf{Z} \to \mathsf{Z}'$ the following holds:

$$(\mathsf{P}\ g \boxast_{i=0}^{m-1} x_i) \circledast (\mathsf{P}\ f \boxast_{i=m}^{n-1} x_i) = \mathsf{P}\ (g \circ_{m,n} f) \boxast_{i=0}^{n-1} x_i \tag{3.28}$$

In particular for $m = 0$:

$$\mathit{liftA}\ g\ (\mathsf{P}\ f \boxast_{i=0}^{n-1} x_i) = \mathsf{P}\ (g \circ_{0,n} f) \boxast_{i=0}^{n-1} x_i \tag{3.29}$$

Proof. We first show that $\mathit{runWith}\ \mathit{batch} = \mathit{id}$ by induction. The base case follows from the first rule of $\mathit{runWith}$ and the fact that $\mathit{pure} = \mathsf{P}$ in the Batch idiom. For the induction step:

$$\begin{aligned}
&\mathit{runWith}\ \mathit{batch}\ (v \boxast x)\\
=\quad&\langle \text{ definition of } \mathit{runWith}\ \rangle\\
&\mathit{runWith}\ \mathit{batch}\ v \circledast \mathit{batch}\ x\\
=\quad&\langle \text{ induction hypothesis } \rangle\\
&v \circledast \mathit{batch}\ x\\
=\quad&\langle \text{ definition of } \mathit{batch}\ \rangle\\
&v \circledast (\mathsf{P}\ \mathit{id} \boxast x)\\
=\quad&\langle \text{ third rule for } (\circledast)\ \rangle\\
&(\mathsf{P}\ (\circ) \circledast v \circledast \mathsf{P}\ \mathit{id}) \boxast x
\end{aligned}$$

$$\begin{aligned}
= &\quad \langle \text{ interchange law (3.6) } \rangle \\
&(\mathsf{P}\ (\lambda f \to f\ id) \circledast (\mathsf{P}\ (\circ) \circledast v)) \boxast x \\
= &\quad \langle \text{ composition law (3.4) } \rangle \\
&(\mathsf{P}\ (\circ) \circledast \mathsf{P}\ (\lambda f \to f\ id) \circledast \mathsf{P}\ (\circ) \circledast v) \boxast x \\
= &\quad \langle \text{ homomorphism law (3.5) (twice) } \rangle \\
&(\mathsf{P}\ ((\lambda f \to f\ id) \circ (\circ)) \circledast v) \boxast x \\
= &\quad \langle \text{ see below } \rangle \\
&(\mathsf{P}\ id \circledast v) \boxast x \\
= &\quad \langle \text{ identity law (3.3) } \rangle \\
&v \boxast x
\end{aligned}$$

Here, we have used that $(\lambda f \to f\ id) \circ (\circ) = id$. Indeed:

$$\begin{aligned}
&((\lambda f \to f\ id) \circ (\circ))\ z \\
= &\quad \langle \text{ definition of } (\circ) \rangle \\
&(\lambda f \to f\ id)\ ((\circ)\ z) \\
= &\quad \langle \text{ beta equivalence } \rangle \\
&(\circ)\ z\ id \\
= &\quad \langle \text{ property of } (\circ) \rangle \\
&z
\end{aligned}$$

Now, we have

$$\begin{aligned}
&(\mathsf{P}\ g \boxast_{i=0}^{m-1} x_i) \circledast (\mathsf{P}\ f \boxast_{i=m}^{n-1} x_i) \\
= &\quad \langle\ runWith\ batch = id\ \rangle \\
&runWith\ batch\ (\mathsf{P}\ g \boxast_{i=0}^{m-1} x_i) \circledast runWith\ batch\ (\mathsf{P}\ f \boxast_{i=m}^{n-1} x_i) \\
= &\quad \langle \text{ lemma 3.5.3 } \rangle \\
&(pure\ g \circledast_{i=0}^{m-1} batch\ x_i) \circledast (pure\ f \circledast_{i=m}^{n-1} batch\ x_i) \\
= &\quad \langle \text{ flattening formula (3.7) } \rangle \\
&pure\ (g \circ_{m,n} f) \circledast_{i=0}^{n-1} batch\ x_i \\
= &\quad \langle \text{ lemma 3.5.3 } \rangle \\
&runWith\ batch\ (\mathsf{P}\ (g \circ_{m,n} f) \boxast_{i=0}^{n-1} x_i) \\
= &\quad \langle\ runWith\ batch = id\ \rangle \\
&\mathsf{P}\ (g \circ_{m,n} f) \boxast_{i=0}^{n-1} x_i
\end{aligned}$$

□

Lemma 3.5.7. For all $u :: \mathsf{X} \to \mathsf{A}\ \mathsf{Y}$ the function

$$runWith\ u :: \mathsf{Batch}\ \mathsf{X}\ \mathsf{Y}\ z \to \mathsf{A}\ z$$

is an idiom morphism from $\mathsf{Batch}\ \mathsf{X}\ \mathsf{Y}$ to A.

Proof. Equation (3.10) is obvious from the first rule of *runWith*. The proof of (3.11) relies on the two flattening formulas:

$$\begin{aligned}
&\quad \mathit{runWith}\ u\ ((\mathsf{P}\ g \boxast_{i=0}^{m-1} x_i) \circledast (\mathsf{P}\ f \boxast_{i=m}^{n-1} x_i))\\
&= \quad \langle \text{ flattening formula (3.28) } \rangle\\
&\quad \mathit{runWith}\ u\ (\mathsf{P}\ (g \circ_{m,n} f) \boxast_{i=0}^{n-1} x_i)\\
&= \quad \langle \text{ lemma 3.5.3 } \rangle\\
&\quad \mathit{pure}\ (g \circ_{m,n} f) \circledast_{i=0}^{n-1} u\ x_i\\
&= \quad \langle \text{ flattening formula (3.7) } \rangle\\
&\quad (\mathit{pure}\ g \circledast_{i=0}^{m-1} u\ x_i) \circledast (\mathit{pure}\ f \circledast_{i=m}^{n-1} u\ x_i)\\
&= \quad \langle \text{ lemma 3.5.3 } \rangle\\
&\quad \mathit{runWith}\ u\ (\mathsf{P}\ g \boxast_{i=0}^{m-1} x_i) \circledast \mathit{runWith}\ u\ (\mathsf{P}\ f \boxast_{i=m}^{n-1} x_i)
\end{aligned}$$

□

3.5.2 Proof of the Representation Theorem

Now that we have readied our main tool, the Batch idiom, we can start proving the theorems. The first lemma already tells us how to find all the pieces of the representation, but no claim is made that they actually possess the described properties. In particular, *mk* is not yet claimed to be a make function.

Lemma 3.5.8. For every $t :: \mathsf{T}\ \mathsf{X}$ there is a polymorphic function $\mathit{mk} :: y \rightarrow \ldots \rightarrow y \rightarrow \mathsf{T}\ y$ and values $x_0, \ldots, x_{n-1} :: \mathsf{X}$ with

$$\mathit{traverse}\ u\ t = \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{n-1} u\ x_i \tag{3.30}$$

$$\mathit{fmap}\ f\ t = \mathit{mk}\ (f\ x_0) \ldots (f\ x_{n-1}) \tag{3.31}$$

for all types Y, idioms A, functions $u :: \mathsf{X} \rightarrow \mathsf{A}\ \mathsf{Y}$ and $f :: \mathsf{X} \rightarrow \mathsf{Y}$.

Proof. The polymorphic value $\mathit{traverse}\ \mathit{batch}\ t :: \mathsf{Batch}\ \mathsf{X}\ y\ (\mathsf{T}\ y)$ has to have some concrete representation

$$\mathit{traverse}\ \mathit{batch}\ t = \mathsf{P}\ \mathit{mk} \boxast_{i=0}^{n-1} x_i. \tag{3.32}$$

This step fixes the number n, the n-ary polymorphic function $\mathit{mk} :: y \rightarrow \ldots \rightarrow y \rightarrow \mathsf{T}\ y$ and the n values $x_0, \ldots, x_{n-1} :: \mathsf{X}$. Here, we made essential use of our language being total. Now for the claimed equation (3.30):

$$\begin{aligned}
&\quad \mathit{traverse}\ u\ t\\
&= \quad \langle \text{ lemma 3.5.4 } \rangle\\
&\quad \mathit{traverse}\ (\mathit{runWith}\ u \circ \mathit{batch})\ t\\
&= \quad \langle \text{ idiom morphism law (3.14) and lemma 3.5.7 } \rangle\\
&\quad \mathit{runWith}\ u\ (\mathit{traverse}\ \mathit{batch}\ t)\\
&= \quad \langle \text{ concrete representation (3.32) } \rangle\\
&\quad \mathit{runWith}\ u\ (\mathsf{P}\ \mathit{mk} \boxast_{i=0}^{n-1} x_i)\\
&= \quad \langle \text{ lemma 3.5.3 } \rangle\\
&\quad \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{n-1} u\ x_i
\end{aligned}$$

(3.31) follows easily:

$$\begin{aligned}
&\quad \mathit{fmap}\ f\ t\\
=&\quad \langle\ \mathit{fmap} \text{ in terms of } \mathit{traverse}\ \rangle\\
&\quad \mathit{runIdentity}\ (\mathit{traverse}\ (\mathsf{Identity} \circ f)\ t)\\
=&\quad \langle\ \text{property (3.30) of } \mathit{mk}\ \rangle\\
&\quad \mathit{runIdentity}\ (\mathit{pure}\ \mathit{mk} \circledast \mathsf{Identity}\ (f\ x_0) \circledast \ldots \circledast \mathsf{Identity}\ (f\ x_{n-1}))\\
=&\quad \langle\ \text{calculation in } \mathsf{Identity} \text{ idiom}\ \rangle\\
&\quad \mathit{runIdentity}\ (\mathsf{Identity}\ (\mathit{mk}\ (f\ x_0) \ldots (f\ x_{n-1})))\\
=&\quad \langle\ \text{inverse isomorphisms}\ \rangle\\
&\quad \mathit{mk}\ (f\ x_0) \ldots (f\ x_{n-1})
\end{aligned}$$

□

Lemma 3.5.9. For the function *mk* given by lemma 3.5.8

$$\mathit{traverse}\ u\ (\mathit{mk}\ y_0 \ldots y_{n-1}) = \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{n-1}\ u\ y_i \tag{3.33}$$

holds for all types Y, Z, idioms A, elements $y_0, \ldots, y_{n-1} :: \mathsf{Y}$ and functions $u :: \mathsf{Y} \to \mathsf{A}\ \mathsf{Z}$.

Proof. The claim is proved by comparing the fields of the P constructor in the first and last line of the following calculation:

$$\begin{aligned}
&\quad \mathsf{Compose}\ (\mathsf{P}\ (\lambda y_0 \ldots y_{n-1} \to \mathit{traverse}\ u\ (\mathit{mk}\ y_0 \ldots y_{n-1}))\ \boxast_{i=0}^{n-1}\ x_i)\\
=&\quad \langle\ \text{definition of } \circ_{\cdot,\cdot}\ \rangle\\
&\quad \mathsf{Compose}\ (\mathsf{P}\ (\mathit{traverse}\ u \circ_{0,n} \mathit{mk})\ \boxast_{i=0}^{n-1}\ x_i)\\
=&\quad \langle\ \text{flattening formula (3.29)}\ \rangle\\
&\quad \mathsf{Compose}\ (\mathit{liftA}\ (\mathit{traverse}\ u)\ (\mathsf{P}\ \mathit{mk}\ \boxast_{i=0}^{n-1}\ x_i))\\
=&\quad \langle\ \text{concrete representation (3.32)}\ \rangle\\
&\quad \mathsf{Compose}\ (\mathit{liftA}\ (\mathit{traverse}\ u)\ (\mathit{traverse}\ \mathit{batch}\ t))\\
=&\quad \langle\ \text{composition law (3.13)}\ \rangle\\
&\quad \mathit{traverse}\ (\mathsf{Compose} \circ \mathit{liftA}\ u \circ \mathit{batch})\ t\\
=&\quad \langle\ \text{lemma 3.5.8}\ \rangle\\
&\quad \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{n-1} \mathsf{Compose}\ (\mathit{liftA}\ u\ (\mathit{batch}\ x_i))\\
=&\quad \langle\ \text{definition of } \mathit{batch}\ \rangle\\
&\quad \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{n-1} \mathsf{Compose}\ (\mathit{liftA}\ u\ (\mathsf{P}\ \mathit{id} \boxast x_i))\\
=&\quad \langle\ \text{flattening formula (3.29)}\ \rangle\\
&\quad \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{n-1} \mathsf{Compose}\ (\mathsf{P}\ u \boxast x_i)\\
=&\quad \langle\ \text{equation (3.8)}\ \rangle\\
&\quad \mathsf{Compose}\ (\mathsf{P}\ (\lambda v_0 \ldots v_{n-1} \to \mathit{pure}\ \mathit{mk} \circledast v_0 \circledast\ \ldots \circledast v_{n-1}) \circledast_{i=0}^{n-1}\ (\mathsf{P}\ u \boxast x_i))\\
=&\quad \langle\ \text{see below}\ \rangle\\
&\quad \mathsf{Compose}\ (\mathsf{P}\ (\lambda y_0 \ldots y_{n-1} \to \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{n-1}\ u\ y_i)\ \boxast_{i=0}^{n-1}\ x_i)
\end{aligned}$$

The last step is an n-fold application of the flattening formula (3.28) for the Batch idiom. More precisely the below is used n times for $m = 0, \ldots, n-1$.

$$\begin{array}{ll}
& \mathsf{P}\ (\lambda y_0 \ldots y_{m-1}\ v_m \ldots v_{n-1} \to \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{m-1} u\ y_i \circledast_{i=m}^{n-1} v_i)\ \boxast_{i=0}^{m-1} x_i \\
& \quad \circledast (\mathsf{P}\ u \boxast x_m) \\
= & \quad \langle \text{ flattening formula (3.28) } \rangle \\
& \mathsf{P}\ ((\lambda y_0 \ldots y_{m-1}\ v_m \ldots v_{n-1} \to \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{m-1} u\ y_i \circledast_{i=m}^{n-1} v_i) \circ_{m,m+1} u) \\
& \quad \boxast_{i=0}^{m} x_i \\
= & \quad \langle \text{ definition of } \circ_{\cdot,\cdot} \rangle \\
& \mathsf{P}\ (\lambda y_0 \ldots y_{m-1}\ y_m \to (\lambda v_m \ldots v_{n-1} \to \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{m-1} u\ y_i \circledast_{i=m}^{n-1} v_i)\ (u\ y_m)) \\
& \quad \boxast_{i=0}^{m} x_i \\
= & \quad \langle \text{ beta equivalence } \rangle \\
& \mathsf{P}\ (\lambda y_0 \ldots y_{m-1}\ y_m\ v_{m+1} \ldots v_{n-1} \to \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{m-1} u\ y_i \circledast u\ y_m \circledast_{i=m+1}^{n-1} v_i) \\
& \quad \boxast_{i=0}^{m} x_i \\
= & \quad \langle \text{ absorbing } (\circledast) \rangle \\
& \mathsf{P}\ (\lambda y_0 \ldots y_{m-1}\ y_m\ v_{m+1} \ldots v_{n-1} \to \mathit{pure}\ \mathit{mk} \circledast_{i=0}^{m} u\ y_i \circledast_{i=m+1}^{n-1} v_i) \\
& \quad \boxast_{i=0}^{m} x_i
\end{array}$$

□

Proof of theorem 3.5.1. Let *make* be a given make function of arity n. Lemma 3.5.8 applied to $t = \mathit{make}\ 0 \ldots (n-1)$ yields m, an m-ary function mk and $i_0, \ldots, i_{m-1} :: \mathsf{Integer}$, such that

$$\mathit{traverse}\ u\ (\mathit{make}\ 0 \ldots (n-1)) = \mathit{pure}\ \mathit{mk} \circledast u\ i_0 \circledast \ldots \circledast u\ i_{m-1}$$

for all types Y, idioms A and $u :: \mathsf{Integer} \to \mathsf{A}\ \mathsf{Y}$. In particular for $u = \mathit{note}$ we have:

$$\begin{array}{ll}
& [0\,..\,(n-1)] \\
= & \quad \langle\ \mathit{make} \text{ satisfies (3.23) } \rangle \\
& \mathit{contents}\ (\mathit{make}\ 0 \ldots (n-1)) \\
= & \quad \langle \text{ definition of } \mathit{contents}\ \rangle \\
& \mathit{getConst}\ (\mathit{traverse}\ \mathit{note}\ (\mathit{make}\ 0 \ldots (n-1))) \\
= & \quad \langle \text{ lemma 3.5.8 } \rangle \\
& \mathit{getConst}\ (\mathit{pure}\ \mathit{mk} \circledast \mathit{note}\ i_0 \circledast \ldots \circledast \mathit{note}\ i_{m-1}) \\
= & \quad \langle \text{ definition of } \mathit{note}\ \rangle \\
& \mathit{getConst}\ (\mathit{pure}\ \mathit{mk} \circledast \mathsf{Const}\ [i_0] \circledast \ldots \circledast \mathsf{Const}\ [i_{m-1}]) \\
= & \quad \langle\ (\circledast) \text{ in } \mathsf{Const} \text{ idiom } \rangle \\
& \mathit{getConst}\ (\mathsf{Const}\ [i_0, \ldots, i_{m-1}]) \\
= & \quad \langle \text{ inverse isomorphisms } \rangle \\
& [i_0, \ldots, i_{m-1}]
\end{array}$$

The two lists being equal implies $m = n$ and $i_j = j$. We also want to prove $\mathit{mk} = \mathit{make}$:

$$\begin{array}{ll}
& \mathit{make}\ y_0 \ldots y_{n-1} \\
= & \quad \langle \text{ property of (!!) } \rangle \\
& \mathit{make}\ ([y_0, \ldots, y_{n-1}]\ !!\ 0) \ldots ([y_0, \ldots, y_{n-1}]\ !!\ (n-1)) \\
= & \quad \langle\ \mathit{make} \text{ satisfies (3.24) } \rangle
\end{array}$$

$$
\begin{aligned}
& \mathit{fmap}\ ([y_0, \ldots, y_{n-1}]!!)\ (\mathit{make}\ 0 \ldots (n-1)) \\
= \quad & \langle \text{ lemma 3.5.8 } \rangle \\
& \mathit{mk}\ ([y_0, \ldots, y_{n-1}]\ !!\ 0) \ldots ([y_0, \ldots, y_{n-1}]\ !!\ (n-1)) \\
= \quad & \langle \text{ property of } (!!) \ \rangle \\
& \mathit{mk}\ y_0 \ldots y_{n-1}
\end{aligned}
$$

Since equality holds for arbitrary $y_0, \ldots, y_{n-1}$ of arbitrary type Y the two functions are equal. Thus, lemma 3.5.9 applies to *make*, which is the claim of the theorem. □

Lemma 3.5.10. The function *mk* given by lemma 3.5.8 is a make function.

Proof. We have to check equations (3.23) and (3.24). Using lemma 3.5.9 for $\mathit{note} :: \mathsf{X} \to \mathsf{Const}\ [\mathsf{X}]\ y$ gives

$$\mathit{traverse\ note}\ (\mathit{mk}\ x_0 \ldots x_{n-1}) = \mathit{pure\ mk} \circledast_{i=0}^{n-1} \mathit{note}\ x_i$$

which is equivalent to

$$\mathsf{Const}\ (\mathit{contents}\ (\mathit{mk}\ x_0 \ldots x_{n-1})) = \mathsf{Const}\ [x_0, \ldots, x_{n-1}],$$

i.e., (3.23) up to a Const wrapper. The steps are similar to the first derivation in the proof of theorem 3.5.1.
Using lemma 3.5.9 again but for $\mathsf{Identity} \circ f :: \mathsf{X} \to \mathsf{Identity\ Y}$ gives

$$\mathit{traverse}\ (\mathsf{Identity} \circ f)\ (\mathit{mk}\ x_0 \ldots x_{n-1}) = \mathit{pure\ mk} \circledast_{i=0}^{n-1} \mathsf{Identity}\ (f\ x_i)$$

which is equivalent to

$$\mathsf{Identity}\ (\mathit{fmap}\ f\ (\mathit{mk}\ x_0 \ldots x_{n-1})) = \mathsf{Identity}\ (\mathit{mk}\ (f\ x_0) \ldots (f\ x_{n-1})),$$

i.e., (3.24) up to an Identity wrapper. The steps are similar to the second derivation in the proof of lemma 3.5.8. □

Proof of theorem 3.5.2. Let $t :: \mathsf{T\ X}$ be given. Fix n, $\mathit{mk} :: y \to \ldots \to y \to \mathsf{T}\ y$ and $x_0, \ldots, x_{n-1} :: \mathsf{X}$ as in lemma 3.5.8. Because of lemma 3.5.10, *mk* is indeed a make function. We have to show that $t = \mathit{mk}\ x_0 \ldots x_{n-1}$:

$$
\begin{aligned}
& t \\
= \quad & \langle \text{ property of } \mathit{fmap} \ \rangle \\
& \mathit{fmap\ id}\ t \\
= \quad & \langle \text{ lemma 3.5.8 } \rangle \\
& \mathit{mk}\ (\mathit{id}\ x_0) \ldots (\mathit{id}\ x_{n-1}) \\
= \quad & \langle \text{ definition of } \mathit{id} \ \rangle \\
& \mathit{mk}\ x_0 \ldots x_{n-1}
\end{aligned}
$$

This concludes the existence part.

Now for uniqueness. If one traversable t has two representations $\mathit{make}\ x_0 \ldots x_{n-1} = \mathit{make}'\ y_0 \ldots y_{m-1}$, then theorem 3.5.1 applies to both make and make'. For the first representation we have:

$$\begin{array}{ll}
 & \mathit{traverse}\ \mathit{batch}\ (\mathit{make}\ x_0 \ldots x_{n-1}) \\
= & \langle\ \text{theorem 3.5.1}\ \rangle \\
 & \mathit{pure}\ \mathit{make} \circledast \mathit{batch}\ x_0 \circledast \ldots \circledast \mathit{batch}\ x_{n-1} \\
= & \langle\ \text{definition of } \mathit{batch}\ \rangle \\
 & \mathit{pure}\ \mathit{make} \circledast (\mathsf{P}\ \mathit{id} \boxast x_0) \circledast \ldots \circledast (\mathsf{P}\ \mathit{id} \boxast x_{n-1}) \\
= & \langle\ \text{flattening formula (3.28)}\ \rangle \\
 & \mathsf{P}\ \mathit{make} \boxast x_0 \boxast \ldots \boxast x_{n-1}
\end{array}$$

And since the analogous statement applies to the second representation,

$$\mathsf{P}\ \mathit{make} \boxast x_0 \boxast \ldots \boxast x_{n-1} = \mathsf{P}\ \mathit{make}' \boxast y_0 \boxast \ldots \boxast y_{m-1}.$$

By inspecting the fields we find that both representations coincide.

□

3.6 Examples

3.6.1 Proving the Labeling Claim using the Representation Theorem

Now that we have the representation theorem at hand, we find an easier proof for the tree labeling claim, i.e.,

$$\mathit{runState}\ (\mathit{label}\ t)\ 0 = (t', n) \implies \mathit{contents}\ t' = [0\,..\,n-1].$$

Given some $t :: \mathsf{Tree\ X}$, we can represent it as $\mathit{make}\ x_0 \ldots x_{n-1}$ using theorem 3.5.2. The result of applying label is calculated easily using theorem 3.5.1:

$$\begin{array}{ll}
 & \mathit{label}\ (\mathit{make}\ x_0 \ldots x_{n-1}) \\
= & \langle\ \text{definition of } \mathit{label}\ \rangle \\
 & \mathit{traverse}\ (\mathit{const}\ \mathit{fresh})\ (\mathit{make}\ x_0 \ldots x_{n-1}) \\
= & \langle\ \text{theorem 3.5.1}\ \rangle \\
 & \mathit{pure}\ \mathit{make} \circledast_{i=0}^{n-1} \mathit{const}\ \mathit{fresh}\ x_i \\
= & \langle\ \text{definition of } \mathit{const}\ \rangle \\
 & \mathit{pure}\ \mathit{make} \circledast_{i=0}^{n-1} \mathit{fresh} \\
= & \langle\ \text{definition of } \mathit{fresh}\ \rangle \\
 & \mathit{pure}\ \mathit{make} \circledast_{i=0}^{n-1} \mathbf{do}\ \{\, l \leftarrow \mathit{get}; \mathit{put}\ (l+1); \mathit{return}\ l \} \\
= & \langle\ \text{rewriting into a } \mathbf{do}\text{-block}\ \rangle \\
 & \mathbf{do}\ \{\, l_0 \leftarrow \mathit{get}; \mathit{put}\ (l_0+1); \ldots; l_{n-1} \leftarrow \mathit{get}; \mathit{put}\ (l_{n-1}+1) \\
 & \quad ; \mathit{return}\ (\mathit{make}\ l_0 \ldots l_{n-1}) \} \\
= & \langle\ \text{simplifying}\ \rangle \\
 & \mathbf{do}\ \{\, l_0 \leftarrow \mathit{get}; \mathit{put}\ (l_0+n); \mathit{return}\ (\mathit{make}\ l_0\ (l_0+1) \ldots (l_0+n-1)) \}
\end{array}$$

```
treverse :: (Traversable t, Applicative a) ⇒ (x → a y) → t x → a (t y)
treverse u = forwards ∘ traverse (Backwards ∘ u)
adorn :: x → State [l] (x, l)
adorn x = do
  l : ls ← get
  put ls
  return (x, l)
label :: Traversable t ⇒ t x → State [l] (t (x, l))
label = traverse adorn
strip :: (x, l) → State [l] x
strip (x, l) = do
  ls ← get
  put (l : ls)
  return x
unlabel :: Traversable t ⇒ t (x, l) → State [l] (t x)
unlabel = treverse strip
```

Figure 3.6: Code base for labeling à la Gibbons and Bird

So applying *runState* with the initial state 0 gives *make* $0 \ldots (n-1)$ as result and n as the final state. And indeed, *contents* (*make* $0 \ldots (n-1)$) $= [0\,..\,n-1]$ because *make* is a make function. (Alternatively we give *contents* in terms of *traverse* and use theorem 3.5.1 a second time.)
This proof is rather short. In fact it is hard to imagine a shorter proof, as all problem specific definitions have to be taken into account somewhere. More importantly, the proof does not introduce any auxiliary definitions. It works with what is given and does so in a very straightforward way.

3.6.2 Inversion Law

In [Bird et al., 2013] we gave a different proof, which first changes the claim in that both labeling and collecting the labels is done in the same monad. The relevant definitions can be seen in figure 3.6. This reformulation by Gibbons and Bird uses a list of labels, which is managed by a State monad. The *adorn* function takes one argument, extracts the first label from the list and returns a tuple consisting of the argument and the label. The original content is not simply overwritten, as this approach crucially depends on effects being reversible. The inverse action is *strip*, which puts the label back to the front of the list. Given the list of labels is long enough,

$$\mathit{adorn}\ x \gg\!\!= \mathit{strip} = \mathit{return}\ x$$

holds.

Now, we might be tempted to assume

$$\mathit{traverse}\ \mathit{adorn}\ t \gg\!= \mathit{traverse}\ \mathit{strip} = \mathit{return}\ t$$

also held, but it does not. The first entry in the initial list of labels is extracted first and becomes the label of the first leaf (in whatever order the tree is traversed). When the labels are collected again, this label is put back first and all subsequently collected labels are put in front of it. Thus, the order of the used labels is reversed.
The solution is to perform either the labeling or the collecting process backwards, i.e., to visit all the leafs in reversed order. This is accomplished by the function *treverse* also defined in figure 3.6, which relies on Backwards idioms (see section 3.1.2). The claim thus becomes

$$\mathit{traverse}\ \mathit{adorn}\ t \gg\!= \mathit{treverse}\ \mathit{strip} = \mathit{return}\ t$$

and this equation does indeed hold as we will see.
In fact, a more general statement holds: If $u :: \mathsf{X} \to \mathsf{M}\ \mathsf{Y}$ and $v :: \mathsf{Y} \to \mathsf{M}\ \mathsf{X}$ are functions such that

$$u\ x \gg\!= v = \mathit{return}\ x \tag{3.1}$$

then also

$$\mathit{traverse}\ u\ t \gg\!= \mathit{treverse}\ v = \mathit{return}\ t \tag{3.2}$$

for any traversable type constructor T and $t :: \mathsf{T}\ \mathsf{X}$. The *adorn* and *strip* functions are an example for two mutually inverse effectful functions u and v. In this case the claim is $\mathit{label}\ t \gg\!= \mathit{unlabel} = \mathit{return}\ t$ since *unlabel* is defined using *treverse*.

To deal with *treverse*, we prove a formula similar to the representation theorem:

$$\mathit{treverse}\ u\ (\mathit{make}\ x_0 \ldots x_{n-1}) = \mathit{pure}\ (\lambda y_{n-1} \ldots y_0 \to \mathit{make}\ y_0 \ldots y_{n-1}) \circledast u\ x_{n-1} \circledast \ldots \circledast u\ x_0$$

The proof relies on equation (3.9):

$$\begin{aligned}
& \mathit{treverse}\ u\ (\mathit{make}\ x_0 \ldots x_{n-1}) \\
= \quad & \langle \text{ definition of } \mathit{treverse} \ \rangle \\
& \mathit{forwards}\ (\mathit{traverse}\ (\mathsf{Backwards} \circ u)\ (\mathit{make}\ x_0 \ldots x_{n-1})) \\
= \quad & \langle \text{ theorem 3.5.1 } \rangle \\
& \mathit{forwards}\ (\mathit{pure}\ \mathit{make} \circledast \mathsf{Backwards}\ (u\ x_0) \circledast \ldots \circledast \mathsf{Backwards}\ (u\ x_{n-1})) \\
= \quad & \langle \text{ equation (3.9) } \rangle \\
& \mathit{forwards}\ (\mathsf{Backwards}\ (\mathit{pure}\ (\lambda y_{n-1} \ldots y_0 \to \mathit{make}\ y_0 \ldots y_{n-1}) \\
& \quad \circledast u\ x_{n-1} \circledast \ldots \circledast u\ x_0)) \\
= \quad & \langle \text{ inverse isomorphisms } \rangle \\
& \mathit{pure}\ (\lambda y_{n-1} \ldots y_0 \to \mathit{make}\ y_0 \ldots y_{n-1}) \circledast u\ x_{n-1} \circledast \ldots \circledast u\ x_0
\end{aligned}$$

If the idiom is monadic, the last line can be written as a **do**-block:

do
  $y_{n-1} \leftarrow u\ x_{n-1}$

```
  $\ldots$
  $y_0 \quad \leftarrow u\ x_0$
  $return\ (make\ y_0 \ldots y_{n-1})$
```

Now for the proof of the inversion formula itself. We again start by assuming t to be given in its decomposed representation $make\ x_0 \ldots x_{n-1}$. Then we have:

```
  do
    $t' \leftarrow traverse\ u\ (make\ x_0 \ldots x_{n-1})$
    $treverse\ v\ t'$
=   ⟨ theorem 3.5.1 ⟩
  do
    $y_0 \quad \leftarrow u\ x_0$
    $\ldots$
    $y_{n-1} \leftarrow u\ x_{n-1}$
    $t' \leftarrow return\ (make\ y_0 \ldots y_{n-1})$
    $treverse\ v\ t'$
=   ⟨ first monad law ⟩
  do
    $y_0 \quad \leftarrow u\ x_0$
    $\ldots$
    $y_{n-1} \leftarrow u\ x_{n-1}$
    $treverse\ v\ (make\ y_0 \ldots y_{n-1})$
=   ⟨ see above ⟩
  do
    $y_0 \quad \leftarrow u\ x_0$
    $\ldots$
    $y_{n-1} \leftarrow u\ x_{n-1}$
    $z_{n-1} \leftarrow v\ y_{n-1}$
    $\ldots$
    $z_0 \quad \leftarrow v\ y_0$
    $return\ (make\ z_0 \ldots z_{n-1})$
=   ⟨ see below ⟩
  do
    $y_0 \quad \leftarrow u\ x_0$
    $\ldots$
    $y_{n-2} \leftarrow u\ x_{n-2}$
    $z_{n-2} \leftarrow v\ y_{n-2}$
    $\ldots$
    $z_0 \quad \leftarrow v\ y_0$
    $return\ (make\ z_0 \ldots z_{n-2}\ x_{n-1})$
=   ⟨ and so on ⟩
  $return\ (make\ x_0 \ldots x_{n-1})$
```

The crucial step is to realize that z_{n-1} is x_{n-1} and the effect of the two middle lines cancel because of the precondition (3.1). This step can be applied n times to cancel all the lines in the **do**-block until only the *return* statement is left.

3.6.3 Composition of Monadic Traversals

Let us quickly return to the proposed monadic composition law

$$\mathit{traverse}\ u \ggg \mathit{traverse}\ v = \mathit{traverse}\ (u \ggg v) \tag{3.17}$$

which we claimed to be wrong in general. Now that we know the representation theorem, we can understand more easily why it does not hold.
Given some $t = \mathit{make}\ x_0 \ldots x_{n-1}$ the left side applied to t is

```
do
  $y_0$     $\leftarrow u\ x_0$
  ...
  $y_{n-1}$ $\leftarrow u\ x_{n-1}$
  $z_0$     $\leftarrow v\ y_0$
  ...
  $z_{n-1}$ $\leftarrow v\ y_{n-1}$
  $\mathit{return}\ (\mathit{make}\ z_0 \ldots z_{n-1})$
```

while the right side applied to t is:

```
do
  $y_0$     $\leftarrow u\ x_0$
  $z_0$     $\leftarrow v\ y_0$
  ...
  $y_{n-1}$ $\leftarrow u\ x_{n-1}$
  $z_{n-1}$ $\leftarrow v\ y_{n-1}$
  $\mathit{return}\ (\mathit{make}\ z_0 \ldots z_{n-1})$
```

The two blocks are clearly syntactically different if $n > 1$ and choices for u and v that actually lead to different behavior can easily be found. For example if u pushes its argument to a stack and v pops the stack's top element, the former **do**-block reverses the contents while the latter does nothing.
Seeing the traversals as **do**-blocks also helps us to identify conditions that make equation (3.17) true. If the monad is commutative, i.e., the order of effects does not matter, the equation holds. This has been shown by Gibbons and Oliveira [2009] already and their proof relies on

$$\mathit{join} \circ \mathit{runCompose} :: \mathsf{Monad}\ m \Rightarrow \mathsf{Compose}\ m\ m\ x \rightarrow m\ x$$

being an idiom morphism. Using the above **do**-blocks we can give an easier proof: In commutative monads we may change the order of statements (unless that conflicts with the variable scopes), so both **do**-blocks are semantically equal.

But even if the monad is not commutative, it might still be fine to swap *certain* computations. Consider for example a monad $\mathsf{QueueState}\ x$ that manages a queue with entries of type x and provides functions:

$$\begin{aligned}&\mathit{enqueue} :: x \to \mathsf{QueueState}\ x\ ()\\ &\mathit{dequeue} :: \qquad \mathsf{QueueState}\ x\ x\end{aligned}$$

The latter function may fail, as the *queue* might be empty (so in particular some effects are not reversible). The monad is not commutative, because the order in which entries are enqueued matters. Neither do enqueuing and dequeuing commute in general because, when starting with an empty queue, dequeuing first leads to failure while the other order is fine. But does

$$\mathit{traverse}\ \mathit{enqueue} \gg \mathit{traverse}\ (\mathit{const}\ \mathit{dequeue}) = \mathit{traverse}\ (\mathit{enqueue} \gg \mathit{const}\ \mathit{dequeue})$$

hold, i.e., are the two **do**-blocks

```
do
  $enqueue\ x_0$
  ...
  $enqueue\ x_{n-1}$
  $z_0 \quad\ \leftarrow dequeue$
  ...
  $z_{n-1} \leftarrow dequeue$
  $return\ (make\ z_0 \ldots z_{n-1})$
```

and

```
do
  $enqueue\ x_0$
  $z_0 \quad\ \leftarrow dequeue$
  ...
  $enqueue\ x_{n-1}$
  $z_{n-1} \leftarrow dequeue$
  $return\ (make\ z_0 \ldots z_{n-1})$
```

semantically equal?

Yes they are: None of the *dequeue* operations leads to failure, because we never extract more entries from the queue than we added before. Both the enqueuing and the dequeuing operations remain in their respective orders and exchanging operations of different types is fine as long as the queue cannot run dry.

In this case we used particular knowledge about queues to convince ourselves that the above equation holds. We could generalize the statement by claiming equation (3.17) whenever a number of preconditions concerning u and v holds. We could try to find the weakest possible conditions or give a list of alternatives to pick from. However, there is little insight to be gained – the equation holds whenever the two written-out **do**-blocks are equal. That is easy enough a condition and can hopefully be decided for every particular case for which it matters.

Chapter 4

Reasoning about Lazy Functional-Logic Languages

Contents

This chapter can be understood as two different things. On the one hand, it is a collection of complementary approaches to understand the semantics of the functional-logic language Curry [Hanus, 2013]. These approaches have their respective advantages and disadvantages, but ultimately they describe the same language and are interrelated. Seeing the chapter this first way, it is a thorough investigation of different topics, each of which is interesting in its own right.

On the other hand, there is a goal, that has driven the research, motivates the whole development, and explains the necessity of all the various considerations. This goal is proving free theorems [Wadler, 1989] for Curry. Seeing the chapter this second way, it is a serpentine route to a definitive end, in which every detour has a forcing reason. So before we dive into the technicalities, we want to outline this serpentine route.

Haskell and Curry both feature parametrically polymorphic functions, i.e., functions that work for different types but relying on the same code. In Haskell, parametric polymorphism implies nontrivial relations, called free theorems, between different instantiations of the same function. Among other things, free theorems are the foundation of short cut fusion [Gill et al., 1993] – a surprising and very useful compiler optimization strategy. It is reasonable to believe that there is something to be found when it comes to Curry as well.

Christiansen et al. [2010] have studied free theorems for Curry in a phenomenological manner and have given a number of example statements they believe to hold. While the paper does not give any proof of the positive results, it does give some counter-examples. These counter-examples guide our intuition as to what can and cannot be expected to hold. The paper states side conditions restricting nondeterminism, that at least exclude the counter-examples, but hopefully turn out to be sufficient already. Also, vocabulary for these side conditions is established.

The need for additional side conditions is not surprising. Staying in the realm of Haskell, one can distinguish between a simplified, total version of Haskell and a succession of better and better approximations to real code. The first step away from the naive version is to include possible failure. Free theorems have to take this into account and the presence of possible failure requires an additional side condition [Wadler, 1989]. The more features we add to our language, the more side conditions we need, to still make free theorems hold [Voigtländer and Johann, 2007]. Considering Curry to be an extended Haskell with built-in nondeterminism, we expect even more such restrictions to appear in free theorems.

The question remains how to prove the positive results around free theorems in Curry. Christiansen et al. [2011a] devised the following plan:

1. Define a denotational semantics for Curry, that is similar to denotational semantics for functional languages.

2. Establish a formal connection between this functional-style denotational semantics and other semantics for Curry.

3. Adapt free theorems and the machinery behind them from the functional setting to the functional-logic setting.

The remainder of this introduction explains each of these steps in more detail.
None of these steps is actually done for full Curry. Instead, the object of study will be a simplified version called CuMin (Curry Minor), which is presented in section 4.1. It still represents the way nondeterminism, failure and sharing interact in Curry, but some other features are exempt from the discussion. Section 4.1 contains details regarding these simplifications. The difference will be brushed over for the remainder of this introduction.

Defining a Functional-Style Denotational Semantics

There are different types of semantics for Curry. The Curry Report [Hanus, 2006] uses a lazy operational semantics by Albert et al. [2005]. The other established semantics [González-Moreno et al., 1999] is based on re-writing and is more closely associated with the functional-logic language TOY [López-Fraguas and Sánchez-Hernández, 1999]. Curry and TOY are closely related, which is witnessed by the fact that both semantics are equivalent [López-Fraguas et al., 2007].
The motivation for introducing yet another type of semantics is to come as close as possible to the functional paradigm. Therefore, Christiansen et al. [2011a] propose a denotational semantics inspired by [Søndergaard and Sestoft, 1992], that assigns a mathematical meaning to every expression in a compositional manner. The interpretations of types are directed-complete partial orders (dcpos), which is the usual choice for functional languages with general recursion [Abramsky and Jung, 1994]. Since Curry is nondeterministic, expressions have a set-valued semantics, so a power domain construction is used, i.e., a construction for domains analogous to power sets.
Being compositional means that the semantics of an expression is determined by the semantics of its subexpressions. In a language with general recursion, a function can appear as a subexpression in the function's defining rule. So, the semantics of the function cannot be defined in a way that assumes all subexpressions to already have a semantics, because this would lead to a cyclic dependency. Instead, the defining rules of the semantics have to be interpreted as a system of equations. To still have a well-defined semantics, this system of equations has to be interpreted in a framework that guarantees the existence of solutions and distinguishes one of these solutions. This extra requirement stops us from using plain sets as semantic domains for a deterministic

language with general recursion (like Haskell) and is the motivation for using dcpos in the first place [Abramsky and Jung, 1994].

The first major contribution in this chapter is a simplified denotational semantics for Curry, which also appeared in [Mehner et al., 2014]. It uses partially ordered sets (posets) instead of dcpos and thus avoids domain theory and in particular the somewhat contrived construction of power domains. The required extra structure – taking limits of elements – usually is provided by dcpos (see [Søndergaard and Sestoft, 1992] and [Abramsky and Jung, 1994]). In our setting, it is already provided on the level of sets by taking unions. Therefore, there is no need for limits on the element level and we can simply avoid the complication. This makes the element level conceptually easier because we can use partially ordered sets instead of dcpos. It also makes the set level easier because we do not have to make sure the extra structure is preserved. The construction is discussed in detail in section 4.3.

Relating the Different Semantics

The next step is to establish a connection between the functional-style denotational semantics and one of the existing semantics. The rewriting semantics [González-Moreno et al., 1999] and the lazy operational semantics [Albert et al., 2005] are equivalent [López-Fraguas et al., 2007], so taking one of them into account is enough. Christiansen et al. [2011a] decided that connecting their denotational semantics to the lazy operational semantics would be easier, because a similar statement is already shown in [Launchbury, 1993].

Launchbury [1993] compares two such semantics for a very simple, lazy, deterministic functional language. The result therein consists of two parts: correctness and adequacy. Correctness of the operational semantics with respect to the denotational semantics[1] means that evaluation preserves the denotation. More precisely, if the evaluation of an expression produces a value after some finite time, this result has the same denotation as the unevaluated expression. Adequacy means that evaluation succeeds exactly if the denotation of the expression is not failure. In that case, the result of the computation is automatically correct because we already know the evaluation to not change the denotation.

In our nondeterministic setting, these concepts have to be amended. The denotation of an expression will be a set to represent the possible multitude of different results. In the operational semantics, nondeterminism manifests in a different form: Sometimes there are different possible computations steps that can be done next. By choosing one or the other, the set of values can become smaller. So in our context, correctness will include the possibility of the set of results becoming smaller during the course of evaluation. Adequacy still means that for any non-failure result there is a terminating computation. Yet we now additionally require this computation to result in an expression that still has the given value in its denotation.

[1] The terminology suggests that the denotational semantics is given and the operational semantics is being studied. This is the case for Launchbury's paper, while here, the roles are reversed. We still stick to the terminology.

Christiansen et al. [2011a] claimed to have shown correctness, but later revoked that claim when one of the technical lemmas turned out to be incorrect. In this thesis, a slightly different approach to the correctness proof is used in section 4.4.

Christiansen et al. [2011a] also gave a blueprint for proving adequacy, but this blueprint turns out to be even more questionable. As in [Launchbury, 1993], the idea is to introduce a step-indexed version of the denotational semantics, i.e., a denotational semantics depending on a time parameter. If the number of steps remains unrestricted, this corresponds to the original denotational semantics. Yet in the proof of the adequacy statement, the step-indexing allows to use induction because it provides upper bounds for the number of operational steps needed for evaluation.

The details of how to propagate indexes in [Christiansen et al., 2011a] are different from the version given in [Launchbury, 1993]. However, no justification for this amendment is given and the alteration might even have been unintentional. In any case, the proof for the adequacy statement does not seem to work out – at least not in any straightforward way – when using the step-indexing described in [Christiansen et al., 2011a]. While the denotational semantics has been simplified considerably in other respects in [Mehner et al., 2014], step-indexing is still done as in [Christiansen et al., 2011a].

Yet returning to the version of step-indexing from [Launchbury, 1993] does not solve the problem either. Therefore, a third way of distributing indexes is introduced, while sticking to the poset-based denotational semantics given in [Mehner et al., 2014] otherwise. Using this version of the denotational semantics, the adequacy part of the statement is finally proved in section 4.5.

The details in distributing step-indexes are ultimately irrelevant, because the denotational semantics can be defined without using step-indexes. Taking this viewpoint, the denotational semantics given here coincides with the denotational semantics in [Mehner et al., 2014].

Parametricity and Deriving Free Theorems

Having convinced ourselves that the denotational semantics is a faithful representation of the way the language is supposed to work, we can proceed and put it to use. This development follows [Mehner et al., 2014], though the initial situation is slightly altered because of the changes in (the details of) the denotational semantics.

What we have to do is prove parametricity, the technical foundation for free theorems. Parametricity goes back to Reynolds [1983] and relates interpretations of one expression in different environments. The connection is established by a so called *logical relation* between the two interpretations. Free theorems [Wadler, 1989] are then derived by instantiating this relation to the denotation of a function in the language itself. Thus, instead of two denotations being related, we get a simpler statement about two different expressions having the same semantics (given some preconditions hold).

Parametricity does hold for Curry, as we will show, but deriving free theorems from parametricity is problematic. The catch is in instantiating the logical relation to the semantics of a syntactic function. Like in Haskell, not every syntactic function produces a suitable logical relation, but unlike in Haskell, the condition is hard to formalize.

Christiansen et al. [2010] already pointed out that nondeterminism has to be restricted in this function (only so called *multi-deterministic* functions are apt).

Curry's type system (by design) does not discern this property, so reasoning on the basis of types alone is impossible. On the semantic level we have a clear understanding of what 'deterministic' means, but we would rather omit having to descend this far down. The semantics not only makes nondeterminism explicit, but also the handling of variable environments and possibly even step-indexing.

The solution is to introduce an intermediate language called *Sets and Lambda Terms* (SaLT) in which nondeterminism is made explicit by using sets as a syntactic construct. These set types have a monadic interface, thus nondeterminism as an effect is marked in the type. SaLT is given a denotational semantics and Curry can be translated into SaLT by a purely syntax-directed translation procedure, that preserves this denotational semantics. In SaLT, there is a notion of being deterministic, which is being semantically equivalent to a singleton set. This does of course still rely on the semantics, but when arguing about programs we can use equational reasoning and do not have to actually compute the semantics.

The language SaLT, its denotational semantics, how to equationally argue about SaLT, and the translation procedure are discussed in section 4.6. The language itself and the translation have been slightly altered as compared to [Mehner et al., 2014] to make dealing with functions easier.

Braßel et al. [2011] also present a translation from Curry to a deterministic language, which is Haskell in their case. In both approaches, nondeterminism is made explicit using a monadic interface. However, the motivation behind the respective approaches is very different. Braßel et al. [2011] choose Haskell as the target language because they want the translated programs to be executable using the existing tools around Haskell. So, the language is fixed and the translation is designed to fit the language. The reason for introducing SaLT is that the translation can be simplified by amending the target language. Indeed, SaLT is not a fragment of Haskell, because Haskell distinguishes between failing monadic computations (*mzero*) and actual errors (*undefined*), while both are the same in SaLT. The price for changing the target language is that the existing infrastructure cannot be used any more. This is acceptable, as SaLT is intended to be a tool for reasoning about programs, rather than for running them. For the same reason, SaLT is not given an operational semantics.

Instead of proving parametricity for Curry, we prove it for SaLT (section 4.7). Thereby, the statement still is shown for all Curry expressions because of the semantics preserving translation. Also having it available for the SaLT expressions that do not arise as translations of CuMin expressions gives us additional degrees of freedom.

Relational parametricity for a language with monadic effects is unsurprising, as there is a general framework by Møgelberg and Simpson [2009]. Nondeterminism is indeed discussed as a special case of this framework, but in a simpler setting with no logic variables and no general recursion. It is reasonable to assume, that relational parametricity for SaLT arises as a special case of the general framework (or a further generalization

of it). However, since we are only interested in one special case, SaLT, we simply give a tailored version of the parametricity theorem with a tailored proof.
Here, one contribution is finding a suitable extension of the logical relation to our particular power set construction. Also, we generalize the statement (as compared to [Mehner et al., 2014]) to an inequational style as in [Johann and Voigtländer, 2006]. We deem this extension especially useful in the context of a nondeterministic language, where the inequalities are statements about set inclusion rather than about partial values.

Finally, all the tools are combined to prove free theorems. In particular, the side conditions from [Christiansen et al., 2010] are formalized. The overall strategy is to translate the claim and all the side conditions into SaLT. There, equational reasoning and SaLT's own free theorems can be used. The method is presented in section 4.8 by deriving some example free theorems that also appeared in [Mehner et al., 2014].
A further application is short cut fusion [Gill et al., 1993] for Curry, which is proved in section 4.9. Using the translation to SaLT, the claim is reduced to monadic short cut fusion [Ghani and Johann, 2007]. The side conditions are surprisingly weak, in that nondeterminism remains possible almost everywhere. The remaining restriction only excludes uncommon uses of nondeterminism and is easily checkable.

4.1 The Language CuMin

Curry [Hanus, 2013] is a general-purpose programming language and incorporates a variety of features. This expressiveness is convenient for the programmer, but less so when reasoning formally about the language. We will therefore be making use of a language fragment called *CuMin* (short for *Curry Minor* or *Curry Minus*) introduced in [Mehner et al., 2014]. It is a simplified version of Curry that faithfully represents the way lazyness and nondeterminism interact. Before we give a formal specification of the language, we review some features of Curry and explain why CuMin does or does not share them.
Another language to take into account here is FlatCurry, an internal representation for Curry used by various tools. A lot of syntactic features of Curry are redundant and are removed when translating to FlatCurry. The required transformations are described in detail in the Curry Report [Hanus, 2006] and can also be used for the transition from Curry to CuMin. FlatCurry however is an untyped language and type checking is assumed to happen on the level of full Curry.
Not all differences between Curry and CuMin fall into this category. There are some aspects in which both languages actually differ and those deserve special attention.
Finally, there are some tools around the language CuMin. Zaiser [2015] has implemented[2] the operational semantics and the translation into SaLT[3]. The denotational semantics has been implemented[4] by Thorand [2015].

[2] `github.com/fanzier/cumin-operational`

[3] `github.com/fanzier/cumin2salt`

[4] `github.com/fatho/bachelor-thesis`

4.1.1 Actual Simplifications

Recursive let-Bindings

Local variable definitions can be recursive in Curry, but CuMin will not allow this. For example, the binding $\mathbf{let}\ xs = [\,]\ ?\ [length\ xs]\ \mathbf{in} \ldots$ is fine in Curry. According to the operational semantics, $[\,]$ and $[1]$ are possible values while $[0]$ is not. This is due to the evaluation strategy: Once the right alternative is chosen, i.e., xs is bound to $[length\ xs]$, the length of xs can only be one.

This is impossible to model in a completely compositional denotational semantics, as showed in [Christiansen et al., 2011b]. There, the issue is discussed at length and the claims from [Christiansen et al., 2011a] are weakened to exclude recursive let bindings. In this thesis, recursive let bindings are excluded from the language already.

Encapsulated Search

Another feature we disregard completely in this thesis is *encapsulated search.* A nondeterministic computation performed within a capsule does not make the overall computation branch. Instead, the possible results are collected in a data structure, which can be processed as a whole. In particular, encapsulated search can be used to implement *negation by failure*: The absence of solutions to a search problem can be observed within the language itself by performing the search in a capsule. Without the capsule, the failing computation would cause the overall program to fail.

If no encapsulated search is possible, the language is monotone in the following sense: If a subexpression in a program is replaced by failure, the set of possible solutions can only become smaller. Encapsulated search breaks this monotonicity, since failure in a negated capsule leads to success on the outside.

There is a denotational semantics for encapsulated search in [Christiansen et al., 2013], which extends the denotational semantics given in [Christiansen et al., 2011a]. The missing monotonicity is mirrored on the semantic level, which makes the semantics less well-behaved. It remains unclear whether the developments presented in this thesis can be generalized to such a non-monotonous setting. Doing so might be possible but presumably not without a considerable amount of additional effort.

4.1.2 The Data Typeclass

In Curry, logic variables can be of any type, including function types, while in CuMin, logic variables can only have certain types collected in a typeclass [Wadler and Blott, 1989] Data. This typeclass contains base types such as Bool and Nat, as well as lists and tuples if their entry types are themselves data types. In earlier accounts, including [Mehner et al., 2014], two different versions of the $\forall$ quantifier were used to distinguish constrained type variables from arbitrary type variables.

The reason for this deviation from Curry lies in the free theorems [Wadler, 1989] we want to establish. For example we would like all functions f of type $\forall a.[a] \to [a]$ to satisfy $map\ g \circ f = f \circ map\ g$ for all $g :: \mathsf{A} \to \mathsf{B}$. Even in Haskell $g\ undefined = undefined$

$$\begin{array}{l}
\mathit{guard} :: \forall a.\mathsf{Bool} \to a \to a \\
\mathit{guard}\ b\ x = \textbf{case}\ b\ \textbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to \textbf{failed}_{\,a}\} \\[4pt]
\mathit{minus} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathit{minus}\ n\ m = \textbf{let}\ k :: \mathsf{Nat}\ \textbf{free in}\ \mathit{guard}_{\,\mathsf{Nat}}\ (m + k == n)\ k \\[4pt]
\mathit{choice} :: \forall a.a \to a \to a \\
\mathit{choice}\ x\ y = \textbf{let}\ b :: \mathsf{Bool}\ \textbf{free in case}\ b\ \textbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to y\} \\[4pt]
\mathit{coin} :: \mathsf{Nat} \\
\mathit{coin} = \mathit{choice}_{\,\mathsf{Nat}}\ 0\ 1 \\[4pt]
\mathit{double} :: \mathsf{Nat} \to \mathsf{Nat} \\
\mathit{double}\ n = n + n \\[4pt]
\mathit{suffix} :: \forall a.[\,a\,] \to [\,a\,] \\
\mathit{suffix}\ l = \mathit{choice}_{\,[a]}\ l\ (\textbf{case}\ l\ \textbf{of}\ \{\mathsf{Nil} \to \textbf{failed}_{\,[a]}\,; \mathsf{Cons}\ a\ as \to \mathit{suffix}_{\,a}\ as\}) \\[4pt]
\mathit{reverse} :: \forall a.[\,a\,] \to [\,a\,] \\
\mathit{reverse}\ xs = \mathit{reverseAppend}_{\,a}\ xs\ \mathsf{Nil}_{\,a} \\[4pt]
\mathit{reverseAppend} :: \forall a.[\,a\,] \to [\,a\,] \to [\,a\,] \\
\mathit{reverseAppend}\ x\ y = \textbf{case}\ x\ \textbf{of} \\
\quad \mathsf{Nil} \qquad\quad\ \to y \\
\quad \mathsf{Cons}\ a\ as \to \mathit{reverseAppend}_{\,a}\ as\ (\mathsf{Cons}_{\,a}\ a\ y)
\end{array}$$

Figure 4.1: Some example functions in CuMin

has to be required because of functions like $f\ \_ = \mathit{undefined} : [\,]$. In Curry, yet another possible function is $f\ \_ = \textbf{let}\ x\ \textbf{free in}\ x : [\,]$, so in order for the free theorem to still hold, g has to be surjective in some sense. A function being surjective is virtually never the case and very hard to check automatically. Thus, having such a restrictive condition would render the whole theory almost useless.
Restricting the use of logic variables to a typeclass offers a rather elegant means of further distinction. The function $f\ \_ = \textbf{let}\ x\ \textbf{free in}\ x : [\,]$ uses a logic variable whose type is the type argument, thus the typeclass constraint $\mathsf{Data}\ a$ is necessary. Thus, the free theorem only holds for surjective g or has to be weakened in some other manner, for example by only giving an inequational statement. On the other hand, the more restrictive type $[\,a\,] \to [\,a\,]$ without the class constraint excludes functions f using logic variables of type a. For all remaining functions f, the statement is true without requiring g to be surjective. This does not confine f to deterministic functions – it can still use logic variables of types not containing the type variable a (like *suffix* does in figure 4.1).

So we find ourselves in a situation where our results crucially depend on a feature that standard Curry does not offer. At the same time – at least from a Haskell point of view – the absence of typeclasses is strange: In Curry, the equality test can be applied to any two expressions having the same type, including function types. Yet running the test for function-typed expressions immediately produces an error because there is no implementation. Other identifiers, like (+), rely on a typeclass in Haskell but are monomorphic in Curry. Addition of floating point numbers is denoted (+.) to avoid the name clash with integer addition.

So the idea of having typeclasses in Curry is quite natural. For example, the Zinc compiler is an extension of the Münster Curry Compiler that features them.[5] Böhm [2013] describes how to include a typeclass mechanism into KiCS2 and an implementation is being worked on at the moment.

Both of the mentioned projects have predefined typeclasses known from Haskell like Eq, Num or Show, which allow to overload common symbols like (==) or (+) or the *show* function. This avoids the mentioned quirks of comparing functions and adding floats. Also the user is allowed to define new typeclasses and create instances.

Yet neither project introduces a Data typeclass restricting the use of logic features. Since such a typeclass can only be predefined, logic features remain unrestricted. So the gap between theory and practice is partially closed by switching to one of the existing extensions of Curry that do feature typeclasses. At the same time the absence of the one typeclass we actually care about does leave an (albeit smaller) remaining gap.

One argument for introducing a Data typeclass is that it makes the results of this thesis applicable. This argument is not entirely convincing, as – following the same logic – banning nondeterminism altogether makes even more results applicable, i.e., everything we know about free theorems in Haskell. Yet there are other reasons beyond the scope of this thesis for restricting the types of logic variables.

Constraint equality (=:=) means strict equality, i.e., the constraint of two terms being equal only is satisfied if they can be reduced to equal ground terms. For example, an infinite list is not 'strictly equal' to itself, because it cannot be reduced to a ground term. Therefore a constraint cannot be satisfied by simply unifying a logic variable with a given list, but totality has to be checked.

For function types on the other hand, the meaning of strict equality is not clear. One possibility would be to compare thunks, i.e., unevaluated expressions. If they do coincide, both functions indeed are exchangeable. However the functions could still behave the same even though their thunks are different. Extensional equality on the other hand, i.e., producing the same (set of) results for all inputs, cannot be checked in general.

Also, data types have generator functions [Antoy and Hanus, 2006] producing all possible values of that type. Thus, even if there are no clues allowing to find possible values for a logic variable in a targeted manner, trial-and-error remains as a less targeted but possible strategy. This does not work for functions either: Even if unification is permitted to bind logic variables in principle, nothing can be done in the absence of suitable constraints.

For data types, the meaning of strict equality is clear and all possible values can be generated. Thus, restricting these functionalities to certain types is the cleaner solution.

[5] http://zinc-project.sourceforge.net

Restricting logic features this way is a major change at the heart of the language. Implementing this quite possibly is a lot of work and breaks existing code. Yet, in order to still use the results presented here, workarounds are conceivable like using a special kind of annotation instead of typeclasses.

4.1.3 Features Orthogonal to Nondeterminism

User-Defined Data Types

The most important difference between Curry and CuMin in this category is the absence of user-defined data types. If we were to use such, we would need to keep track of the type constructor names as well as the corresponding value constructor names. Thus, all statements on the meta-level would have to be abstracted over type constructors, value constructors and arity, leading to a vast number of indexes. This effort would not lead to any additional insight, so the discussion is restricted to a preselected set of data type constructors. All of the following would still work if user defined data types were used.

Type Signatures and Instantiation

Curry, like Haskell, not only checks but also infers types. If a polymorphic function is invoked, its type has to be inferred from the given arguments or from the type of the expected result. Also type signatures can often be omitted, if the type is clear from the definition or the way the function is used.

In CuMin, we want to be able to determine the type of every expression bottom-up. Thus, whenever a logic variable is introduced, its type has to be given. All function definitions must have a type signature, so that a type can be assigned to invocations of the function. Also all type variables have to be instantiated explicitly when calling a function or using a constructor symbol.

There is no deep reason behind the different approach to typing – it simply avoids type inference, which would complicate the representation otherwise. It is of course possible to use type inference algorithms to generate type signatures or fill in missing ones. Indeed, we will sometimes be sloppy in our examples and rely on the reader to infer types.

Example code is shown in figure 4.1: In the *minus* function the type of k is explicitly given and instantiations of polymorphic functions are denoted as indexes. Note that polymorphic constructors like Cons are instantiated when appearing in an expression, but not when appearing in a pattern (like in *suffix*).

Input/Output

CuMin does not provide an IO wrapper for computations interacting with the environment. Such computations cannot be nondeterministic anyhow, because the world state cannot be duplicated. So in Curry there is no interesting interaction between the IO monad and nondeterminism either. Since there is nothing to be studied, there is no reason to include IO into CuMin.

On the other hand, there is a reason not to: If we did include input/output capabilities, we would have to make sure they indeed do not clash with nondeterminism. In Curry the easiest way to do so is to only use nondeterminism in a capsule and to only use input/output outside any capsule. This however is a design choice to avoid run-time errors and not enforced by the type system.

4.1.4 Redundant Features of Curry

Multiple Rules and Pattern Matching

A Curry function is defined by one or more rules, each of which contains (possibly nested) patterns. The order in which arguments to a function are evaluated can depend on all the patterns of all the rules of the function. In the operational semantics given in the Curry report [Hanus, 2006], this is captured in so called *definitional trees*. Rather than specifying directly what the semantics of a function is, the report gives an algorithm for transforming a function's rules into a definitional tree and defines the semantics of such trees.
To avoid this indirection the syntax of function definitions is restricted (like it is in FlatCurry): In CuMin every function has to be given by a single rule $f\ x_1 \dots x_n = \dots$ where the x_i are distinct variables. Thus, neither pattern matching nor guards are possible on the rule level.
Pattern matching can only be done in the function's body using **case**...**of** $\{\dots\}$. Additionally, only flat patterns may be used, i.e., of the form $\mathsf{C}\ y_1 \dots y_n$. Thus, the order in which arguments are evaluated is captured on the syntactic level already and does not have to be generated in a preprocessing step.

Binary Choice

In Curry there are three sources of nondeterminism: overlapping rules, binary choice and logic variables. We have already discussed why CuMin does not have overlapping rules. There is no binary choice primitive either, because binary choice can be implemented by pattern matching on a Boolean logic variable (see the *choice* function in figure 4.1).

Local Definitions

In Curry, local definitions can be made using **let**...**in**... and ...**where**.... Definitions made in a **where** clause are also visible in guards. Since rules cannot have guards in CuMin, there is no need for **where** clauses and CuMin only features **let**.
In CuMin and FlatCurry, functions can only be defined on the top-level. All sections, anonymous functions and local functions have to be moved to the top-level and receive unique names. For example the function *reverseAppend* in figure 4.1 might have well been local to *reverse* in Curry, as this is the only function calling *reverseAppend* (other than *reverseAppend* itself). If a local function depends on local variables, they have to be passed as arguments when moving the function to the top-level and the call has to be amended accordingly.

$$\begin{aligned}
P &::= D; P \mid D \\
D &::= f :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau; f\ \overline{x_n} = e \\
\tau &::= \alpha \mid \tau \to \tau' \mid \mathsf{Bool} \mid \mathsf{Nat} \mid [\tau] \mid (\tau, \tau') \\
e &::= x \mid \mathbf{let}\ \overline{x_n = e_n}\ \mathbf{in}\ e \mid \mathbf{let}\ \overline{x_n :: \tau_n}\ \mathbf{free\ in}\ e \\
&\quad \mid \mathbf{failed}_{\tau} \mid f_{\overline{\tau_m}} \mid e_1\ e_2 \\
&\quad \mid n \mid e_1 + e_2 \mid e_1 == e_2 \\
&\quad \mid \mathsf{True} \mid \mathsf{False} \mid \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{True} \to e_1; \mathsf{False} \to e_2\} \\
&\quad \mid \mathsf{Nil}_{\tau} \mid \mathsf{Cons}_{\tau} \mid \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\} \\
&\quad \mid \mathsf{Pair}_{\tau,\tau'} \mid \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Pair}\ l\ r \to e_1\}
\end{aligned}$$

Figure 4.2: Syntax of CuMin

CuMin also does not feature modules (unlike FlatCurry), so all modules have to be compiled into one monolithic program (again possibly renaming functions).
In Curry, several local variables can be introduced at the same time by binding an expression to a pattern using **let**. Both in FlatCurry and CuMin, only the trivial pattern, i.e., a single variable, is allowed as the left hand side of a local binding. Thus, to achieve the same behavior, the whole expression has to be bound to a name first and then parts can be accessed using **case** . . . **of**.

4.1.5 Formal Specification of CuMin

Syntax

The syntax of CuMin is formally defined in figure 4.2, where $\overline{x_n}$ and $\overline{\alpha_m}$ represent zero or more variable names and $\overline{\mathsf{Data}\ \alpha_{i_j}}$ and $\overline{\tau_m}$ a comma-separated list of zero or more class constraints resp. types.
A *program* is a sequence of *(function) definitions*. Every function definition consists of a *type signature* and a *rule*. In figure 4.2 these are separated by semicolons, but we take the liberty to use newlines when giving code examples instead. A type signature gives the function's name and a *type scheme*. The type scheme consists of a sequence of distinct type variable names, a sequence of typeclass constraints and a type. A typeclass constraint can mark one of the type variables as representing a data type. If the type signature does not mention any typeclass constraints, the () ⇒ may be dropped and if there are no type variables, the ∀. may be dropped as well. The function type constructor → associates to the right, but parenthesis can be used to construct higher order types.
A rule has a left hand side consisting of the function name and distinct variables and a body, which is some *expression*. The number of variables on the left hand side of a rule is called the *arity* of the function.
Pattern matching can only be done using **case** . . . **of**, which is also used for Booleans (instead of **if** . . . **then** . . . **else** . . .) to make the syntax more homogeneous. Instead of braces and semicolons, the alternatives may be given in (properly indented) separate lines.

Juxtaposition represents application, associates to the left and binds most tightly. Addition also associates to the left (though this will hardly ever matter) and has intermediate binding precedence. The equality test == does not associate and binds least tightly. Parenthesis may be used to structure expressions.
When calling polymorphic functions or the **failed** primitive, they have to be instantiated by giving a comma-separated sequence of types. Polymorphic constructors also need to be instantiated when called, but not when they appear in patterns. The type variables given in the signature are also in scope throughout the rule, such that the calling function can pass on its type variables to the called function. Functions can call themselves and each other arbitrarily and may only be defined on the top-level.
Using **let** . . . **in** . . . expressions can be bound to variable names locally or be marked as logic variables. These variable bindings may not be cyclic, e.g., a variable x cannot appear in the expression the variable itself is bound to.
There is also an ASCII-representation of code, which is mostly straightforward from the definition given here. The $\forall$ symbol is written as `forall` (and *forall* can not be used as a type variable name). Type instantiations for functions, constructors and primitives are enclosed in `<:` and `:>` instead of appearing as subscripts.

Typing

The typing rules for CuMin use four kinds of judgments:

Γ context	"Γ is a context"
$\Gamma \vdash \tau \in \mathsf{Type}$	"τ is a type within Γ"
$\Gamma \vdash \tau \in \mathsf{Data}$	"τ is a data type within Γ"
$\Gamma \vdash e :: \tau$	"e has type τ within Γ"

Figure 4.3 shows how the first two auxiliary kinds of judgments are derived. Contexts are unordered and can contain (distinct) type variables, type variable constraints and (distinct) term variables with their respective types. Being a type within a context means that only type variables mentioned in the context can appear in the type. The third kind of judgment describes the class of data types and is derived by the rules in Figure 4.4. Both Bool and Nat are data types, list types are data types if the entry type is a data type and likewise for tuples.
The actual typing of expressions is done by the last kind of judgment. Formally this would have to also mention a program, as the type of a function invocation can only be seen from the function's signature. We suppress this additional dependency and rather assume a program to be fixed all the time, with respect to which all expressions are typed. Figure 4.5 shows the rules for doing so, where $[\tau/\alpha]$ denotes substitution of the variable α by τ.

We distinguish between *being a function* and *having a function type*. Something *is a function* if it is defined by a rule on the top level, regardless of whether it has a function type and regardless of its arity. For example *coin* as given in figure 4.1 is a

$$\emptyset \text{ context}$$

$$\frac{\Gamma \text{ context}}{\Gamma, \alpha \text{ context}} \qquad \frac{\Gamma, \alpha \text{ context}}{\Gamma, \alpha, \alpha \in \mathsf{Data} \text{ context}} \qquad \frac{\Gamma \text{ context} \qquad \Gamma \vdash \tau \in \mathsf{Type}}{\Gamma, x :: \tau \text{ context}}$$

$$\Gamma, \alpha \vdash \alpha \in \mathsf{Type} \qquad \Gamma \vdash \mathsf{Bool} \in \mathsf{Type} \qquad \Gamma \vdash \mathsf{Nat} \in \mathsf{Type}$$

$$\frac{\Gamma \vdash \tau \in \mathsf{Type} \qquad \Gamma \vdash \tau' \in \mathsf{Type}}{\Gamma \vdash \tau \to \tau' \in \mathsf{Type}} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Type}}{\Gamma \vdash [\tau] \in \mathsf{Type}}$$

$$\frac{\Gamma \vdash \tau \in \mathsf{Type} \qquad \Gamma \vdash \tau' \in \mathsf{Type}}{\Gamma \vdash (\tau, \tau') \in \mathsf{Type}}$$

Figure 4.3: Rules for auxiliary typing judgments in CuMin

$$\Gamma, \alpha, \alpha \in \mathsf{Data} \vdash \alpha \in \mathsf{Data} \qquad \Gamma \vdash \mathsf{Bool} \in \mathsf{Data} \qquad \Gamma \vdash \mathsf{Nat} \in \mathsf{Data}$$

$$\frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash [\tau] \in \mathsf{Data}} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Data} \qquad \Gamma \vdash \tau' \in \mathsf{Data}}{\Gamma \vdash (\tau, \tau') \in \mathsf{Data}}$$

Figure 4.4: Rules for being a data type in CuMin

function. Expressions and variables can have a function type but do not have an arity. Constructors on the other hand do have an arity, which is the number of fields.
A program is well-typed if all function rules match the claimed type signatures. Formally, this means for all n-ary function definitions

$$f :: \forall \overline{\alpha_m}.(\mathsf{Data}\ \alpha_{i_1}, .., \mathsf{Data}\ \alpha_{i_j}) \Rightarrow \tau; f\ \overline{x_n} = e$$

the type τ equals $\tau_1 \to .. \to \tau_n \to \tau'$, i.e., is comprised of (at least) n function arrows, and the typing judgment

$$\alpha_1, .., \alpha_m, \alpha_{i_1} \in \mathsf{Data}, .., \alpha_{i_j} \in \mathsf{Data}, x_1 :: \tau_1, .., x_n :: \tau_n \vdash e :: \tau'$$

can be derived.

4.2 Operational Semantics

4.2.1 Operational Semantics with Logic Variables

The operational semantics for CuMin presented here is an adaptation of the operational semantics given for Curry in [Albert et al., 2005] and later revised in [Braßel and Huch, 2007]. These in turn are adaptations of Launchbury's lazy operational semantics [Launchbury, 1993] to the nondeterministic setting. The only auxiliary data structure is a heap, in which unevaluated expressions can be stored.

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash \textbf{failed}_{\tau} :: \tau$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \qquad \ldots \qquad \Gamma, x_1 :: \tau_1, \ldots, x_{n-1} :: \tau_{n-1} \vdash e_n :: \tau_n \qquad \Gamma, x_1 :: \tau_1, \ldots, x_n :: \tau_n \vdash e :: \tau}{\Gamma \vdash \textbf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \textbf{in}\ e :: \tau}$$

$$\frac{\Gamma \vdash \tau_1 \in \mathsf{Data} \qquad \ldots \qquad \Gamma \vdash \tau_n \in \mathsf{Data} \qquad \Gamma, x_1 :: \tau_1, \ldots, x_n :: \tau_n \vdash e :: \tau}{\Gamma \vdash \textbf{let}\ x_1 :: \tau_1, \ldots, x_n :: \tau_n\ \textbf{free in}\ e :: \tau}$$

$$\frac{\Gamma \vdash \tau_{i_1} \in \mathsf{Data} \qquad .. \qquad \Gamma \vdash \tau_{i_j} \in \mathsf{Data}}{\Gamma \vdash f_{\,\overline{\tau_m}} :: \tau\ [\overline{\tau_m/\alpha_m}]}$$

for every function $(f :: \forall\overline{\alpha_m}.(\mathsf{Data}\ \alpha_{i_1}, .., \mathsf{Data}\ \alpha_{i_j}) \Rightarrow \tau; f\ \overline{x_n} = e)$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}$$

$$\Gamma \vdash \mathsf{True} :: \mathsf{Bool} \qquad \Gamma \vdash \mathsf{False} :: \mathsf{Bool} \qquad \Gamma \vdash n :: \mathsf{Nat}$$

$$\Gamma \vdash \mathsf{Nil}_{\tau} :: [\tau] \qquad \Gamma \vdash \mathsf{Cons}_{\tau} :: \tau \to [\tau] \to [\tau] \qquad \Gamma \vdash \mathsf{Pair}_{\tau,\tau'} :: \tau \to \tau' \to (\tau, \tau')$$

$$\frac{\Gamma \vdash e_1 :: \mathsf{Nat} \qquad \Gamma \vdash e_2 :: \mathsf{Nat}}{\Gamma \vdash e_1 + e_2 :: \mathsf{Nat}} \qquad \frac{\Gamma \vdash e_1 :: \mathsf{Nat} \qquad \Gamma \vdash e_2 :: \mathsf{Nat}}{\Gamma \vdash e_1 == e_2 :: \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e :: \mathsf{Bool} \qquad \Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\mathsf{True} \to e_1; \mathsf{False} \to e_2\} :: \tau}$$

$$\frac{\Gamma \vdash e :: [\tau'] \qquad \Gamma \vdash e_1 :: \tau \qquad \Gamma, h :: \tau', t :: [\tau'] \vdash e_2 :: \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\} :: \tau}$$

$$\frac{\Gamma \vdash e :: (\tau_1, \tau_2) \qquad \Gamma, l :: \tau_1, r :: \tau_2 \vdash e_1 :: \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Pair}\ l\ r \to e_1\} :: \tau}$$

Figure 4.5: Typing rules for CuMin

Deviating from the above sources, heaps are ordered in the operational semantics defined here. This is only possible because there are no recursive let bindings in CuMin. The operational semantics would work equally well with unordered heaps, but the proofs in sections 4.4 and 4.5 become easier when using ordered heaps. The restriction of not allowing recursive lets is inherent to the overall approach of using a compositional, functional-style denotational semantics [Christiansen et al., 2011b]. Since the results we are going to prove do not generalize to a setting in which recursive lets are allowed, the proofs do not have to generalize either.

Another deviation from [Braßel and Huch, 2007] is the missing preprocessing step of normalization. Instead, we will apply the necessary changes on the go as in [Hanus and Peemöller, 2014], where a (Flatten) rule is added to the semantics.

In [Braßel and Huch, 2007], logic variables in the heap are denoted by circular bindings $y \mapsto y$. Since in the vast majority of cases these special bindings have to be treated separately anyhow, we use a special notation for logic variables right away. Heap entries thus take one of the forms $y :: \tau \mapsto e$ or $y :: \tau$ **free**. The former binds a variable y of some closed type τ to an expression e and the latter only gives a closed type τ and marks y as a logic variable. Heaps can be given explicitly as a (possibly empty) sequence of comma separated bindings enclosed in brackets. Heaps can also be combined into bigger heaps by juxtaposition. We assume all our heaps to be well-formed, i.e., :

- All variables are distinct.
- If y is bound to an expression, this expression only contains the variables appearing to the left of y in the heap and is typeable to the given type (in the context given by the types of the heap variables).
- If y is introduced as a logic variable, its type is in the Data class.

Within the operational semantics we encounter a second kind of function application $\varphi(y_1, \ldots, y_k)$ where φ is a function or a constructor symbol (with type instantiation) of arity at least k and $y_1, \ldots, y_k$ are variables. For the type system this is equivalent to φ $y_1 \ldots y_k$ but the operational semantics handels them differently, as we will soon see. We identify $f_{\overline{\tau_m}}$ with $f_{\overline{\tau_m}}()$ and $\mathsf{C}_{\overline{\tau_m}}$ with $\mathsf{C}_{\overline{\tau_m}}()$.

A *flat normal form* is a partial or full application $\mathsf{C}_{\overline{\tau_m}}(y_1, \ldots, y_k)$ of a constructor symbol to variables or a partial application $f_{\overline{\tau_m}}(y_1, \ldots, y_k)$ of a function symbol to variables. Here *partial application* requires the number of variables to be smaller than the arity of the function. For example, if $id : \forall a.a \to a$ is given as a unary function, the application $id_{\mathsf{Nat} \to \mathsf{Nat}}(g)$ is not partial, even though it has a function type. On the other hand, partially applied functions and constructors necessarily have a function type and indeed are the flat normal forms for function types. This leaves fully applied constructors, which are the flat normal forms of all other (i.e., non-function) types. Here we consider literals as nullary (and therefore necessarily fully applied) constructors.

A *heap expressions pair* $\Delta : e$ is a heap Δ together with an expression e that is typeable in the context given by all heap variables (and their respective types). A *value* (w.r.t. a heap) is a flat normal form or a logic variable. Thus a function-typed value has to be a partial application of a function or a constructor to variables while for all other types a value is a fully applied constructor or a logic variable.

The operational semantics derives judgments of the form $\Delta : e \Downarrow^f \Delta' : v$ and $\Delta : e \Downarrow^l \Delta' : v$, meaning e under the heap Δ can evaluate to v under the heap Δ'. Because the language is nondeterministic, given $\Delta : e$ there can be many different pairs $\Delta' : v$ for which such judgments are derivable or none at all. The first judgment guarantees that v is a flat normal form and we say $\Delta : e$ evaluates to $\Delta' : v$ *functionally.* The second judgment guarantees that v is a value (w.r.t. Δ') and we say $\Delta : e$ evaluates to $\Delta' : v$ *logically.*

A representative subset of the rules is given in figure 4.6. The (Val) rule can be used whenever the expression to be evaluated is a value already. As this includes logic variables, (Lookup) is only needed for variables bound to an expression within the heap. This expression is then evaluated and the result is both written into the heap and returned as the result of the lookup. Thus all subsequent lookups allow to immediately use rule (Val) for already evaluated expressions. We do not need black hole detection when looking up variables as in [Braßel and Huch, 2007] as no circular let-bindings are allowed.

The rules (Let) and (Free) allow to evaluate **let** . . . **in** bindings. They add variables to the heap, either binding them to expressions or marking them as logic variables. In any case, evaluation is deferred until the variable is needed.

There are two rules governing application. (Fun) can only be used if a function is fully applied and all arguments are heap variables. This rule invokes the function's body, in which the formal parameters (on type as well as term level) are replaced by the actual parameters. Note that this is the only possible occurance of the new application syntax that is not handled by (Val) already. General binary application is handled by (Apply), which incorporates the rule (Flatten) from [Hanus and Peemöller, 2014]. The left premise brings the function into a value form, which for type reasons is a partially applied constructor or function symbol $\varphi(y_1, \dots, y_k)$. A further variable x can be added without exceeding the arity, where x is bound to the (possibly complex) argument e_2. If e_2 is a variable already, it could be added to the arguments of φ without first binding it to a new name. Also instead of using **let** . . . **in** the rule could add the new variable to the heap right away, as this has to be the next step anyhow. Stating the rule in the given form helps our discussion later, so we leave the obvious simplifications to those concerned about runtime.

The (Plus) rule is exemplary for calls to primitive (or external) functions, that do not accept logic variables as input. Therefore both arguments have to be evaluated to a flat normal form, which in the case of (Plus) has to be a literal.

The (Case) rules also require functional evaluation for the scrutinee, while the body is allowed to evaluate to a logic variable. There are of course similar rules for all constructors, which work analogously to the rules shown. Since all of them require evaluating the scrutinee first, an interpreter may do so and then decide afterwards *which* (Case) rule to apply.

The remaining rules derive judgments for functional evaluation. This can be done by the (FNF) rule if the value is in flat normal form already and by *guessing.* If the

$$(\textsc{Val}) \quad \Delta : v \Downarrow^{l} \Delta : v \qquad \text{if } v \text{ is a value w.r.t. } \Delta$$

$$(\textsc{Lookup}) \quad \frac{\Delta : e \Downarrow^{l} \Delta' : v}{\Delta\,[y :: \tau \mapsto e]\,\Omega : y \Downarrow^{l} \Delta'\,[y :: \tau \mapsto v]\,\Omega : v}$$

$$(\textsc{Let}) \quad \frac{\Delta\,[\overline{y_n :: \tau_n \mapsto e_n\,[\overline{y_n/x_n}]}] : e\,[\overline{y_n/x_n}] \Downarrow^{l} \Delta' : v}{\Delta : \textbf{let } \overline{x_n = e_n} \textbf{ in } e \Downarrow^{l} \Delta' : v} \qquad y_1, \ldots, y_n \text{ fresh}$$

$$(\textsc{Free}) \quad \frac{\Delta\,[\overline{y_n :: \tau_n \textbf{ free}}] : e\,[\overline{y_n/x_n}] \Downarrow^{l} \Delta' : v}{\Delta : \textbf{let } \overline{x_n :: \tau_n} \textbf{ free in } e \Downarrow^{l} \Delta' : v} \qquad y_1, \ldots, y_n \text{ fresh}$$

$$(\textsc{Fun}) \quad \frac{\Delta : e\,[\overline{\tau_m/\alpha_m}, \overline{y_n/x_n}] \Downarrow^{l} \Delta' : v}{\Delta : f_{\,\overline{\tau_m}}\,(y_1, \ldots, y_n) \Downarrow^{l} \Delta' : v} \qquad \text{for } f :: \forall \alpha_1 \ldots \alpha_m . \tau ; f\ \overline{x_n} = e \text{ in P}$$

$$(\textsc{Apply}) \quad \frac{\Delta : e_1 \Downarrow^{l} \Delta' : \varphi(y_1, \ldots, y_k) \qquad \Delta' : \textbf{let } x = e_2 \textbf{ in } \varphi(y_1, \ldots, y_k, x) \Downarrow^{l} \Delta'' : v}{\Delta : e_1\ e_2 \Downarrow^{l} \Delta'' : v}$$

$$(\textsc{Plus}) \quad \frac{\Delta : e_1 \Downarrow^{f} \Delta' : n_1 \qquad \Delta' : e_2 \Downarrow^{f} \Delta'' : n_2}{\Delta : e_1 + e_2 \Downarrow^{l} \Delta'' : n} \qquad \text{for literals } n_1,\ n_2 \text{ and } n = n_1 + n_2$$

$$(\textsc{Case}_{\mathsf{Nil}}) \quad \frac{\Delta : e \Downarrow^{f} \Delta' : \mathsf{Nil}_{\,\tau} \qquad \Delta' : e_1 \Downarrow^{l} \Delta'' : v}{\Delta : \textbf{case } e \textbf{ of } \{\mathsf{Nil} \to e_1 ; \mathsf{Cons}\ h\ t \to e_2\} \Downarrow^{l} \Delta'' : v}$$

$$(\textsc{Case}_{\mathsf{Cons}}) \quad \frac{\Delta : e \Downarrow^{f} \Delta' : \mathsf{Cons}_{\,\tau}\,(y_1, y_2) \qquad \Delta' : e_2\,[y_1/h, y_2/t] \Downarrow^{l} \Delta'' : v}{\Delta : \textbf{case } e \textbf{ of } \{\mathsf{Nil} \to e_1 ; \mathsf{Cons}\ h\ t \to e_2\} \Downarrow^{l} \Delta'' : v}$$

$$(\text{FNF}) \quad \frac{\Delta : e \Downarrow^{l} \Delta' : v}{\Delta : e \Downarrow^{f} \Delta' : v} \qquad \text{where } v \text{ is in flat normal form}$$

$$(\textsc{Guess}_n) \quad \frac{\Delta : e \Downarrow^{l} \Delta'\,[y :: \mathsf{Nat} \textbf{ free}]\,\Omega : y}{\Delta : e \Downarrow^{f} \Delta'\,[y :: \mathsf{Nat} \mapsto n]\,\Omega : n} \qquad \text{for any literal } n$$

$$(\textsc{Guess}_{\mathsf{Nil}}) \quad \frac{\Delta : e \Downarrow^{l} \Delta'\,[y :: [\tau] \textbf{ free}]\,\Omega : y}{\Delta : e \Downarrow^{f} \Delta'\,[y :: [\tau] \mapsto \mathsf{Nil}_{\,\tau}]\,\Omega : \mathsf{Nil}_{\,\tau}}$$

$$(\textsc{Guess}_{\mathsf{Cons}}) \quad \frac{\Delta : e \Downarrow^{l} \Delta'\,[y :: [\tau] \textbf{ free}]\,\Omega : y}{\Delta : e \Downarrow^{f} \Delta'\,[y_1 :: \tau \textbf{ free}, y_2 :: [\tau] \textbf{ free}, y :: [\tau] \mapsto \mathsf{Cons}_{\,\tau}\,(y_1, y_2)]\,\Omega : \mathsf{Cons}_{\,\tau}\,(y_1, y_2)}$$

y_1, y_2 fresh

Figure 4.6: Operational semantics for CuMin

result of some computation is a logic variable one may simply pick any constructor of the respective type and introduce new logic variables for the fields. The resulting flat normal form is a possible value since logic variables represent all values of their type. There is a (Guess)-rule for every constructor and they work analogously to the ones shown in the figure. Which rule to use can be decided according to the result in the premise: If it is a flat normal form, (FNF) can be applied, otherwise, it is a logic variable and its type can be looked up in the heap.
There is no rule for **failed**$_\tau$, as it does not have any flat normal form.

In [Braßel and Huch, 2007], the distinction between functional and logic evaluation is not made. Instead, there are different rules (Select) and (Guess), both dealing with **case** statements: (Select) can be applied if evaluating the scrutinee yields a flat normal form, while (Guess) binds a logic variable if one results from evaluating the scrutinee. The (Select) rule corresponds to only (Case). The rule (Guess) in [Braßel and Huch, 2007] is split up into (Guess) and another application of (Case) here.
This is connected to another slight deviation from Curry and the semantics given in [Braßel and Huch, 2007]: Curry distinguishes between flexible and rigid case statements. The plain **case** syntax means rigid pattern matching – the scrutinee is evaluated and the corresponding case is picked. Flexible pattern matching, like pattern matching in function definitions, also works for logic variables. If the scrutinee of a flexible pattern match is a logic variable, it can be bound to the constructor given in the pattern. In the operational semantics, this is modeled by rule (Guess) only being applicable for flexible pattern matches.
This leaves the question of what to do when the scrutinee of a rigid pattern match yields a logic variable. In this case, evaluation according to the operational semantics in [Braßel and Huch, 2007] simply fails for the lack of any applicable rule. However, constraints in Curry are supposed to be evaluated in a concurrent manner as desribed in the Curry report [Hanus, 2006]. Thus the whole computation is not necessarily stuck, as there could be another constraint that is being solved in parallel. At some point, the variable might get bound to some constructor and the rigid pattern match can be performed.
We generally ignore the issue of parallel constraint solving here because the interaction between different constraints is difficult to model in a denotational semantics. Thus we do not need to distinguish between flexible and rigid case statements anymore and instead treat every case statement as flexible.

Figure 4.7 shows a derivation for *double coin* as defined in figure 4.1. This example is important because it shows how call-time-choice is modelled in the semantics. The term suggests that choice occurs at call-time to fix the values of all arguments before evaluating the function body. This is not actually the case, as the semantics is still lazy. In the example derivation in figure 4.7, *coin* is bound to the name y_1 and *double* is applied to y_1. Then the body of *double* is inlined *before* the body of *coin*. Choice occurs only when evaluation of the scrutinee b is forced. Later, when y_1 is looked up a second time, it is bound to the value 1 in the heap already, thus *coin* is not called again.

$$
\begin{array}{l}
[\,] : \mathit{double}\ \mathit{coin} \\
\left[\begin{array}{l}
[\,] : \mathit{double} \\
[\,] : \mathit{double}
\end{array}\right. \\
[\,] : \mathbf{let}\ x = \mathit{coin}\ \mathbf{in}\ \mathit{double}\ (x) \\
[y_1 :: \mathsf{Nat} \mapsto \mathit{coin}] : \mathit{double}\ (y_1) \\
[y_1 :: \mathsf{Nat} \mapsto \mathit{coin}] : y_1 + y_1 \\
\left[\begin{array}{l}
[y_1 :: \mathsf{Nat} \mapsto \mathit{coin}] : y_1 \\
\left[\begin{array}{l}
[\,] : \mathit{coin} \\
[\,] : \mathit{choice}_{\mathsf{Nat}}\ 0\ 1 \\
\left[\begin{array}{l}
[\,] : \mathit{choice}_{\mathsf{Nat}}\ 0 \\
\left[\begin{array}{l}
[\,] : \mathit{choice}_{\mathsf{Nat}} \\
[\,] : \mathit{choice}_{\mathsf{Nat}}
\end{array}\right. \\
[\,] : \mathbf{let}\ x = 0\ \mathbf{in}\ \mathit{choice}_{\mathsf{Nat}}\ (x) \\
[y_2 :: \mathsf{Nat} \mapsto 0] : \mathit{choice}_{\mathsf{Nat}}\ (y_2)
\end{array}\right. \\
[y_2 :: \mathsf{Nat} \mapsto 0] : \mathbf{let}\ x = 1\ \mathbf{in}\ \mathit{choice}_{\mathsf{Nat}}\ (y_2, x) \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1] : \mathit{choice}_{\mathsf{Nat}}\ (y_2, y_3) \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1] : \mathbf{let}\ b :: \mathsf{Bool}\ \mathbf{free\ in\ case}\ b\ \mathbf{of}\ \{\mathsf{True} \to y_2; \mathsf{False} \to y_3\} \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool}\ \mathbf{free}] : \mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to y_2; \mathsf{False} \to y_3\} \\
\left[\begin{array}{l}
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool}\ \mathbf{free}] : b \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}] : \mathsf{False}
\end{array}\right. \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}] : y_3 \\
\left[\begin{array}{l}
[\,] : 1 \\
[\,] : 1
\end{array}\right. \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}] : 1
\end{array}\right. \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}, y_1 :: \mathsf{Nat} \mapsto 1] : 1
\end{array}\right. \\
\left[\begin{array}{l}
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}, y_1 :: \mathsf{Nat} \mapsto 1] : y_1 \\
\left[\begin{array}{l}
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}] : 1 \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}] : 1
\end{array}\right. \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}, y_1 :: \mathsf{Nat} \mapsto 1] : 1
\end{array}\right. \\
[y_2 :: \mathsf{Nat} \mapsto 0, y_3 :: \mathsf{Nat} \mapsto 1, y_4 :: \mathsf{Bool} \mapsto \mathsf{False}, y_1 :: \mathsf{Nat} \mapsto 1] : 2
\end{array}
$$

Figure 4.7: Possible derivation for the *double coin* example

The following lemma makes sure the order of the heap (which we keep track of) is preserved during evaluation.

Lemma 4.2.1. If a judgment $\Delta : e \Downarrow^l \Delta' : v$ or $\Delta : e \Downarrow^f \Delta' : v$ can be derived by the operational semantics, then Δ' binds all variables bound in Δ and the orders coincide.

Proof. Induction over all derivations. □

4.2.2 Removing Logic Variables

We now present a version of the operational semantics that does not use logic variables. The same transition is made in [Braßel and Huch, 2007], but the idea goes back to Antoy and Hanus [2006]: Logic variables as a language feature are redundant in lazy functional-logic languages because they can be implemented using lazy evaluation. This is achieved by so called *generator functions*, which produce every value of some type using overlapping rules and recursive calls to themselves and other generator functions. Instead of marking a variable as free, it is bound to the generator function corresponding to the variable's type. Because of lazy evaluation, the variable is only instantiated as much as needed.
We introduce a special expression $\textbf{unknown}_\tau$, which serves as generator function and represents every possible value of some type τ whenever $\tau \in \mathsf{Data}$. Since this is the only remaining source of nondeterminism, it cannot be implemented using other language features, but needs special rules.
The rules are given in 4.8 and are mostly the same as in 4.6 (replacing both $\Downarrow^l$ and $\Downarrow^f$ by $\Downarrow$). The (Val) rule now requires v to be a flat normal form. As there are no logic variables, (Lookup) can be used whenever the expression to be evaluated is a variable. The rule (Free) introduces the special expression **unknown** by writing it into the heap. To be able to evaluate the expression **unknown** (when it has been looked up) different (Guess) rules are provided. As for the (Apply) rule, it would be possible to add the new variables to the heap directly, but we choose to use **let** ... **in** to limit the number of rules that manipulate the heap. All other rules remain unchanged (as compared to 4.6) except for the missing annotations on the derivation arrows.

The following lemma relates both versions of the operational semantics. In a nutshell, $\Downarrow$ corresponds to $\Downarrow^f$ and thus the latter version of the operational semantics allows to work with functional evaluation without having to refer to logic evaluation all the time.

Lemma 4.2.2. $\Delta : e \Downarrow^f \Delta' : v$ is derivable if and only if $\Delta^u : e \Downarrow \Delta'^u : v$ is derivable, where $e \neq \textbf{unknown}_\tau$ is some expression, v a flat normal form and Δ^u is Δ with all entries $[y :: \tau' \ \textbf{free}]$ replaced by $[y :: \tau' \mapsto \textbf{unknown}_{\tau'}]$ (and Δ'^u accordingly).

Proof. The proof is by manipulating derivation trees. We start with the "only if" part and fix some derivation tree for a $\Downarrow^f$ judgment.
Consider some heap variable y introduced by (Free). The initial heap entry $[y :: \tau \ \textbf{free}]$ has to be replaced by $[y :: \tau \mapsto \textbf{unknown}_\tau]$ because of the different (Free) rule. If y

$$(\textsc{Val})\quad \Delta : v \Downarrow \Delta : v \qquad\qquad \text{if } v \text{ is a flat normal form}$$

$$(\textsc{Lookup})\quad \frac{\Delta : e \Downarrow \Delta' : v}{\Delta\ [y :: \tau \mapsto e]\ \Omega : y \Downarrow \Delta'\ [y :: \tau \mapsto v]\ \Omega : v}$$

$$(\textsc{Let})\quad \frac{\Delta\ [\overline{y_n :: \tau_n \mapsto e_n\ [\overline{y_n/x_n}]}] : e\ [\overline{y_n/x_n}] \Downarrow \Delta' : v}{\Delta : \textbf{let}\ \overline{x_n = e_n}\ \textbf{in}\ e \Downarrow \Delta' : v} \qquad\qquad y_1, \ldots, y_n \text{ fresh}$$

$$(\textsc{Free})\quad \frac{\Delta\ [\overline{y_n :: \tau_n \mapsto \textbf{unknown}_{\tau_n}}] : e\ [\overline{y_n/x_n}] \Downarrow \Delta' : v}{\Delta : \textbf{let}\ \overline{x_n :: \tau_n}\ \textbf{free in}\ e \Downarrow \Delta' : v} \qquad\qquad y_1, \ldots, y_n \text{ fresh}$$

$$(\textsc{Fun})\quad \frac{\Delta : e\ [\overline{\tau_m/\alpha_m}, \overline{y_n/x_n}] \Downarrow \Delta' : v}{\Delta : f_{\,\overline{\tau_m}}\,(y_1, \ldots, y_n) \Downarrow \Delta' : v} \qquad\qquad \text{for } f :: \forall \alpha_1 \mathinner{..} \alpha_m . \tau ; f\ \overline{x_n} = e \text{ in P}$$

$$(\textsc{Apply})\quad \frac{\Delta : e_1 \Downarrow \Delta' : \varphi(y_1, \ldots, y_k) \qquad \Delta' : \textbf{let}\ x = e_2\ \textbf{in}\ \varphi(y_1, \ldots, y_k, x) \Downarrow \Delta'' : v}{\Delta : e_1\ e_2 \Downarrow \Delta'' : v}$$

$$(\textsc{Plus})\quad \frac{\Delta : e_1 \Downarrow \Delta' : n_1 \qquad \Delta' : e_2 \Downarrow \Delta'' : n_2}{\Delta : e_1 + e_2 \Downarrow \Delta'' : n} \qquad\qquad \text{for literals } n_1,\ n_2 \text{ and } n = n_1 + n_2$$

$$(\textsc{Case}_{\mathsf{Nil}})\quad \frac{\Delta : e \Downarrow \Delta' : \mathsf{Nil}_{\,\tau} \qquad \Delta' : e_1 \Downarrow \Delta'' : v}{\Delta : \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \to e_1 ; \mathsf{Cons}\ h\ t \to e_2\} \Downarrow \Delta'' : v}$$

$$(\textsc{Case}_{\mathsf{Cons}})\quad \frac{\Delta : e \Downarrow \Delta' : \mathsf{Cons}_{\,\tau}\,(y_1, y_2) \qquad \Delta' : e_2\ [y_1/h, y_2/t] \Downarrow \Delta'' : v}{\Delta : \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \to e_1 ; \mathsf{Cons}\ h\ t \to e_2\} \Downarrow \Delta'' : v}$$

$$(\textsc{Guess}_n)\quad \Delta : \textbf{unknown}_{\,\mathsf{Nat}} \Downarrow \Delta : n$$

$$(\textsc{Guess}_{\mathsf{Nil}})\quad \Delta : \textbf{unknown}_{\,[\tau]} \Downarrow \Delta : \mathsf{Nil}_{\,\tau}$$

$$(\textsc{Guess}_{\mathsf{Cons}})\quad \frac{\Delta : \textbf{let}\ h :: \tau, t :: [\tau]\ \textbf{free in}\ \mathsf{Cons}_{\,\tau}\,(h, t) \Downarrow \Delta' : v}{\Delta : \textbf{unknown}_{\,[\tau]} \Downarrow \Delta' : v}$$

Figure 4.8: Operational semantics without logic variables for CuMin

is never looked up, the heap entry remains untouched throughout the whole derivation and it is simply replaced using **unknown** everywhere. Otherwise, we have the following situation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta'\ [y :: \tau\ \mathbf{free}]\ \Omega' : y \Downarrow^l \Delta'\ [y :: \tau\ \mathbf{free}]\ \Omega' : y}{\vdots \qquad\qquad\qquad\qquad \vdots}}{\vdots}}{\Delta : e \Downarrow^l \Delta'\ [y :: \tau\ \mathbf{free}]\ \Omega'' : y}}{\Delta : e \Downarrow^f \Delta'\ \Phi\ [y :: \tau \mapsto v]\ \Omega'' : v}\ (\textsc{Guess})$$

The first use of (Val) for the logic variable y results in y appearing on the value side, where it is handed down through the derivation. As the root of the tree is a functional derivation judgment, it does not allow y as its result. Therefore at some point the value y has to be replaced by some flat normal form. This can only happen via some application of (Guess). The logic variable y is replaced by a fully applied constructor, whose fields (if any) are some new logic variables bound within the (possibly empty) heap Φ.

In the operational semantics without logic variables, y is bound to $\mathbf{unknown}_\tau$ in the heap. Since y cannot serve as a value, instead of (Val) the rule (Lookup) has to be used, leading to the following situation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta'^u : \mathbf{unknown}_\tau \Downarrow \Delta'^u\ \Phi^u : v}{\Delta'^u\ [y :: \tau \mapsto \mathbf{unknown}_\tau]\ \Omega : y \Downarrow \Delta'^u\ \Phi^u\ [y :: \tau \mapsto v]\ \Omega'^u : v}\ (\textsc{Lookup})}{\vdots \qquad\qquad\qquad\qquad \vdots}}{\vdots}}{\Delta^u : e \Downarrow \Delta'^u\ \Phi^u\ [y :: \tau \mapsto v]\ \Omega''^u : v}$$

The missing premise of the (Guess) rule can always be chosen in such a way, that $\Delta'^u\ \Phi^u : v$ is the result. Instead of y the result is v, which is handed down to the point where guessing took place in the original derivation. Since y has already been evaluated, no further guessing occurs. For the remaining derivation y is no longer logic and no further adjustments are necessary when transforming the derivation tree.

A logic variable y can also be introduced by a (Guess) rule which determines the value of some other logic variable z in the operational semantics with logic variables. Yet if the transformation described above is applied for the variable z first, the introduction of y is done via (Free) and thus the transformation is applicable for y as well.

The other direction of the claim is proved using the inverse transformation: Instead of evaluating **unknown**, return the logic variable and hand it down until functional evaluation is forced.

□

4.3 Denotational Semantics

In this section a functional-style denotational semantics for CuMin is given, which is based on pointed partially ordered sets. As this is a somewhat nonstandard choice, we

start with a motivating remark about recursion and nondeterminism.
We imagine a language that is non-strict, has logic variables, and only allows structural recursion. Structural recursion means that the language has primitives for folding over lists and natural numbers. So, recursion is always guided by some structure, as opposed to general recursion, which allows functions to call each other and themselves arbitrarily. Using the given features, general recursion can be implemented.
Take, for example, the recursive function *repeat* :: $a \to [a]$ producing an infinite list of copies of its input. Instead of using general recursion, the function can be given as

$$\begin{array}{l} \mathit{repeat}' :: (a \to [a]) \to a \to [a] \\ \mathit{repeat}'\ f\ x = x : f\ x \\ \mathit{repeat} :: a \to [a] \\ \mathit{repeat}\ x = \textbf{let}\ n\ \textbf{free in}\ \mathit{foldn}\ \mathit{repeat}'\ \textbf{failed}\ n\ x \end{array}$$

where we assume *foldn* :: $(a \to a) \to a \to \mathsf{Nat} \to a$ to be structural recursion over natural numbers, i.e., n-fold application of the same function. Replacing a recursive function by an n-fold iteration of a non-recursive function works just as well in a deterministic language. However, in the nondeterministic setting, the number of allowed steps does not have to be specified. Instead, the nondeterministic search mechanism can be used to find the number of steps necessary to achieve a result.

Taking these considerations into account, it seems reasonable to treat recursion and infinite structures alongside nondeterminism. This leads to a significant simplification of the denotational semantics as compared to earlier versions based on dcpos like [Christiansen et al., 2011a]. We do not even have to formally introduce dcpos, because the notion will not show up anywhere. For the element level, we use partially ordered sets, which are more general than dcpos and conceptually easier because of the missing limit structure. For the set level we use semi-lattices, which are more specific than dcpos but also conceptually easier because taking limits is not restricted to special families.

4.3.1 Preliminaries on Posets

A *partially ordered set*, or *poset* for short, is a set (sometimes called the *underlying set*) together with a reflexive, antisymmetric and transitive relation on it. We think of these relations as definedness relations and denote them by $\sqsubseteq$. All posets will have a least element $\perp$. While technically it would therefore be more appropriate to call them "pointed posets", we stick to "posets" and take this to include being pointed as well.
A function between posets that is order-preserving is called *monotone*. We do not require monotone functions to preserve the least element. If they still do, they are called *strict*.
A *lower set* of a poset P is a nonempty subset A of P with the property that $\mathbf{x} \in A$ and $\mathbf{y} \sqsubseteq \mathbf{x}$ imply $\mathbf{y} \in A$. Note that all lower sets contain $\perp$. The set of all lower sets of a poset P is denoted by $\mathcal{L}(P)$ and is itself partially ordered by set inclusion $\subseteq$. The least element of $\mathcal{L}(P)$ is $\{\perp\}$, which is therefore also denoted $\perp$. The poset $\mathcal{L}(P)$ admits taking unions of arbitrary collections of sets, as the union of lower sets is always a lower

$$\begin{aligned}
[\![\alpha]\!]_\theta &= [\![\theta(\alpha)]\!] \\
[\![\tau \to \tau']\!]_\theta &= \{\mathbf{f} : [\![\tau]\!]_\theta \times \widehat{\mathbb{N}} \to \mathcal{L}\left([\![\tau']\!]_\theta\right) \mid \mathbf{f} \text{ monotone}\} \\
[\![\mathsf{Bool}]\!]_\theta &= \{\mathbf{True}, \mathbf{False}\}_\perp \\
[\![\mathsf{Nat}]\!]_\theta &= \mathbb{N}_\perp \\
[\![[\tau]]\!]_\theta &= \{\mathbf{x}_1 : \ldots : \mathbf{x}_n : \mathbf{e} \mid n \geq 0, \mathbf{x}_i \in [\![\tau]\!]_\theta, \mathbf{e} \in \{\perp, []\}\} \\
[\![(\tau, \tau')]\!]_\theta &= \{(\mathbf{l}, \mathbf{r}) \mid \mathbf{l} \in [\![\tau]\!]_\theta, \mathbf{r} \in [\![\tau']\!]_\theta\}_\perp
\end{aligned}$$

Figure 4.9: Denotational type semantics for CuMin

set itself. By convention, building the union of an empty collection of sets results in $\{\perp\}$ rather than the empty set because the latter is not a lower set.
The *closure* $\downarrow\mathbf{x}$ of an element $\mathbf{x}$ of some poset P is the (lower) set $\{\mathbf{y} \mid \mathbf{y} \in P, \mathbf{y} \sqsubseteq \mathbf{x}\} \in \mathcal{L}(P)$. The function taking $\mathbf{x} \in P$ to $\downarrow\mathbf{x} \in \mathcal{L}(P)$ is a strict monotone function.

4.3.2 Semantics of Types

The semantics of types is defined in two steps. First we define the semantics $[\![\tau]\!]$ for closed types τ and then we generalize the definition to all types. For the second step, we need a *type environment* and define the semantics $[\![\tau]\!]_\theta$ of an arbitrary type τ with respect to this environment. Given some context Γ, an environment corresponding to Γ maps every type variable α to a closed type $\theta(\alpha)$, where $\theta(\alpha) \in \mathsf{Data}$ has to hold whenever the constraint $\alpha \in \mathsf{Data}$ is part of the context Γ. The rules for both steps are given in figure 4.9 and coincide for closed types, as there is no actual dependency on θ in this case.
$\widehat{\mathbb{N}}$ is the poset given by the underlying set $\mathbb{N} \cup \{\infty\}$ and the (total) order $n \sqsubseteq n' \iff n \leq n'$ (so in $\widehat{\mathbb{N}}$ we have $\perp = 0$). Being monotone means $\mathbf{f}(\mathbf{a}, n) \subseteq \mathbf{f}(\mathbf{a}', n')$ whenever $\mathbf{a} \sqsubseteq \mathbf{a}'$ and $n \leq n'$. Functions are compared point-wise where the order on the target is set-inclusion as discussed above.
$\mathbb{N}$ and $\{\mathbf{True}, \mathbf{False}\}$ otherwise carry the flat order, i.e., different elements are incomparable. The subscript $\perp$ adds a least element and preserves the order between the other elements.
The set of lists carries the smallest partial order satisfying $\perp \sqsubseteq \mathbf{v}$ for all $\mathbf{v}$ and $\mathbf{x} : \mathbf{xs} \sqsubseteq \mathbf{y} : \mathbf{ys}$ whenever $\mathbf{x} \sqsubseteq \mathbf{y}$ and $\mathbf{xs} \sqsubseteq \mathbf{ys}$. In particular, $\mathbf{x}_1 : \ldots : \mathbf{x}_n : \mathbf{e_x} \sqsubseteq \mathbf{y}_1 : \ldots : \mathbf{y}_m : \mathbf{e_y}$ holds if $n = m$, $\mathbf{e_y} = []$ and $\mathbf{x}_i \sqsubseteq \mathbf{y}_i$ for $1 \leq i \leq n$ or $n \leq m$, $\mathbf{e_x} = \perp$ and $\mathbf{x}_i \sqsubseteq \mathbf{y}_i$ for $1 \leq i \leq n$. Pairs are compared entry-wise but again an element $\perp$ is added as least element.

4.3.3 Semantics of Terms

Next we define the semantics of terms. Let $\Gamma \vdash e :: \tau$ be a typing judgment where X is the set of term variables in Γ. Let θ be a type environment as before. Additionally, let σ a term environment, i.e., a mapping with $\sigma(x) \in [\![\Gamma\ (x)]\!]_\theta$ for all $x \in X$. We will define

$[\![e]\!]^t_{\theta,\sigma} \in \mathcal{L}\left([\![\tau]\!]_\theta\right)$ for all $t \in \widehat{\mathbb{N}}$. For all expressions e we define $[\![e]\!]^0_{\theta,\sigma} = \{\bot\}$. For finite t the denotational semantics is defined recursively in figures 4.10 and 4.11. For $t = \infty$ we define $[\![e]\!]^\infty_{\theta,\sigma} = \bigcup_{t\in\mathbb{N}} [\![e]\!]^t_{\theta,\sigma}$. This is the smallest fixpoint of figures 4.10 and 4.11 read as a system of equations for $t = \infty$ (where $\infty + n = \infty$).
Note that figure 4.10 also gives a denotational semantics for $\varphi(y_1, \ldots, y_k)$, which we will need when comparing the denotational and the operational semantics. Also, the denotational semantics for $\mathbf{unknown}_\tau$ is used, which is defined in figure 4.11. The λ symbol represents lambda abstractions on the semantic level and introduces two variables: one for a semantic value and one for a step-index. The operators $+^\bot$ and $=^\bot$ are strict extensions of addition and the Boolean-valued semantic equality test.

In [Launchbury, 1993] the denotational semantics of function types is defined as

$$[\![\tau \to \tau']\!]_\theta = \left\{\mathbf{f} : \left(\widehat{\mathbb{N}} \to [\![\tau]\!]_\theta\right) \times \widehat{\mathbb{N}} \to [\![\tau']\!]_\theta \mid \mathbf{f} \text{ monotone}\right\},$$

while in [Mehner et al., 2014] it is defined as:

$$[\![\tau \to \tau']\!]_\theta = \left\{\mathbf{f} : [\![\tau]\!]_\theta \to \mathcal{L}\left([\![\tau']\!]_\theta\right) \mid \mathbf{f} \text{ monotone}\right\}$$

This shows the different approaches to step-indexing. In [Launchbury, 1993] the semantic function $\mathbf{f}$ receives the whole sequence of denotations (i.e., for every step-index) of the argument. Environments also contain sequences of denotations. When variables are looked up in the environment, the current step-index is used to pick one element of the sequence. Intuitively, the function 'decides' how much time is available for evaluating the argument. In the version presented here, there is no such dependency, semantic functions expect a single value, and single values are stored in the environment. The same is true for the version given in [Mehner et al., 2014], going back to [Christiansen et al., 2011a].
The semantic function $\mathbf{f}$ expects a second argument in both Launchbury's version and the version given here. This argument determines, how much time is available for evaluating the function body.

The unrestricted version of the denotational semantics is uniquely determined by being the smallest fixed point of the defining equations. We can use this as an alternative definition, thus skipping the detour via a step-indexed version. In this case (semantic) functions are only ever applied to $t = \infty$. We can thus simplify the semantics by using

$$[\![\tau \to \tau']\!]_\theta = \left\{\mathbf{f} : [\![\tau]\!]_\theta \to \mathcal{L}\left([\![\tau']\!]_\theta\right) \mid \mathbf{f} \text{ monotone}\right\}$$

on the type level,

$$[\![e_1\ e_2]\!]_{\theta,\sigma} = \bigcup_{\mathbf{a}\in[\![e_2]\!]_{\theta,\sigma}} \bigcup_{\mathbf{f}\in[\![e_1]\!]_{\theta,\sigma}} \mathbf{f}(\mathbf{a})$$

on the term level and not abstracting over t in all the lambda abstractions.

$$\begin{aligned}
\llbracket x\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft\sigma(x) & \llbracket \mathsf{Cons}_{\,\tau}\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft(\lambda\,\mathbf{h}\ t_1.\downharpoonleft(\lambda\,\mathbf{t}\ t_2.\downharpoonleft(\mathbf{h}:\mathbf{t})))\\
\llbracket \mathbf{failed}_{\,\tau}\rrbracket^{t+1}_{\theta,\sigma} &= \{\bot\} & \llbracket \mathsf{Cons}_{\,\tau}\ (y_1)\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft(\lambda\,\mathbf{t}\ t_2.\downharpoonleft(\sigma(y_1):\mathbf{t}))\\
\llbracket n\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft\mathbf{n} & \llbracket \mathsf{Cons}_{\,\tau}\ (y_1,y_2)\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft(\sigma(y_1):\sigma(y_2))\\
\llbracket \mathsf{True}\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft\mathbf{True} & \llbracket \mathsf{Pair}_{\,\tau,\tau'}\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft(\lambda\,\mathbf{l}\ t_1.\downharpoonleft(\lambda\,\mathbf{r}\ t_2.\downharpoonleft(\mathbf{l},\mathbf{r})))\\
\llbracket \mathsf{False}\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft\mathbf{False} & \llbracket \mathsf{Pair}_{\,\tau,\tau'}\ (y_1)\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft(\lambda\,\mathbf{r}\ t_2.\downharpoonleft(\sigma(y_1),\mathbf{r}))\\
\llbracket \mathsf{Nil}_{\,\tau}\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft[] & \llbracket \mathsf{Pair}_{\,\tau,\tau'}\ (y_1,y_2)\rrbracket^{t+1}_{\theta,\sigma} &= \downharpoonleft(\sigma(y_1),\sigma(y_2))
\end{aligned}$$

$$\llbracket \mathbf{let}\ \overline{x_n = e_n}\ \mathbf{in}\ e\rrbracket^{t+1}_{\theta,\sigma} = \bigcup_{\mathbf{x}_1\in\llbracket e_1\rrbracket^{t}_{\theta,\sigma}} \cdots \bigcup_{\mathbf{x}_n\in\llbracket e_n\rrbracket^{t}_{\theta,\sigma[x_1\mapsto\mathbf{x}_1,\ldots,x_{n-1}\mapsto\mathbf{x}_{n-1}]}} \llbracket e\rrbracket^{t}_{\theta,\sigma[x_1\mapsto\mathbf{x}_1,\ldots,x_n\mapsto\mathbf{x}_n]}$$

$$\llbracket \mathbf{let}\ \overline{x_n :: \tau_n}\ \mathbf{free\ in}\ e\rrbracket^{t+1}_{\theta,\sigma} = \bigcup_{\mathbf{x}_1\in\llbracket \mathbf{unknown}_{\tau_1}\rrbracket^{t}_{\theta,\sigma}} \cdots \bigcup_{\mathbf{x}_n\in\llbracket \mathbf{unknown}_{\tau_n}\rrbracket^{t}_{\theta,\sigma}} \llbracket e\rrbracket^{t}_{\theta,\sigma[x_1\mapsto\mathbf{x}_1,\ldots,x_n\mapsto\mathbf{x}_n]}$$

$$\llbracket f\ \overline{\tau_m}\ (y_1,\ldots,y_k)\rrbracket^{t_k+1}_{\theta,\sigma} = \downharpoonleft\left(\lambda\,\mathbf{a}_{k+1}\ t_{k+1}.\cdots\downharpoonleft\left(\lambda\,\mathbf{a}_n\ t_n.\llbracket e\rrbracket^{t_n}_{\overline{[\alpha_m\mapsto\tau_m\theta]},\sigma'}\right)\cdots\right)$$

$$\text{with } k\le n,\ f :: \forall\alpha_1\,..\,\alpha_m.\tau; f\ x_1\,..\,x_n = e \text{ in } P \text{ and}$$
$$\sigma' = [x_1\mapsto\sigma(y_1),\ldots,x_k\mapsto\sigma(y_k),x_{k+1}\mapsto\mathbf{a}_{k+1},\ldots,x_n\mapsto\mathbf{a}_n]$$

$$\begin{aligned}
\llbracket e_1\ e_2\rrbracket^{t+2}_{\theta,\sigma} &= \bigcup_{\mathbf{a}\in\llbracket e_2\rrbracket^{t+1}_{\theta,\sigma}}\ \bigcup_{\mathbf{f}\in\llbracket e_1\rrbracket^{t+1}_{\theta,\sigma}} \mathbf{f}(\mathbf{a},t)\\
\llbracket e_1 + e_2\rrbracket^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{n}_1\in\llbracket e_1\rrbracket^{t}_{\theta,\sigma}}\ \bigcup_{\mathbf{n}_2\in\llbracket e_2\rrbracket^{t}_{\theta,\sigma}} \downharpoonleft(\mathbf{n}_1 +^{\bot} \mathbf{n}_2)\\
\llbracket e_1 == e_2\rrbracket^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{n}_1\in\llbracket e_1\rrbracket^{t}_{\theta,\sigma}}\ \bigcup_{\mathbf{n}_2\in\llbracket e_2\rrbracket^{t}_{\theta,\sigma}} \downharpoonleft(\mathbf{n}_1 =^{\bot} \mathbf{n}_2)\\
\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{True}\to e_1;\mathsf{False}\to e_2\}\rrbracket^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{b}\in\llbracket e\rrbracket^{t}_{\theta,\sigma}} \begin{cases}\llbracket e_1\rrbracket^{t}_{\theta,\sigma} & \mathbf{b}=\mathbf{True}\\ \llbracket e_2\rrbracket^{t}_{\theta,\sigma} & \mathbf{b}=\mathbf{False}\\ \{\bot\} & \mathbf{b}=\bot\end{cases}\\
\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil}\to e_1;\mathsf{Cons}\ h\ t\to e_2\}\rrbracket^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{l}\in\llbracket e\rrbracket^{t}_{\theta,\sigma}} \begin{cases}\llbracket e_1\rrbracket^{t}_{\theta,\sigma} & \mathbf{l}=[]\\ \llbracket e_2\rrbracket^{t}_{\theta,\sigma[h\mapsto\mathbf{h},t\mapsto\mathbf{t}]} & \mathbf{l}=\mathbf{h}:\mathbf{t}\\ \{\bot\} & \mathbf{l}=\bot\end{cases}\\
\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Pair}\ l\ r\to e_1\}\rrbracket^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{p}\in\llbracket e\rrbracket^{t}_{\theta,\sigma}} \begin{cases}\llbracket e_1\rrbracket^{t}_{\theta,\sigma[l\mapsto\mathbf{l},r\mapsto\mathbf{r}]} & \mathbf{p}=(\mathbf{l},\mathbf{r})\\ \{\bot\} & \mathbf{p}=\bot\end{cases}
\end{aligned}$$

Figure 4.10: Denotational term semantics for CuMin

$$
\begin{aligned}
[\![\textbf{unknown}_{\,\alpha}]\!]^{t+1}_{\theta,\sigma} &= [\![\textbf{unknown}_{\,\theta(\alpha)}]\!]^{t+1}_{\theta,\sigma}\\
[\![\textbf{unknown}_{\,\mathsf{Bool}}]\!]^{t+1}_{\theta,\sigma} &= [\![\mathsf{Bool}]\!]\\
[\![\textbf{unknown}_{\,\mathsf{Nat}}]\!]^{t+1}_{\theta,\sigma} &= [\![\mathsf{Nat}]\!]\\
[\![\textbf{unknown}_{\,[\tau]}]\!]^{t+1}_{\theta,\sigma} &= \downarrow[] \cup \bigcup_{\mathbf{h}\in[\![\textbf{unknown}_{\tau}]\!]^{t}_{\theta,\sigma}} \; \bigcup_{\mathbf{t}\in[\![\textbf{unknown}_{[\tau]}]\!]^{t}_{\theta,\sigma}} \downarrow(\mathbf{h}:\mathbf{t})\\
[\![\textbf{unknown}_{\,(\tau,\tau')}]\!]^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{l}\in[\![\textbf{unknown}_{\tau}]\!]^{t}_{\theta,\sigma}} \; \bigcup_{\mathbf{r}\in[\![\textbf{unknown}_{\tau'}]\!]^{t}_{\theta,\sigma}} \downarrow(\mathbf{l},\mathbf{r})
\end{aligned}
$$

Figure 4.11: Denotational term semantics for **unknown** primitive in CuMin

Lemma 4.3.1.

$$[\![\varphi(y_1,\ldots,y_k)]\!]^{t+1}_{\theta,\sigma} = \downarrow\left(\lambda\,\mathbf{a}\;t'.[\![\varphi(y_1,\ldots,y_k,x)]\!]^{t'+1}_{\theta,\sigma[x\mapsto\mathbf{a}]}\right)$$

Proof. We apply the definition of the semantics of a (partial) application twice

$$
\begin{aligned}
&[\![f_{\,\overline{\tau_m}}\,(y_1,\ldots,y_k)]\!]^{t+1}_{\theta,\sigma}\\
&= \downarrow\left(\lambda\,\mathbf{a}_{k+1}\;t_{k+1}.\cdots\downarrow\left(\lambda\,\mathbf{a}_n\;t_n.[\![e]\!]^{t_n}_{[\overline{\alpha_m\mapsto\tau_m\theta}],\sigma'}\right)\cdots\right)\\
&= \downarrow\left(\lambda\,\mathbf{a}_{k+1}\;t_{k+1}.[\![f_{\,\overline{\tau_m}}\,(y_1,\ldots,y_k,x)]\!]^{t_{k+1}+1}_{\theta,\sigma[x\mapsto\mathbf{a}_{k+1}]}\right)
\end{aligned}
$$

where $\sigma' = [x_1 \mapsto \sigma(y_1),\ldots,x_k \mapsto \sigma(y_k), x_{k+1} \mapsto \mathbf{a}_{k+1},\ldots,x_n \mapsto \mathbf{a}_n]$. Indeed $\sigma[x \mapsto \mathbf{a}_{k+1}](y_i) = \sigma(y_i)$ and $\sigma[x \mapsto \mathbf{a}_{k+1}](x) = \mathbf{a}_{k+1}$. □

Lemma 4.3.2. The denotational semantics of a term is monotone w.r.t. the variable bindings and the step index $t \in \widehat{\mathbb{N}}$.

Proof. Induction on the syntax. □

Lemma 4.3.3. $[\![\textbf{unknown}_{\,\tau}]\!]^{\infty}_{\theta,\sigma} = [\![\tau]\!]_\theta$ for all $\tau \in \mathsf{Data}$.

Proof. The proof is by induction on the rules for $\tau \in \mathsf{Data}$. If τ is some type variable α which is marked as representing a data type by the context, then $\theta(\tau)$ also has to be in the Data class. For closed types, the claim follows from the definition of $[\![\textbf{unknown}_{\,\tau}]\!]$ and $[\![\tau]\!]$. □

4.4 Correctness

The first important result (Theorem 2) in [Launchbury, 1993] is correctness of the operational semantics w.r.t. the denotational semantics, i.e., evaluation does not change the denotational meaning. The formalization uses denotational semantics of heaps: Given a heap Δ and an environment σ, an extended environment $\{\!\{\Delta\}\!\}_\sigma$ is defined, which contains semantic values for all variables bound in Δ or σ. The correctness statement claims that

$$[\![e]\!]_{\{\!\{\Delta\}\!\}_\sigma} = [\![v]\!]_{\{\!\{\Delta'\}\!\}_\sigma}$$

holds for all derivable judgements $\Delta : e \Downarrow \Delta' : v$ and all environments σ.[6] The theorem also provides a result about the heaps: If x is a variable with $\{\!\{\Delta\}\!\}_\sigma(x) \neq \bot$, then $\{\!\{\Delta\}\!\}_\sigma(x) = \{\!\{\Delta'\}\!\}_\sigma(x)$. Both claims are proven together by induction on all derivations of the operational semantics. The proof is rather straightforward and requires little additional knowledge besides the definitions of the semantics.

In [Launchbury, 1993], the denotational semantics for expressions and heaps rely on each other and therefore are defined jointly. We have already defined the denotational semantics of CuMin expressions without referring to heaps. This simplification is possible because of the absence of recursively defined local variables in CuMin. Comparing the denotational semantics to the operational semantics does however require the following definition.

The denotational semantics of a heap $\Delta = [\, y_1 :: \tau_1 \mapsto e_1, .., y_l :: \tau_l \mapsto e_l]$ with respect to an environment σ is defined as:

$$\{\!\{\Delta\}\!\}_\sigma := \bigcup_{\mathbf{v}_1 \in [\![e_1]\!]^\infty_\sigma} \bigcup_{\mathbf{v}_2 \in [\![e_2]\!]^\infty_{\sigma[y_1 \mapsto \mathbf{v}_1]}} \cdots \bigcup_{\mathbf{v}_l \in [\![e_l]\!]^\infty_{\sigma[y_1 \mapsto \mathbf{v}_1, \ldots, y_{l-1} \mapsto \mathbf{v}_{l-1}]}} \downarrow [y_1 \mapsto \mathbf{v}_1, \ldots, y_l \mapsto \mathbf{v}_l]$$

In CuMin, the denotational semantics of a heap is a *set of* environments, just like the denotational semantics of an expression is a *set of* values. Here the definedness order on environments is point-wise, i.e., an environment σ is at most as defined as another environment σ' (for the same sequence of variables having corresponding types) if $\sigma(x) \sqsubseteq \sigma'(x)$ for all variables x in the environments. We will often write $\{\!\{\Delta\}\!\}$ instead of $\{\!\{\Delta\}\!\}_{[]}$.

Now we can state this section's main result:

Theorem 4.4.1. For all judgments $\Delta : e \Downarrow \Delta' : v$ the following holds:

$$\bigcup_{\delta \in \{\!\{\Delta\}\!\}} [\![e]\!]_\delta \supseteq \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} [\![v]\!]_{\delta'} \tag{4.1}$$

Taking unions is necessary because of the denotational semantics of both expressions and environments being sets. The claim is a subset relation rather than an equality of sets, because any one particular course of evaluation only needs to produce some of the possible results.

Christiansen et al. [2011a] try to prove the above theorem directly by induction. No definition for the denotational semantics of heaps is given, but only for heap expression

[6] When trying to formalize Launchbury's work in Isabelle, Breitner [2014] found out that this claim is not true in full generality, though it does hold in all relevant cases. Breitner makes different suggestions for fixing this issue, one of which is changing the definition of the denotational semantics and thus making Launchbury's original proof correct.

pairs. Yet Launchbury's approach suggests that we also need control over the involved heaps independent of the expressions. To this end, he introduces a relation $\sigma \leq \sigma'$ between environments stating that all variables bound to a non-bottom value in σ are bound to the same value in σ'.

We use the same idea though in a different guise. If $\Delta\!: e \Downarrow \Delta'\!: v$ is a derivable judgement and X and X' are the sets of variables appearing in Δ and Δ' we define the map $\pi_X^{X'}$ as the projection keeping the entries for all variables in Δ and dropping the rest. In most cases we will omit the sets X and X' if they are clear from the context. Using the new notation, the relation $\sigma \leq \sigma'$ can be written as $\sigma = \pi_X^{X'}(\sigma')$.

Theorem 4.4.1 does not state equality though. Therefore we cannot expect equality to hold when it comes to the heaps, either. Rather, we expect the following:

$$\{\!\{\Delta\}\!\} \supseteq \left\{ \pi_X^{X'}\left(\delta'\right) \mid \delta' \in \{\!\{\Delta'\}\!\} \right\} \tag{4.2}$$

Both this and theorem 4.4.1 will be consequences of the next lemma. Yet the lemma does claim more than simply the conjunction of (4.1) and (4.2). To do the inductive proof, it is necessary to know which *combinations* of environments δ and semantic values $\mathbf{v}$ occur. We will prove the following:

Lemma 4.4.2. If a judgment $\Delta\!: e \Downarrow \Delta'\!: v$ can be derived by the operational semantics, then

$$\bigcup_{\delta \in \{\!\{\Delta\}\!\}} \bigcup_{\mathbf{v} \in [\![e]\!]_\delta} \downarrow\!(\delta, \mathbf{v}) \supseteq \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \bigcup_{\mathbf{v} \in [\![v]\!]_{\delta'}} \downarrow \left(\pi_X^{X'}(\delta'), \mathbf{v} \right) \tag{4.3}$$

where X and X' are the sets of variables defined in Δ and Δ'.

This somewhat bulky inequation implies both (4.1) and (4.2). The former inequation (the theorem) emerges by taking the right projection of all tuples in both sides of (4.3) and the latter emerges by taking left projections.

Before we do the proof, we single out another key idea as an additional lemma:

Lemma 4.4.3. If v is a value under some heap Δ then $[\![v]\!]_\delta$ is the closure of a single element for every $\delta \in \{\!\{\Delta\}\!\}$.

Proof. This is immediate from the definition of the denotational semantics, where all values are explicitly given as closures. □

Having these ideas in mind the actual proof is mainly bookkeeping.

Proof of lemma 4.4.2. The proof is by induction over all derivations and splitting cases according to the rules of the operational semantics. For every rule, we may assume the claim to already be shown for all premises.

For the (Val) rule the claim is trivially satisfied.

Now for lookup:

$$(\textsc{Lookup})\ \frac{\Delta : e \Downarrow \Delta' : v}{\Delta\, [x :: \tau \mapsto e]\, \Omega : x \Downarrow \Delta'\, [x :: \tau \mapsto v]\, \Omega : v}$$

$$\bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} \bigcup_{\xi\in\{\!\!\{[x\mapsto e]\}\!\!\}_{\delta}} \bigcup_{\omega\in\{\!\!\{\Omega\}\!\!\}_{\delta\xi}} \bigcup_{\mathbf{v}\in\llbracket x\rrbracket_{\delta\xi\omega}} \downarrow(\delta\xi\omega,\mathbf{v})$$

Use $\mathbf{x}=\xi(x)$.

$$\begin{aligned}
&= \bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} \bigcup_{\mathbf{x}\in\llbracket e\rrbracket_{\delta}} \bigcup_{\omega\in\{\!\!\{\Omega\}\!\!\}_{\delta[x\mapsto\mathbf{x}]}} \bigcup_{\mathbf{v}\in\llbracket x\rrbracket_{\delta[x\mapsto\mathbf{x}]\omega}} \downarrow(\delta[x\mapsto\mathbf{x}]\omega,\mathbf{v})\\
&= \bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} \bigcup_{\mathbf{x}\in\llbracket e\rrbracket_{\delta}} \bigcup_{\omega\in\{\!\!\{\Omega\}\!\!\}_{\delta[x\mapsto\mathbf{x}]}} \bigcup_{\mathbf{v}\in\downarrow\mathbf{x}} \downarrow(\delta[x\mapsto\mathbf{x}]\omega,\mathbf{v})\\
&= \bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} \bigcup_{\mathbf{x}\in\llbracket e\rrbracket_{\delta}} \bigcup_{\omega\in\{\!\!\{\Omega\}\!\!\}_{\delta[x\mapsto\mathbf{x}]}} \downarrow(\delta[x\mapsto\mathbf{x}]\omega,\mathbf{x})
\end{aligned}$$

Induction hypothesis

$$\supseteq \bigcup_{\delta'\in\{\!\!\{\Delta'\}\!\!\}} \bigcup_{\mathbf{x}\in\llbracket v\rrbracket_{\delta'}} \bigcup_{\omega\in\{\!\!\{\Omega\}\!\!\}_{\pi(\delta')[x\mapsto\mathbf{x}]}} \downarrow(\pi(\delta')[x\mapsto\mathbf{x}]\omega,\mathbf{x})$$

Because of lemma 4.4.3, $\llbracket v\rrbracket_{\delta'}$ is the closure of a single element and we may introduce another union:

$$\begin{aligned}
&= \bigcup_{\delta'\in\{\!\!\{\Delta'\}\!\!\}} \bigcup_{\mathbf{x}\in\llbracket v\rrbracket_{\delta'}} \bigcup_{\omega\in\{\!\!\{\Omega\}\!\!\}_{\pi(\delta')[x\mapsto\mathbf{x}]}} \bigcup_{\mathbf{v}\in\llbracket v\rrbracket_{\delta'[x\mapsto\mathbf{x}]\omega}} \downarrow(\pi(\delta')[x\mapsto\mathbf{x}]\omega,\mathbf{v})\\
&= \bigcup_{\delta'\in\{\!\!\{\Delta'\}\!\!\}} \bigcup_{\xi\in\{\!\!\{[x\mapsto v]\}\!\!\}_{\delta'}} \bigcup_{\omega\in\{\!\!\{\Omega\}\!\!\}_{\pi(\delta')\xi}} \bigcup_{\mathbf{v}\in\llbracket v\rrbracket_{\delta'\xi\omega}} \downarrow(\pi(\delta')\xi\omega,\mathbf{v})
\end{aligned}$$

$$(\textsc{Let})\ \frac{\Delta\ [y::\tau_1\mapsto e_1]: e\ [y/x]\Downarrow\Delta' : v}{\Delta : \mathbf{let}\ x=e_1\ \mathbf{in}\ e\Downarrow\Delta' : v} \qquad y \text{ fresh and } \tau_1 \text{ is the type of } \Delta : e_1$$

$$(\textsc{Free})\ \frac{\Delta\ [y::\tau_1\mapsto \mathbf{unknown}_{\tau_1}]: e\ [y/x]\Downarrow\Delta' : v}{\Delta : \mathbf{let}\ x::\tau_1\ \mathbf{free\ in}\ e\Downarrow\Delta' : v} \qquad y \text{ fresh}$$

(Let) and (Free):

$$\begin{aligned}
&\bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} \bigcup_{\mathbf{v}\in\llbracket \mathbf{let}\ x=e_1\ \mathbf{in}\ e\rrbracket_{\delta}} \downarrow(\delta,\mathbf{v})\\
&= \bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} \bigcup_{\mathbf{x}\in\llbracket e_1\rrbracket_{\delta}} \bigcup_{\mathbf{v}\in\llbracket e\rrbracket_{\delta[x\mapsto\mathbf{x}]}} \downarrow(\delta,\mathbf{v})\\
&= \bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} \bigcup_{\xi\in\{\!\!\{[y\mapsto e_1]\}\!\!\}_{\delta}} \bigcup_{\mathbf{v}\in\llbracket e\ [y/x]\rrbracket_{\delta\xi}} \downarrow(\pi(\delta\xi),\mathbf{v})\\
&\supseteq \bigcup_{\delta'\in\{\!\!\{\Delta'\}\!\!\}} \bigcup_{\mathbf{v}\in\llbracket v\rrbracket_{\delta'}} \downarrow(\pi(\delta'),\mathbf{v})
\end{aligned}$$

(Fun):

$$
\begin{aligned}
& \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket f_{\overline{\tau_m}}(y_1,..,y_n) \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) \\
= & \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket e \rrbracket_{\overline{[\alpha_m \mapsto \llbracket \tau_m \rrbracket]}\,\overline{[x_n \mapsto \delta(y_n)]}}} \downarrow(\delta, \mathbf{v}) \\
= & \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket e\ [\overline{\tau_m/\alpha_m}, \overline{y_n/x_n}] \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) \\
\supseteq & \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket v \rrbracket_{\delta'}} \downarrow(\pi(\delta'), \mathbf{v})
\end{aligned}
$$

(Apply):

$$
\begin{aligned}
& \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket e_1\ e_2 \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) \\
= & \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \; \bigcup_{\mathbf{a} \in \llbracket e_2 \rrbracket_\delta} \; \bigcup_{\mathbf{f} \in \llbracket e_1 \rrbracket_\delta} \; \bigcup_{\mathbf{v} \in \mathbf{f}(\mathbf{a})} \downarrow(\delta, \mathbf{v}) \\
\supseteq & \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \; \bigcup_{\mathbf{a} \in \llbracket e_2 \rrbracket_{\delta'}} \; \bigcup_{\mathbf{f} \in \llbracket \varphi(y_1,..,y_k) \rrbracket_{\delta'}} \; \bigcup_{\mathbf{v} \in \mathbf{f}(\mathbf{a})} \downarrow(\pi(\delta'), \mathbf{v})
\end{aligned}
$$

Using lemma 4.3.1:

$$
\begin{aligned}
= & \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \; \bigcup_{\mathbf{a} \in \llbracket e_2 \rrbracket_{\delta'}} \; \bigcup_{\mathbf{f} \in \downarrow\left(\lambda\, \mathbf{a}'. \llbracket \varphi(y_1,..,y_k,x) \rrbracket_{\delta'[x \mapsto \mathbf{a}']}\right)} \; \bigcup_{\mathbf{v} \in \mathbf{f}(\mathbf{a})} \downarrow(\pi(\delta'), \mathbf{v}) \\
= & \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \; \bigcup_{\mathbf{a} \in \llbracket e_2 \rrbracket_{\delta'}} \; \bigcup_{\mathbf{v} \in \llbracket \varphi(y_1,..,y_k,x) \rrbracket_{\delta'[x \mapsto \mathbf{a}]}} \downarrow(\pi(\delta'), \mathbf{v}) \\
= & \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket \mathbf{let}\ x = e_2\ \mathbf{in}\ \varphi(y_1,..,y_k,x) \rrbracket_{\delta'}} \downarrow(\pi(\delta'), \mathbf{v}) \\
\supseteq & \bigcup_{\delta'' \in \{\!\{\Delta''\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket v \rrbracket_{\delta''}} \downarrow(\pi(\delta''), \mathbf{v})
\end{aligned}
$$

(Plus)

$$
\begin{aligned}
& \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \; \bigcup_{\mathbf{v} \in \llbracket e_1 + e_2 \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) \\
= & \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \; \bigcup_{\mathbf{n}_1 \in \llbracket e_1 \rrbracket_\delta} \; \bigcup_{\mathbf{n}_2 \in \llbracket e_2 \rrbracket_\delta} \; \bigcup_{\mathbf{v} \in \downarrow(\mathbf{n}_1 +^\perp \mathbf{n}_2)} \downarrow(\delta, \mathbf{v}) \\
\supseteq & \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \; \bigcup_{\mathbf{n}_1 \in \llbracket n_1 \rrbracket_{\delta'}} \; \bigcup_{\mathbf{n}_2 \in \llbracket e_2 \rrbracket_{\delta'}} \; \bigcup_{\mathbf{v} \in \downarrow(\mathbf{n}_1 +^\perp \mathbf{n}_2)} \downarrow(\pi(\delta'), \mathbf{v}) \\
\supseteq & \bigcup_{\delta'' \in \{\!\{\Delta''\}\!\}} \; \bigcup_{\mathbf{n}_1 \in \llbracket n_1 \rrbracket_{\delta''}} \; \bigcup_{\mathbf{n}_2 \in \llbracket n_2 \rrbracket_{\delta''}} \; \bigcup_{\mathbf{v} \in \downarrow(\mathbf{n}_1 +^\perp \mathbf{n}_2)} \downarrow(\pi(\delta''), \mathbf{v})
\end{aligned}
$$

$$= \bigcup_{\delta'' \in \{\!\{\Delta''\}\!\}} \bigcup_{\mathbf{v} \in \llbracket n \rrbracket_{\delta''}} \downarrow(\pi(\delta''), \mathbf{v})$$

Where n is the sum of n_1 and n_2.
(Case$_{\mathsf{Nil}}$)

$$\begin{aligned}
&\bigcup_{\delta \in \{\!\{\Delta\}\!\}} \bigcup_{\mathbf{v} \in \llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\} \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) \\
&= \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \bigcup_{\mathbf{l} \in \llbracket e \rrbracket_\delta} \begin{cases} \bigcup_{\mathbf{v} \in \llbracket e_1 \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) & \mathbf{l} = [] \\ \bigcup_{\mathbf{v} \in \llbracket e_2 \rrbracket_{\delta[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]}} \downarrow(\delta, \mathbf{v}) & \mathbf{l} = \mathbf{h} : \mathbf{t} \\ \{\bot\} & \mathbf{l} = \bot \end{cases} \\
&\supseteq \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \bigcup_{\mathbf{l} \in \llbracket \mathsf{Nil}_\tau \rrbracket_{\delta'}} \begin{cases} \bigcup_{\mathbf{v} \in \llbracket e_1 \rrbracket_{\delta'}} \downarrow(\pi(\delta'), \mathbf{v}) & \mathbf{l} = [] \\ \bigcup_{\mathbf{v} \in \llbracket e_2 \rrbracket_{\delta'[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]}} \downarrow(\pi(\delta'), \mathbf{v}) & \mathbf{l} = \mathbf{h} : \mathbf{t} \\ \{\bot\} & \mathbf{l} = \bot \end{cases} \\
&= \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \bigcup_{\mathbf{v} \in \llbracket e_1 \rrbracket_{\delta'}} \downarrow(\pi(\delta'), \mathbf{v}) \\
&\supseteq \bigcup_{\delta'' \in \{\!\{\Delta''\}\!\}} \bigcup_{\mathbf{v} \in \llbracket v \rrbracket_{\delta''}} \downarrow(\pi(\delta''), \mathbf{v})
\end{aligned}$$

(Case$_{\mathsf{Cons}}$)

$$\begin{aligned}
&\bigcup_{\delta \in \{\!\{\Delta\}\!\}} \bigcup_{\mathbf{v} \in \llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\} \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) \\
&= \bigcup_{\delta \in \{\!\{\Delta\}\!\}} \bigcup_{\mathbf{l} \in \llbracket e \rrbracket_\delta} \begin{cases} \bigcup_{\mathbf{v} \in \llbracket e_1 \rrbracket_\delta} \downarrow(\delta, \mathbf{v}) & \mathbf{l} = [] \\ \bigcup_{\mathbf{v} \in \llbracket e_2 \rrbracket_{\delta[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]}} \downarrow(\delta, \mathbf{v}) & \mathbf{l} = \mathbf{h} : \mathbf{t} \\ \{\bot\} & \mathbf{l} = \bot \end{cases} \\
&\supseteq \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \bigcup_{\mathbf{l} \in \llbracket \mathsf{Cons}_\tau\, y_1\ y_2 \rrbracket_{\delta'}} \begin{cases} \bigcup_{\mathbf{v} \in \llbracket e_1 \rrbracket_{\delta'}} \downarrow(\pi(\delta'), \mathbf{v}) & \mathbf{l} = [] \\ \bigcup_{\mathbf{v} \in \llbracket e_2 \rrbracket_{\delta'[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]}} \downarrow(\pi(\delta'), \mathbf{v}) & \mathbf{l} = \mathbf{h} : \mathbf{t} \\ \{\bot\} & \mathbf{l} = \bot \end{cases} \\
&= \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \bigcup_{\mathbf{v} \in \llbracket e_2 \rrbracket_{\delta'[h \mapsto \delta'(y_1), t \mapsto \delta'(y_2)]}} \downarrow(\pi(\delta'), \mathbf{v}) \\
&= \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} \bigcup_{\mathbf{v} \in \llbracket e_2\ [y_1/h, y_2/t] \rrbracket_{\delta'}} \downarrow(\pi(\delta'), \mathbf{v}) \\
&\supseteq \bigcup_{\delta'' \in \{\!\{\Delta''\}\!\}} \bigcup_{\mathbf{v} \in \llbracket v \rrbracket_{\delta''}} \downarrow(\pi(\delta''), \mathbf{v})
\end{aligned}$$

For the various (Guess)-rules, we use lemma 4.3.3. For literals and constructors without fields, the lemma is sufficient already:

$$\begin{aligned}&\bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}}\ \bigcup_{\mathbf{v}\in[\![\mathbf{unknown}_{\mathsf{Nat}}]\!]_\delta} \downarrow(\delta,\mathbf{v})\\ \supseteq\ &\bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}}\ \bigcup_{\mathbf{v}\in[\![n]\!]_\delta} \downarrow(\delta,\mathbf{v})\end{aligned}$$

For constructors with fields, the induction hypothesis has to be used:

$$\begin{aligned}&\bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}}\ \bigcup_{\mathbf{v}\in[\![\mathbf{unknown}_{[\tau]}]\!]_\delta} \downarrow(\delta,\mathbf{v})\\ \supseteq\ &\bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}}\ \bigcup_{\mathbf{v}\in[\![\mathbf{let}\ h::\tau,t::[\tau]\ \mathbf{free\ in}\ \mathsf{Cons}_\tau(h,t)]\!]_\delta} \downarrow(\delta,\mathbf{v})\\ \supseteq\ &\bigcup_{\delta'\in\{\!\!\{\Delta'\}\!\!\}}\ \bigcup_{\mathbf{v}\in[\![v]\!]_{\delta'}} \downarrow(\pi(\delta'),\mathbf{v})\end{aligned}$$

(Guess) is the only case containing actual estimates (as opposed to set equalities), while all other cases only need estimates where the induction hypothesis is applied.

□

4.5 Adequacy

In this section we prove the following theorem:

Theorem 4.5.1. For every heap expression pair $\Delta : e$ the following holds

$$\bigcup_{\delta\in\{\!\!\{\Delta\}\!\!\}} [\![e]\!]_\delta = \bigcup_{\Delta:e\Downarrow\Delta':v}\ \bigcup_{\delta'\in\{\!\!\{\Delta'\}\!\!\}} [\![v]\!]_{\delta'} \tag{4.4}$$

where the first union on the right hand side is over all operational derivations.

The right hand side is a subset of the left hand side because of the correctness theorem 4.4.1. What remains to be done is constructing suitable derivation trees, which is this section's actual aim.

The corresponding claim in [Launchbury, 1993] is that whenever the denotation is something other than $\bot$, there is a derivation tree in the operational semantics. The proof introduces two auxiliary concepts: A resourced denotational semantics and an alternative operational semantics. At first glance, our approach is the same, but the details differ due to the different nature of the language in question.
Our denotational semantics already is resourced, though not in the same way Launchbury's semantics is. The reason for this deviation will become apparent later, after we have introduced the other auxiliary semantics.
For the duration of the proof, Launchbury replaces the lookup rule with a slightly different version. The original version of the rule looks up a variable in the heap, evaluates

the expression the variable is bound to, and updates the heap afterwards. If the same variable is looked up again, the heap already contains the result and evaluation can proceed immediately. Within the adequacy proof, this rule is replaced by a version that does not update the heap. So, the expression is evaluated again and again every time the variable is looked up. If this rule was used in an implementation, it would make the program much slower.

However, the statement is only concerned with whether evaluation succeeds. For this question it is irrelevant whether the heap is updated: If the expression has been evaluated successfully before, it can be done again taking the same (finite) time and leading to the same result. Yet, if the evaluation diverges, there is no result independently of updates. Therefore, it is fine to alter the operational semantics in this respect to enable a simpler proof.

In our case, the same adjustment would however change the set of possible results. There can be more than one normal form the expression can be evaluated to. Evaluating the expression again for subsequent lookups would break sharing. So, in order to get the correct behavior, the heap has to store the result.

On the other hand, we do not want to deal with updates in the proof either. So, we introduce a version of the operational semantics called *medial semantics* that avoids updates another way. When a new variable is introduced, the expression it is bound to can be evaluated before the variable is added to the heap. However, the expression does not have to be evaluated. Looking up a variable only succeeds if the variable is bound to a value. So, if the variable will be looked up, the expression has to be evaluated before being stored in the heap. This enforces sharing. If the variable will not be looked up, the expression can be stored in unevaluated form. Since possible failure remains unnoticed, this semantics is not strict.

The name *medial* is apt for two reasons. For one thing the medial semantics is an intermediate step between the operational and the denotational semantics. The given rules resemble the rules of the operational semantics and in both cases the existence of a derivation tree proves a value to be a possible result of a program. There is also an important similarity between the medial and the denotational semantics in that (syntactic resp. semantic) values have to be fixed for all heap variables *before* evaluating the expression itself. This leads to the second reason for the name *medial*: When deciding on the value of some variable, there is no indication of whether this variable will be looked up or not. Thus execution of the medial semantics would require predictions about the future use (or non-use) of variables.

4.5.1 Medial Semantics

The medial semantics is a variant of the operational semantics shown in figure 4.8 subject to some changes. The (Lookup) rule is replaced by the one shown in figure 4.12 and can only be used to look up variables bound to values. Thus when using (Lookup) no update of the heap is necessary.

Two new rules, (Let!) and (Free!), are added and allow to evaluate expressions before writing the result into the heap. The rules (Let) and (Free) are however still part of the

(Lookup) $\Delta\,[x :: \tau \mapsto v]\,\Omega : x \Downarrow \Delta\,[x :: \tau \mapsto v]\,\Omega : v$ if v is a flat normal form

(Let!) $\dfrac{\Delta : e_1 \Downarrow \Delta' : v_1 \qquad \Delta'\,[y :: \tau_1 \mapsto v_1] : e\,[y/x] \Downarrow \Delta'' : v}{\Delta : \textbf{let } x = e_1 \textbf{ in } e \Downarrow \Delta'' : v}$ y fresh and τ_1 is the type of $\Delta : e_1$

(Free!) $\dfrac{\Delta : \textbf{unknown}_{\tau_1} \Downarrow \Delta' : v_1 \qquad \Delta'\,[y :: \tau_1 \mapsto v_1] : e\,[y/x] \Downarrow \Delta'' : v}{\Delta : \textbf{let } x :: \tau_1 \textbf{ free in } e \Downarrow \Delta'' : v}$ y fresh

Figure 4.12: Replacement and additional rules for the medial semantics for CuMin

semantics as well, so they can be used to add the unevaluated expression to the heap. Unless the expression happens to be a value, the affected variable can never be looked up.

Lemma 4.5.2. If there is a medial derivation for $\Delta : e \Downarrow \Delta'' : v$, then there is an operational derivation for $\Delta : e \Downarrow \Delta' : v$ and Δ'' emerges from Δ' by forcing evaluation of heap entries.

Proof. Let some medial derivation tree be given. Since the (Lookup) rule has been altered to not update the heap, heap entries can only be added but never changed. Also, new entries are always added to the end of the heap. Therefore, the order in which the variables appear in the heap is the order in which they are added during the derivation. We change the derivation tree variable by variable in the reversed heap order. In the course of this process, the derivation is neither operational nor medial, but can use both versions of (Lookup), (Let) and (Free).
Let y be such a heap variable. If y appears in the current derivation, it has to be added to the heap by some use of (Let), (Let!), (Free) or (Free!). If y is introduced using (Free), it is bound to **unknown** in the heap and can never be looked up in a medial derivation. In this case, no change is necessary.
If y is introduced using (Let), we have to distinguish further. If y is not bound to a value, it can never be looked up and no change is necessary (as in the previous case). If y happens to be bound to a value, lookups can occur. Since the medial (Lookup) rule does not have a premise, but the operational (Lookup) rule does, (Val) has to be added everywhere. Thus the medial (Lookup)

$$\Delta\,[y :: \tau_1 \mapsto v_1]\,\Omega : y \Downarrow \Delta\,[y :: \tau_1 \mapsto v_1]\,\Omega : v_1$$

is replaced by an operational (Lookup) and an application of (Val):

$$\frac{\Delta : v_1 \Downarrow \Delta : v_1}{\Delta\,[y :: \tau_1 \mapsto v_1]\,\Omega : y \Downarrow \Delta\,[y :: \tau_1 \mapsto v_1]\,\Omega : v_1}$$

Otherwise, y is introduced using (Let!) or (Free!) and there is a subtree T_y proving the first premise of the rule introducing y.

$$\begin{array}{c} \vdots \\ \hline T_y \qquad \Delta' \,[y :: \tau_1 \mapsto v_1] : e\,[y/x] \Downarrow \Delta' \,[y : \tau_1 \mapsto v_1]\,\Omega' : v \\ \hline \Delta : \mathbf{let}\ x = e_1\ \mathbf{in}\ e \Downarrow \Delta' \,[y :: \tau_1 \mapsto v_1]\,\Omega' : v \\ \hline \vdots \end{array}$$

We remove this subtree and remember it for later use. The derivation is temporarily incorrect:

$$\begin{array}{c} \vdots \\ \hline \Delta' \,[y :: \tau_1 \mapsto v_1] : e\,[y/x] \Downarrow \Delta' \,[y : \tau_1 \mapsto v_1]\,\Omega' : v \\ \hline \Delta : \mathbf{let}\ x = e_1\ \mathbf{in}\ e \Downarrow \Delta' \,[y :: \tau_1 \mapsto v_1]\,\Omega' : v \\ \hline \vdots \end{array}$$

In this situation it is possible for the variable y to be looked up in the medial derivation. If y still never is looked up, we replace the prefix $\Delta' \,[y :: \tau_1 \mapsto v_1]$ of the heap by $\Delta \,[y :: \tau_1 \mapsto e_1]$ everywhere in the derivation. This makes the rule introducing y a correct use of (Let) or (Free):

$$\begin{array}{c} \vdots \\ \hline \Delta \,[y :: \tau_1 \mapsto e_1] : e\,[y/x] \Downarrow \Delta \,[y : \tau_1 \mapsto e_1]\,\Omega' : v \\ \hline \Delta : \mathbf{let}\ x = e_1\ \mathbf{in}\ e \Downarrow \Delta \,[y :: \tau_1 \mapsto e_1]\,\Omega' : v \\ \hline \vdots \end{array}$$

In this case the subtree T_y is not part of the resulting operational derivation and neither are the heap entries introduced in T_y.
Otherwise, there is at least one lookup of y. The first lookup receives T_y as its premise and evaluates e_1 to v_1 in the operational derivation. Between the rule introducing y and the first lookup, we replace the prefix $\Delta' \,[y :: \tau_1 \mapsto v_1]$ of the heap by $\Delta \,[y :: \tau_1 \mapsto e_1]$. Thus the introducing rule does not change the heap, but the first lookup does:

$$\begin{array}{c} \vdots \\ \hline \Delta : e_1 \Downarrow \Delta' : v_1 \\ \hline \Delta \,[y :: \tau \mapsto e_1]\,\Omega : y \Downarrow \Delta' \,[y :: \tau \mapsto v_1]\,\Omega : v_1 \\ \hline \vdots \end{array}$$

All subsequent lookups get (Val) as their premise:

$$\begin{array}{c} \Delta' : v_1 \Downarrow \Delta' : v_1 \\ \hline \Delta' \,[y :: \tau \mapsto v_1]\,\Omega' : y \Downarrow \Delta' \,[y :: \tau \mapsto v_1]\,\Omega' : v_1 \\ \hline \vdots \end{array}$$

□

4.5.2 Existence of Medial Derivations

The next lemma allows to prove the existence of medial derivations for given semantic values.

Lemma 4.5.3. Let $\Delta : e$ be a heap expression pair. Let $\delta \in \{\!\{\Delta\}\!\}$, such that $\delta(y) = \bot$ unless $\Delta(y)$ is a value. Let $\mathbf{v}$ be such that $\bot \neq \mathbf{v} \in [\![e]\!]^t_\delta$. There exists a derivable $\Delta : e \Downarrow \Delta\ \Omega : v$ and $\omega \in \{\!\{\Omega\}\!\}_\delta$ such that $\omega(y) = \bot$ unless $\Omega(y)$ is a value and $\mathbf{v} \in [\![v]\!]^1_{\delta\omega}$.

Proof. The proof is by induction on t, where in the base case $t = 0$ there is nothing to show because $[\![e]\!]^0 = \{\bot\}$ does not contain any $\mathbf{v} \neq \bot$. So for positive t we may assume the claim to be valid for all smaller t and proceed by splitting cases according to the expression e.

If the expression is a flat normal form, use (Val) to derive $\Delta : v \Downarrow \Delta : v$. The denotation of a value is independent of the step index, which follows directly from the definition of the denotational semantics (figure 4.10). So $\mathbf{v} \in [\![\Delta : v]\!]^{t+1} = [\![\Delta : v]\!]^1$ indeed holds.

If the expression is a variable x and $\Delta(x)$ is not a value, then $\delta(x) = \bot$ and therefore $[\![x]\!]^{t+1}_\delta = \{\bot\}$. Since there is no $\bot \neq \mathbf{v} \in [\![e]\!]^{t+1}_\delta$ the claim is true. If the expression is a variable x and $\Delta(x)$ is a value v we can use (Lookup). Since v is a value, $\mathbf{v} \in [\![v]\!]^1_\delta = [\![v]\!]^{t+1}_\delta = [\![\Delta(x)]\!]^{t+1}_\delta$.

Let $\Delta : \mathbf{let}\ x = e_1\ \mathbf{in}\ e$, δ and $\mathbf{v} \in [\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e]\!]^{t+1}_\delta$ be given.

$$\begin{aligned}\mathbf{v} &\in [\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e]\!]^{t+1}_\delta \\ &= \bigcup_{\mathbf{x} \in [\![e_1]\!]^t_\delta} [\![e]\!]^t_{\delta[x \mapsto \mathbf{x}]}\end{aligned}$$

Choose a suitable $\mathbf{x}$. If $\mathbf{x} = \bot$, apply the induction hypothesis to $\Delta\ [y \mapsto e_1] : e\ [y/x]$, $\delta[y \mapsto \bot]$ and $\mathbf{v}$ and use (Let). If $\mathbf{x} \neq \bot$, apply the induction hypothesis to $\Delta : e_1$, δ and $\mathbf{x}$ to get $\Delta' : v_1$ and δ'. Then apply the induction hypothesis again to $\Delta'\ [y \mapsto v_1] : e\ [y/x]$, $\delta[y \mapsto \mathbf{x}]$ and $\mathbf{v}$ and use (Let!).

Let $\Delta : \mathbf{failed}_\tau$, δ and $\mathbf{v} \in [\![\mathbf{failed}_\tau]\!]^{t+1}_\delta$ be given. Since $\mathbf{v} \in [\![\mathbf{failed}_\tau]\!]^{t+1}_\delta = \{\bot\}$ we have $\mathbf{v} = \bot$ and there is nothing to show.

Let $\Delta : e_1\ e_2$, δ and $\mathbf{v} \in [\![e_1\ e_2]\!]^{t+2}_\delta$ be given.

$$\begin{aligned}\mathbf{v} &\in [\![e_1\ e_2]\!]^{t+2}_\delta \\ &= \bigcup_{\mathbf{f} \in [\![e_1]\!]^{t+1}_\delta} \bigcup_{\mathbf{a} \in [\![e_2]\!]^{t+1}_\delta} \mathbf{f}(\mathbf{a}, t)\end{aligned}$$

Choose a suitable $\mathbf{f}$ and apply the induction hypothesis to $\Delta : e_1$, δ and $\mathbf{f} \in [\![e_1]\!]^{t+1}_\delta$ to get $\Delta : e_1 \Downarrow \Delta' : \varphi(y_1, \ldots, y_k)$, δ' and $\mathbf{f} \in [\![\varphi(y_1, \ldots, y_k)]\!]^1_{\delta'} = {\downarrow}\lambda\, \mathbf{a}'\ t'.[\![\varphi(y_1, \ldots, y_k, x)]\!]^{t'+1}_{\delta'[x \mapsto \mathbf{a}']}$ (using lemma 4.3.1).

$$\mathbf{v} \in \bigcup_{\mathbf{a} \in [\![e_2]\!]^{t+1}_{\delta'}} \mathbf{f}(\mathbf{a}, t)$$

$$\begin{aligned}
&\subseteq \bigcup_{\mathbf{a} \in [\![e_2]\!]^{t+1}_{\delta'}} \left(\lambda\, \mathbf{a}'\ t'. [\![\varphi(y_1, \ldots, y_k, x)]\!]^{t'+1}_{\delta'[x \mapsto \mathbf{a}']}\right) (\mathbf{a}, t) \\
&= \bigcup_{\mathbf{a} \in [\![e_2]\!]^{t+1}_{\delta'}} [\![\varphi(y_1, \ldots, y_k, x)]\!]^{t+1}_{\delta'[x \mapsto \mathbf{a}]} \\
&= [\![\mathbf{let}\ x = e_2\ \mathbf{in}\ \varphi(y_1, \ldots, y_k, x)]\!]^{t+2}_{\delta'}
\end{aligned}$$

Apply the induction hypothesis to $\Delta' : \mathbf{let}\ x = e_2\ \mathbf{in}\ \varphi(y_1, \ldots, y_k, x)$, δ' and $\mathbf{v} \in [\![\mathbf{let}\ x = e_2\ \mathbf{in}\ \varphi(y_1, \ldots, y_k, x)]\!]^{t+2}_{\delta'}$. Even though the index t did not decrease, this application of the induction hypothesis is fine, as we have already proved the **let** . . **in** case.

Let $\Delta : f_{\overline{\tau_m}}(y_1, . . , y_n)$ with f n-ary, δ and $\mathbf{v} \in [\![f_{\overline{\tau_m}}(y_1, . . , y_n)]\!]^{t+1}_{\delta}$ be given.

$$\begin{aligned}
\mathbf{v} &\in [\![f_{\overline{\tau_m}}(y_1, . . , y_n)]\!]^{t+1}_{\delta} \\
&= [\![e]\!]^{t}_{\overline{[\alpha_m \mapsto [\![\tau_m]\!]]}\,\overline{[x_n \mapsto \delta(y_n)]}} \\
&= [\![e\ [\overline{\tau_m/\alpha_m}, \overline{y_n/x_n}]]\!]^{t}_{\delta}
\end{aligned}$$

So we can apply the induction hypothesis and use (Fun).

Let $\Delta : e_1 + e_2$, δ and $\mathbf{v} \in [\![e_1 + e_2]\!]^{t+1}_{\delta}$ be given.

$$\begin{aligned}
\mathbf{v} &\in [\![e_1 + e_2]\!]^{t+1}_{\delta} \\
&= \bigcup_{\mathbf{n}_1 \in [\![e_1]\!]^{t}_{\delta}} \bigcup_{\mathbf{n}_2 \in [\![e_2]\!]^{t}_{\delta}} {\downarrow}(\mathbf{n}_1 +^{\perp} \mathbf{n}_2)
\end{aligned}$$

Choose a suitable $\mathbf{n}_1$. Apply the induction hypothesis to $\Delta : e_1$, δ and $\mathbf{n}_1 \in [\![e_1]\!]^{t}_{\delta}$ to get $\Delta : e_1 \Downarrow \Delta' : n_1$, δ' and $\mathbf{n}_1 \in [\![n_1]\!]^{1}_{\delta'}$.

$$\begin{aligned}
\mathbf{v} &\in \bigcup_{\mathbf{n}_2 \in [\![e_2]\!]^{t}_{\delta}} {\downarrow}(\mathbf{n}_1 +^{\perp} \mathbf{n}_2) \\
&= \bigcup_{\mathbf{n}_2 \in [\![e_2]\!]^{t}_{\delta'}} {\downarrow}(\mathbf{n}_1 +^{\perp} \mathbf{n}_2)
\end{aligned}$$

Choose a suitable $\mathbf{n}_2$. Apply the induction hypothesis to $\Delta' : e_2$, δ' and $\mathbf{n}_2 \in [\![e_2]\!]^{t}_{\delta'}$ to get $\Delta' : e_2 \Downarrow \Delta'' : n_2$, δ'' and $\mathbf{n}_2 \in [\![n_2]\!]^{1}_{\delta''}$.

$$\begin{aligned}
\mathbf{v} &\in {\downarrow}(\mathbf{n}_1 +^{\perp} \mathbf{n}_2) \\
&= [\![n]\!]^{1}_{\delta''}
\end{aligned}$$

So we can derive $\Delta : e_1 + e_2 \Downarrow \Delta'' : n$ where n is the sum of n_1 and n_2 and $\mathbf{v} \in [\![n]\!]^{1}_{\delta''}$.

Let $\Delta : \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\}$, δ and $\mathbf{v} \in [\![\mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\}]\!]^{t+1}_{\delta}$ be given.

$$\mathbf{v} \in [\![\mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\}]\!]^{t+1}_{\delta}$$

$$= \bigcup_{\mathbf{l} \in [\![e]\!]_\delta^t} \begin{cases} [\![e_1]\!]_\delta^t & \mathbf{l} = [] \\ [\![e_2]\!]_{\delta[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]}^t & \mathbf{l} = \mathbf{h} : \mathbf{t} \\ \{\bot\} & \mathbf{l} = \bot \end{cases}$$

Choose a suitable $\mathbf{l}$. Apply the induction hypothesis to $\Delta : e$, δ and $\mathbf{l} \in [\![e]\!]_\delta^t$ to get $\Delta : e \Downarrow \Delta' : v'$, δ' and $\mathbf{l} \in [\![v']\!]_{\delta'}^1$. If v' is Nil_τ, we have $\mathbf{l} \in [\![\mathsf{Nil}_\tau]\!]_{\delta'}^1 = {\downarrow}[]$.

$$\begin{aligned} \mathbf{v} &\in [\![e_1]\!]_\delta^t \\ &= [\![e_1]\!]_{\delta'}^t \end{aligned}$$

Apply the induction hypothesis to $\Delta' : e_1$, δ' and $\mathbf{v} \in [\![e_1]\!]_{\delta'}^t$ to get $\Delta' : e_1 \Downarrow \Delta'' : v$, δ'' and $\mathbf{v} \in [\![v]\!]_{\delta''}^1$.
If v' is $\mathsf{Cons}_\tau\,(y_1, y_2)$, we have $\mathbf{l} = \mathbf{h} : \mathbf{t}$ with $\mathbf{h} = \delta'\ y_1$ and $\mathbf{t} = \delta'\ y_2$ and

$$\begin{aligned} \mathbf{v} &\in [\![e_2]\!]_{\delta[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]}^t \\ &= [\![e_2]\!]_{\delta'[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]}^t \\ &= [\![e_2\ [y_1/h, y_2/t]]\!]_{\delta'}^t \end{aligned}$$

Apply the induction hypothesis to $\Delta' : e_2\ [y_1/h, y_2/t]$, δ' and $\mathbf{v} \in [\![e_2\ [y_1/h, y_2/t]]\!]_{\delta'}^t$ to get $\Delta' : e_2\ [y_1/h, y_2/t] \Downarrow \Delta'' : v$, δ'' and $\mathbf{v} \in [\![v]\!]_{\delta''}^1$.
In either case we can derive $\Delta : \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\} \Downarrow \Delta'' : v$ and $\mathbf{v} \in [\![v]\!]_{\delta''}^1$.
Let $\Delta : \textbf{unknown}_\tau$, δ and $\mathbf{v} \in [\![\textbf{unknown}_\tau]\!]_\delta^{t+1}$ be given. We split cases again, this time for the type τ, which is a data type. We will not go through all of the cases, but just do lists as an example.

$$\begin{aligned} \mathbf{v} &\in [\![\textbf{unknown}_{[\tau]}]\!]_\delta^{t+1} \\ &= {\downarrow}[] \cup \bigcup_{\mathbf{h} \in [\![\textbf{unknown}_\tau]\!]_\delta^t} \bigcup_{\mathbf{t} \in [\![\textbf{unknown}_{[\tau]}]\!]_\delta^t} {\downarrow}(\mathbf{h} : \mathbf{t}) \\ &= [\![\mathsf{Nil}_\tau]\!]_\delta^{t+1} \cup [\![\textbf{let}\ h :: \tau, t :: [\tau]\ \textbf{free in}\ \mathsf{Cons}_\tau\,(h, t)]\!]_\delta^{t+1} \end{aligned}$$

So $\mathbf{v}$ has to be contained in one of the sets and depending on which one that is, we can pick the appropriate guessing rule.

□

Proof of theorem 4.5.1. Theorem 4.4.1 implies the "⊇" direction. For the other direction, let some $\mathbf{v} \in \bigcup_{\delta \in \{\!\{\Delta\}\!\}} [\![e]\!]_\delta$ be given. Lemma 4.5.3 implies that there is a medial derivation for some $\Delta : e \Downarrow \Delta'' : v$, such that $\mathbf{v} \in \bigcup_{\delta'' \in \{\!\{\Delta''\}\!\}} [\![v]\!]_{\delta''}$. Using lemma 4.5.2 we can conclude that there is an operational derivation for some $\Delta : e \Downarrow \Delta' : v$. Again using 4.4.1,

$$\mathbf{v} \in \bigcup_{\delta'' \in \{\!\{\Delta''\}\!\}} [\![v]\!]_{\delta''} \subseteq \bigcup_{\delta' \in \{\!\{\Delta'\}\!\}} [\![v]\!]_{\delta'}$$

holds, since in Δ'' more entries are evaluated than in Δ'. □

4.6 Translating CuMin into SaLT

We have given a denotational semantics for CuMin and could thus start to derive results about CuMin programs by arguing about their denotational semantics. However, it is much simpler to argue on a syntactic level, i.e., by equational reasoning. For example in chapter 3, we have never left the syntactic level.
Equational reasoning in CuMin and Curry is difficult because many of the steps we are used to performing in other language are not valid here, i.e., do not preserve the semantics. Many of the steps in chapter 3 simply were evaluation steps (forwards or backwards), like applying defining rules for functions. In Curry and CuMin this is problematic because evaluation can branch. Other commonly used steps like beta or eta equivalence do not hold in Curry and CuMin, either.
Like in [Mehner et al., 2014], we introduce an intermediate step halfway between CuMin and its denotational semantics: the language SaLT. The acronym is short for *sets and lambda terms* and that is exactly what SaLT is – a lambda calculus with set types. SaLT itself is deterministic, but CuMin can be translated into SaLT, making the nondeterminism explicit by using sets on the syntactic level. These sets have a monadic interface, in which bind corresponds to taking unions and return corresponds to building singleton sets. In [Mehner et al., 2014], the bind operator is even denoted as a union, but we decided to stick to the Haskell notation ($\gg\!=$) here. The translation from CuMin to SaLT will introduce monadic binds wherever the denotational semantics uses unions, thus making the interesting part of the semantics visible. Yet we stay on a syntactic level and do not have to deal with technicalities like switching back and forth between syntactic and semantic variables or even with keeping track of the step indexes. These matters are taken care of by the denotational semantics of SaLT, which is very straightforward since SaLT is a deterministic typed lambda calculus, like e.g., Haskell.

There is a second reason why we need SaLT, which will become important later. The free theorems [Wadler, 1989] we will prove for CuMin have side conditions restricting nondeterminism. Such conditions are hard to express in CuMin because it is intrinsically nondeterministic. Christiansen et al. [2010] show ways to still formulate these conditions using CuMin (or its predecessor) only. Yet we find it easier to have a language that gives an easy notion of determinism. SaLT does just that because it has singleton sets which are clearly deterministic and can be compared with other expressions.

4.6.1 The Language SaLT

We start by describing the syntax, type system and denotational semantics, all of which are very similar to CuMin. Therefore we only point out the differences and refer back to CuMin a lot.
The full syntax of SaLT is given in figure 4.13. The most important change compared to CuMin is the new type constructor Set and its monadic interface $\{\,\}$ and ($\gg\!=$). Unlike in [Mehner et al., 2014], the bind operator does not automatically introduce a variable

$$
\begin{array}{lll}
P & ::= & D; P \mid D \\
D & ::= & f :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau; f\ \overline{x_n} = e \\
\tau & ::= & \alpha \mid \tau \to \tau' \mid \mathsf{Bool} \mid \mathsf{Nat} \mid [\tau] \mid (\tau, \tau') \mid \mathsf{Set}\ \tau \\
e & ::= & x \\
 & & \mid \mathbf{failed}_\tau \mid f\ \overline{\tau_m} \mid \lambda(x :: \tau) \to e \mid e_1\ e_2 \\
 & & \mid n \mid e_1 + e_2 \mid e_1 == e_2 \\
 & & \mid \mathsf{True} \mid \mathsf{False} \mid \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{True} \to e_1; \mathsf{False} \to e_2\} \\
 & & \mid \mathsf{Nil}_\tau \mid \mathsf{Cons}_\tau \mid \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\} \\
 & & \mid \mathsf{Pair}_{\tau,\tau'} \mid \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Pair}\ l\ r \to e_1\} \\
 & & \mid \{e\} \mid e_1 \ggg= e_2 \mid \mathbf{unknown}_\tau
\end{array}
$$

Figure 4.13: Syntax of SaLT

$$
\frac{\Gamma, x :: \tau \vdash e :: \tau'}{\Gamma \vdash \lambda(x :: \tau) \to e :: \tau \to \tau'} \qquad \frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \{e\} :: \mathsf{Set}\ \tau}
$$

$$
\frac{\Gamma \vdash e_1 :: \mathsf{Set}\ \tau \qquad \Gamma \vdash e_2 :: \tau \to \mathsf{Set}\ \tau'}{\Gamma \vdash e_1 \ggg= e_2 :: \mathsf{Set}\ \tau'} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash \mathbf{unknown}_\tau :: \mathsf{Set}\ \tau}
$$

Figure 4.14: Typing rules for SaLT

name. Instead, lambda abstractions can be used (which were also part of the version of SaLT given in [Mehner et al., 2014] though for a different reason).
Finally, there is an **unknown** primitive, which represents the set of all values of some type and is thus closely related to the **unknown** we introduced when discussing the semantics of CuMin. This primitive will allow to create non-trivial sets in the first place.
Unlike in [Mehner et al., 2014], functions can be defined by rules, i.e., giving arguments on the left hand side. This corresponds to a change in the translation process, so we defer the discussion concerning this change.

The type system of SaLT uses the same kinds of judgments we have already seen for CuMin. The rules for building contexts given in figure 4.3 remain untouched. The $\tau \in \mathsf{Type}$ judgments are derived by the rules in the same figure or using the new rule

$$
\frac{\Gamma \vdash \tau \in \mathsf{Type}}{\Gamma \vdash \mathsf{Set}\ \tau \in \mathsf{Type}}
$$

for set types. These types are not members of the Data class, so figure 4.4 remains valid. Finally and most importantly, expressions are typed by the rules in figure 4.5 and the additional rules in figure 4.14. In particular, we want to point out that the type of $\mathbf{unknown}_\tau$ is $\mathsf{Set}\ \tau$ in SaLT, while it is τ in CuMin.
Because of these rules, **unknown** cannot be used to create nested sets. Sets are used as a data structure in many languages, so not including them in the Data class might seem surprising.

$$\begin{aligned}
[\![\alpha]\!]_\theta &= [\![\theta(\alpha)]\!] \\
[\![\tau \to \tau']\!]_\theta &= \{\mathbf{f} : [\![\tau]\!]_\theta \times \mathbb{N} \to [\![\tau']\!]_\theta \mid \mathbf{f} \text{ monotone}\} \\
[\![\mathsf{Bool}]\!]_\theta &= \{\mathbf{True}, \mathbf{False}\}_\perp \\
[\![\mathsf{Nat}]\!]_\theta &= \mathbb{N}_\perp \\
[\![[\tau]]\!]_\theta &= \{\mathbf{x}_1 : \ldots : \mathbf{x}_n : \mathbf{e} \mid n \geq 0, \mathbf{x}_i \in [\![\tau]\!]_\theta, \mathbf{e} \in \{\perp, []\}\} \\
[\![(\tau, \tau')]\!]_\theta &= \{(\mathbf{l}, \mathbf{r}) \mid \mathbf{l} \in [\![\tau]\!]_\theta, \mathbf{r} \in [\![\tau']\!]_\theta\}_\perp \\
[\![\mathsf{Set}\ \tau]\!]_\theta &= \mathcal{L}\left([\![\tau]\!]_\theta\right)
\end{aligned}$$

Figure 4.15: Denotational type semantics for SaLT

There however is a difference between the Set type constructor in Haskell and the Set type constructor in SaLT. In most languages, sets are finite collections of elements, while sets in SaLT can be infinite (like **unknown**$_{\mathsf{Nat}}$, the set of all natural numbers). Thus when the result of a computation is an infinite set, tools cannot list all the elements. However, tools can *start* to list elements and every element will be found after a finite amount of time because there is a way to generate all of them. The same is not true for nested sets: There is an uncountable number of subsets of $\mathbb{N}$, so there is no program that generates all subsets one after the other.

The denotational type semantics of SaLT is given in figure 4.15 and is very similar to the type semantics of CuMin (figure 4.9). The semantics of a type with respect to an environment is a poset (pointed partially ordered set) and the constructions coincide for all Data types. The domain for function types does not rely on the lower subsets construction $\mathcal{L}\left(\right)$ anymore. Instead, it appears in the definition of the semantics of set types.

Given some typing judgment $\Gamma \vdash e :: \tau$, a type environment θ, a term environment σ and a stepindex $t \in \mathbb{N}$, the term level semantics $[\![e]\!]^t_{\theta,\sigma}$ is an element of $[\![\tau]\!]_\theta$. For all expressions $[\![e]\!]^0_{\theta,\sigma}$ is defined to be $\perp$. For positive t, the semantics is given by the equations in figure 4.16. The semantics is by construction monotone in both t and σ.

Unlike in CuMin, the limit $[\![e]\!]^\infty_{\theta,\sigma}$ does not exist in general, but we will use this notation if it does. A limit exists if the sequence $[\![e]\!]^t_{\theta,\sigma}$ becomes stable, i.e., there is some t_0, such that for all $t \geq t_0$ the value is independent of t. Another sufficient condition for the existence of a limit is e having a set type Set τ. In this case

$$[\![e]\!]^\infty_{\theta,\sigma} = \bigcup_{t \in \mathbb{N}} [\![e]\!]^t_{\theta,\sigma}$$

holds, where the union combines (lower) subsets of $[\![\tau]\!]_\theta$. If the semantics of some type τ permits taking limits, then so does the semantics of $\tau' \to \tau$ by taking the point-wise limit. Combining this insight with the previous one, all types $\tau_1 \to \ldots \to \tau_n \to \mathsf{Set}\ \tau$ permit taking limits.

$$
\begin{aligned}
\llbracket x \rrbracket^{t+1}_{\theta,\sigma} &= \sigma(x) & \llbracket \mathsf{False} \rrbracket^{t+1}_{\theta,\sigma} &= \mathbf{False} \\
\llbracket \mathbf{failed}_{\,\tau} \rrbracket^{t+1}_{\theta,\sigma} &= \bot & \llbracket \mathsf{Nil}_{\,\tau} \rrbracket^{t+1}_{\theta,\sigma} &= [] \\
\llbracket n \rrbracket^{t+1}_{\theta,\sigma} &= \mathbf{n} & \llbracket \mathsf{Cons}_{\,\tau} \rrbracket^{t+1}_{\theta,\sigma} &= (\lambda\, \mathbf{h}\ t_1.(\lambda\, \mathbf{t}\ t_2.(\mathbf{h} : \mathbf{t}))) \\
\llbracket \mathsf{True} \rrbracket^{t+1}_{\theta,\sigma} &= \mathbf{True} & \llbracket \mathsf{Pair}_{\,\tau,\tau'} \rrbracket^{t+1}_{\theta,\sigma} &= (\lambda\, \mathbf{l}\ t_1.(\lambda\, \mathbf{r}\ t_2.(\mathbf{l}, \mathbf{r})))
\end{aligned}
$$

$$
\llbracket f\ \overline{\tau_m} \rrbracket^{t_0+1}_{\theta,\sigma} = \Big(\lambda\, \mathbf{a}_1\ t_1. \cdots \Big(\lambda\, \mathbf{a}_n\ t_n. \llbracket e \rrbracket^{t_n}_{\overline{[\alpha_m \mapsto \tau_m\theta]},\sigma'}\Big) \cdots\Big)
$$

with $f :: \forall \alpha_1 \mathinner{..} \alpha_m.\tau; f\ x_1 \mathinner{..} x_n = e$ in P and $\sigma' = [x_1 \mapsto \mathbf{a}_1, \ldots, x_n \mapsto \mathbf{a}_n]$

$$
\begin{aligned}
\llbracket \lambda(x :: \tau) \to e \rrbracket^{t+1}_{\theta,\sigma} &= \lambda\, \mathbf{x}\ t'. \llbracket e \rrbracket^{t'+1}_{\theta,\sigma[x \mapsto \mathbf{x}]} \\
\llbracket e_1\ e_2 \rrbracket^{t+2}_{\theta,\sigma} &= \llbracket e_1 \rrbracket^{t+1}_{\theta,\sigma} \Big(\llbracket e_2 \rrbracket^{t+1}_{\theta,\sigma}, t\Big) \\
\llbracket e_1 + e_2 \rrbracket^{t+1}_{\theta,\sigma} &= \llbracket e_1 \rrbracket^{t}_{\theta,\sigma} +^{\bot} \llbracket e_2 \rrbracket^{t}_{\theta,\sigma} \\
\llbracket e_1 == e_2 \rrbracket^{t+1}_{\theta,\sigma} &= \llbracket e_1 \rrbracket^{t}_{\theta,\sigma} =^{\bot} \llbracket e_2 \rrbracket^{t}_{\theta,\sigma} \\
\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{True} \to e_1; \mathsf{False} \to e_2\} \rrbracket^{t+1}_{\theta,\sigma} &= \begin{cases} \llbracket e_1 \rrbracket^{t}_{\theta,\sigma} & \llbracket e \rrbracket^{t}_{\theta,\sigma} = \mathbf{True} \\ \llbracket e_2 \rrbracket^{t}_{\theta,\sigma} & \llbracket e \rrbracket^{t}_{\theta,\sigma} = \mathbf{False} \\ \bot & \llbracket e \rrbracket^{t}_{\theta,\sigma} = \bot \end{cases} \\
\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2\} \rrbracket^{t+1}_{\theta,\sigma} &= \begin{cases} \llbracket e_1 \rrbracket^{t}_{\theta,\sigma} & \llbracket e \rrbracket^{t}_{\theta,\sigma} = [] \\ \llbracket e_2 \rrbracket^{t}_{\theta,\sigma[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]} & \llbracket e \rrbracket^{t}_{\theta,\sigma} = \mathbf{h} : \mathbf{t} \\ \bot & \llbracket e \rrbracket^{t}_{\theta,\sigma} = \bot \end{cases} \\
\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Pair}\ l\ r \to e_1\} \rrbracket^{t+1}_{\theta,\sigma} &= \begin{cases} \llbracket e_1 \rrbracket^{t}_{\theta,\sigma[l \mapsto \mathbf{l}, r \mapsto \mathbf{r}]} & \llbracket e \rrbracket^{t}_{\theta,\sigma} = (\mathbf{l}, \mathbf{r}) \\ \bot & \llbracket e \rrbracket^{t}_{\theta,\sigma} = \bot \end{cases} \\
\llbracket \{\, e \,\} \rrbracket^{t+1}_{\theta,\sigma} &= {\downarrow} \llbracket e \rrbracket^{t}_{\theta,\sigma} \\
\llbracket e_1 \gg\!= e_2 \rrbracket^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{x} \in \llbracket e_1 \rrbracket^{t}_{\theta,\sigma}} \llbracket e_2 \rrbracket^{t}_{\theta,\sigma}(\mathbf{x}, t) \\
\llbracket \mathbf{unknown}_{\,\alpha} \rrbracket^{t+1}_{\theta,\sigma} &= \Big\llbracket \mathbf{unknown}_{\,\theta(\alpha)} \Big\rrbracket^{t+1}_{\theta,\sigma} \\
\llbracket \mathbf{unknown}_{\,\mathsf{Bool}} \rrbracket^{t+1}_{\theta,\sigma} &= \llbracket \mathsf{Bool} \rrbracket \\
\llbracket \mathbf{unknown}_{\,\mathsf{Nat}} \rrbracket^{t+1}_{\theta,\sigma} &= \llbracket \mathsf{Nat} \rrbracket \\
\Big\llbracket \mathbf{unknown}_{\,[\tau]} \Big\rrbracket^{t+1}_{\theta,\sigma} &= {\downarrow}[] \cup \bigcup_{\mathbf{h} \in \llbracket \mathbf{unknown}_{\tau} \rrbracket^{t}_{\theta,\sigma}} \bigcup_{\mathbf{t} \in \Big\llbracket \mathbf{unknown}_{[\tau]} \Big\rrbracket^{t}_{\theta,\sigma}} {\downarrow}(\mathbf{h} : \mathbf{t}) \\
\Big\llbracket \mathbf{unknown}_{\,(\tau,\tau')} \Big\rrbracket^{t+1}_{\theta,\sigma} &= \bigcup_{\mathbf{l} \in \llbracket \mathbf{unknown}_{\tau} \rrbracket^{t}_{\theta,\sigma}} \bigcup_{\mathbf{r} \in \llbracket \mathbf{unknown}_{\tau'} \rrbracket^{t}_{\theta,\sigma}} {\downarrow}(\mathbf{l}, \mathbf{r})
\end{aligned}
$$

Figure 4.16: Denotational term semantics for SaLT with step indexes

4.6.2 Semantic Equivalence and Equational Reasoning

To do equational reasoning, we need a notion of semantic equivalence, i.e., a formalization of two expressions e and e' being exchangeable. Requiring $[\![e]\!]^t_{\theta,\sigma} = [\![e']\!]^t_{\theta,\sigma}$ to hold for all t would be far too restrictive because indexes are often shifted. For example $2+2$ and 4 would not be semantically equivalent, because evaluating the sum takes up one time step. On the other hand, we cannot use

$$[\![e]\!]^\infty_{\theta,\sigma} = [\![e']\!]^\infty_{\theta,\sigma}$$

as a condition, because the limit does not exist in general. Yet if the limit does exist, the above is exactly what we want. The following turns out to be apt:

Definition 4.6.1. Let e and e' be two SaLT expressions that are typeable to some common type in some context. The two expressions are *semantically equivalent*, if for all t there exists a t' such that

$$[\![e]\!]^t_{\theta,\sigma} \sqsubseteq [\![e']\!]^{t'}_{\theta,\sigma}$$

holds, and for every t' there exists a t such that the converse relation holds.

The relation is obviously reflexive and symmetric, and transitivity can be shown easily. If e and e' are semantically equivalent and either of the sequences $[\![e]\!]^t_{\theta,\sigma}$ and $[\![e']\!]^{t'}_{\theta,\sigma}$ has a limit, then both have the same limit: $[\![e']\!]^\infty_{\theta,\sigma}$ is an upper bound for $[\![e]\!]^t_{\theta,\sigma}$ because for every t there is a t' with $[\![e]\!]^t_{\theta,\sigma} \sqsubseteq [\![e']\!]^{t'}_{\theta,\sigma} \sqsubseteq [\![e']\!]^\infty_{\theta,\sigma}$. There can be no lesser upper bound, because using the same argument in the other direction this would also be an upper bound for $[\![e']\!]^t_{\theta,\sigma}$.

As an example, consider beta equivalence. We want to show $(\lambda(x :: \tau) \to e_1)\ e_2$ and $e_1\ [e_2/x]$ to be semantically equivalent for any e_1 and e_2. We can compute the semantics of the former w.r.t. environments θ, σ:

$$\begin{aligned}
&[\![(\lambda(x :: \tau) \to e_1)\ e_2]\!]^{t+2}_{\theta,\sigma} \\
&= [\![(\lambda(x :: \tau) \to e_1)]\!]^{t+1}_{\theta,\sigma} \left([\![e_2]\!]^{t+1}_{\theta,\sigma}, t\right) \\
&= \lambda\, \mathbf{x}\ s.[\![e_1]\!]^{s+1}_{\theta,\sigma[x\mapsto\mathbf{x}]} \left([\![e_2]\!]^{t+1}_{\theta,\sigma}, t\right) \\
&= [\![e_1]\!]^{t+1}_{\theta,\sigma\left[x\mapsto[\![e_2]\!]^{t+1}_{\theta,\sigma}\right]}
\end{aligned}$$

How does this compare to $[\![e_1\ [e_2/x]]\!]^{t+1}_{\theta,\sigma}$? The only difference is, that the variable x in e_1 is bound to the denotation of e_2 via the environment, while in $e_1\ [e_2/x]$ the variable x is replaced by the expression e_2 syntactically. Semantic equivalence follows directly from the next lemma:

Lemma 4.6.2. Let $\Gamma \vdash e_2 :: \tau'$ be an expression having some type in a context Γ not containing the variable x. Let $\Gamma, x{::}\tau' \vdash e_1{::}\tau$ be another expression that can additionally contain x. Let θ, σ be environments corresponding to Γ and $t_0 > 0$ some fixed natural number. Then for every $t < t_0$ there exists a t' such that

$$[\![e_1]\!]^{t}_{\theta,\sigma[x \mapsto [\![e_2]\!]^{t_0}_{\theta,\sigma}]} \sqsubseteq [\![e_1\ [e_2/x]]\!]^{t'}_{\theta,\sigma}$$

and for every $t' < t_0$ there exists a t such that the converse relation holds.

Proof. The proof is by induction on e_1.
If e_1 is simply x, the left hand side is $[\![e_2]\!]^{t_0}_{\theta,\sigma}$ and the right hand side is $[\![e_2]\!]^{t'}_{\theta,\sigma}$. For any given t, set $t' = t_0$ to make the claim true. For any given t' set $t = t_0$ and the claim is true due to $t' < t_0$ and monotonicity of the semantics.
If e_1 is any other expression, find sufficiently big indexes to make the claim true for all subexpressions by using the induction hypothesis. The maximum of these indexes plus one will be sufficiently big to satisfy the claim because the semantics of an expression is monotone in the semantics of its subexpressions. □

Other semantic equivalences can be shown in similar manner. Since their proofs come down to using essentially the same ideas we have just seen, we only give the statements. In SaLT, eta equivalence holds, i.e., $\lambda(x :: \tau) \to e\ x$ is equivalent to e if x does not occur in e. Also, sets satisfy the monad laws:

$$\{x\} \ggg\!= v = v\ x \tag{4.5}$$

$$u \ggg\!= \lambda x \to \{x\} = u \tag{4.6}$$

$$u \ggg\!= (\lambda x \to v\ x \ggg\!= w) = (u \ggg\!= v) \ggg\!= w \tag{4.7}$$

They are a commutative monad:

$$u \ggg\!= \lambda x \to v \ggg\!= \lambda y \to w\ x\ y = v \ggg\!= \lambda y \to u \ggg\!= \lambda x \to w\ x\ y \tag{4.8}$$

Sets (with a fixed element type) form a monoid using

$$\begin{aligned}&`\mathit{union}` :: \forall a.\mathsf{Set}\ a \to \mathsf{Set}\ a \to \mathsf{Set}\ a\\ &u\ `\mathit{union}`\ v = \mathbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda b \to \mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to u; \mathsf{False} \to v\}\end{aligned}$$

as the monoid operation and $\mathbf{failed}_{\mathsf{Set}\ \tau}$ as the neutral element. Bind does distribute over this binary union

$$(u\ `\mathit{union}`\ v) \ggg\!= w = (u \ggg\!= w)\ `\mathit{union}`\ (v \ggg\!= w) \tag{4.9}$$

but **failed** $\ggg\!= w$ is not necessarily **failed**, so sets do not form an additive monad.
In mathematics, the neutral element of set union is the empty set, but there is no such thing in SaLT. Instead, the semantic equivalence

$$\mathbf{failed}_{\mathsf{Set}\ \tau} = \{\mathbf{failed}_{\tau}\} \tag{4.10}$$

$$
\begin{aligned}
\psi(\alpha) &= \alpha \\
\psi(\tau \rightarrow \tau') &= \psi(\tau) \rightarrow \mathsf{Set}\ (\psi(\tau')) \\
\psi(\mathsf{Bool}) &= \mathsf{Bool} \\
\psi(\mathsf{Nat}) &= \mathsf{Nat} \\
\psi([\tau]) &= [\psi(\tau)] \\
\psi(\tau, \tau') &= (\psi(\tau), \psi(\tau'))
\end{aligned}
$$

Figure 4.17: Translation from CuMin to SaLT for types

holds and every set 'contains' at least the element level failure. This explains why **failed** $\ggg\!= w$ is not always equivalent to **failed**:

$$
\begin{aligned}
& \quad \textbf{failed} \ggg\!= w \\
= & \quad \langle \text{ equation (4.10) } \rangle \\
& \quad \{\textbf{failed}\} \ggg\!= w \\
= & \quad \langle \text{ first monad law (4.5) } \rangle \\
& \quad w\ \textbf{failed}
\end{aligned}
$$

So, unless w is strict, **failed** $\ggg\!= w$ can have a non-bottom value.
Sets interacting with failure in this way corresponds to the way nondeterminism interacts with failure in Curry. In a certain sense, this is the only reason we did not choose a fragment of Haskell as the target of the translation procedure.

4.6.3 The Translation Procedure

Now that we know about SaLT, we can discuss how to translate CuMin programs into SaLT. The translation procedure is purely syntax-driven, i.e., we translate an expression e by translating its subexpressions and combining the results. The type of the resulting SaLT expression $\psi(e)$ only depends on the type of the CuMin expression we translated. In particular, the resulting type does not tell us whether nondeterminism actually occurs, but marks all positions where it could occur.
The rules for translating types are given in figure 4.17 and boil down to wrapping the target of every function into a Set. In CuMin, every function is potentially nondeterministic and we represent the different possible results as a set in SaLT. This corresponds to the denotation of CuMin functions being set-valued semantics functions. Note that the translations of data types are again data types.
Expressions are translated by the rules given in figure 4.18. Since every CuMin expression is potentially nondeterministic, the result always is a set. To be more precise, if a CuMin expression has type τ, then the translated expression $\psi(e)$ has type $\mathsf{Set}\ (\psi(\tau))$.
In the translation rule for **let** ... **in** bindings, τ_i is the result of translating the type of e_i. This is the only choice that makes the application of $(\ggg\!=)$ well-typed. Whenever

$$
\begin{aligned}
\psi(x) &= \{\, x \,\} & \psi(\mathsf{False}) &= \{\, \mathsf{False} \,\} \\
\psi(\mathbf{failed}_{\,\tau}) &= \{\, \mathbf{failed}_{\,\psi(\tau)} \,\} & \psi(\mathsf{Nil}_{\,\tau}) &= \{\, \mathsf{Nil}_{\,\psi(\tau)} \,\} \\
\psi(n) &= \{\, n \,\} & \psi(\mathsf{Cons}_{\,\tau}) &= \mathit{wrap}_2\ (\lambda h\ t \to \{\, \mathsf{Cons}_{\,\psi(\tau)}\ h\ t \,\}) \\
\psi(\mathsf{True}) &= \{\, \mathsf{True} \,\} & \psi(\mathsf{Pair}_{\,\tau,\tau'}) &= \mathit{wrap}_2\ (\lambda l\ r \to \{\, \mathsf{Pair}_{\,\psi(\tau),\psi(\tau')}\ h\ t \,\})
\end{aligned}
$$

$$
\begin{aligned}
\psi(\ \mathbf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e) &= \psi(e_1) \gg\!= \lambda(x_1 :: \tau_1) \to \ldots \psi(e_n) \gg\!= \lambda(x_n :: \tau_n) \to \psi(e) \\
\psi(\ \mathbf{let}\ x_1 :: \tau_1\ \mathbf{free\ in}\ e) &= \mathbf{unknown}_{\,\psi(\tau_1)} \gg\!= \lambda x_1 \to \psi(e) \\
\psi(\mathit{fun}_{\,\overline{\tau_m}}) &= \mathit{wrap}_n\ \mathit{fun}^T{}_{\overline{\psi(\tau_m)}} \\
&\quad \text{(where } n \text{ is the arity of } \mathit{fun}\text{)} \\
\psi(e_1\ e_2) &= \psi(e_1) \gg\!= \lambda f \to \psi(e_2) \gg\!= \lambda a \to f\ a \\
\psi(e_1 + e_2) &= \psi(e_1) \gg\!= \lambda n_1 \to \psi(e_2) \gg\!= \lambda n_2 \to \{\, n_1 + n_2 \,\} \\
\psi(e_1 == e_2) &= \psi(e_1) \gg\!= \lambda n_1 \to \psi(e_2) \gg\!= \lambda n_2 \to \{\, n_1 == n_2 \,\}
\end{aligned}
$$

$$
\begin{aligned}
&\psi(\ \mathbf{case}\ e\ \mathbf{of}\ \{\, \mathsf{True} \to e_1; \mathsf{False} \to e_2 \,\}) = \\
&\quad \psi(e) \gg\!= \lambda b \to \mathbf{case}\ b\ \mathbf{of}\ \{\, \mathsf{True} \to \psi(e_1); \mathsf{False} \to \psi(e_2) \,\} \\
&\psi(\ \mathbf{case}\ e\ \mathbf{of}\ \{\, \mathsf{Nil} \to e_1; \mathsf{Cons}\ h\ t \to e_2 \,\}) = \\
&\quad \psi(e) \gg\!= \lambda l \to \mathbf{case}\ l\ \mathbf{of}\ \{\, \mathsf{Nil} \to \psi(e_1); \mathsf{Cons}\ h\ t \to \psi(e_2) \,\} \\
&\psi(\ \mathbf{case}\ e\ \mathbf{of}\ \{\, \mathsf{Pair}\ l\ r \to e_1 \,\}) = \\
&\quad \psi(e) \gg\!= \lambda p \to \mathbf{case}\ p\ \mathbf{of}\ \{\, \mathsf{Pair}\ l\ r \to \psi(e_1) \,\}
\end{aligned}
$$

Figure 4.18: Translation from CuMin to SaLT for expressions

new variables are introduced by the translation, they have to have sufficiently fresh names. In the discussed examples, we will ensure this manually, while also trying to pick suggestive names. In the translation of a function symbol, n is the function's arity and the functions wrap_n are defined by

$$\begin{aligned} \mathit{wrap}_0\, f &= f \\ \mathit{wrap}_{n+1}\, f &= \{\lambda x \to \mathit{wrap}_n\ (f\ x)\} \end{aligned}$$

so their types are:

$$\begin{aligned} &\mathit{wrap}_n :: \forall a_1, \ldots, a_n, a. \\ &\qquad (a_1 \to \qquad a_2 \to \ldots \to \qquad a_n \to a) \to \\ &\quad \mathsf{Set}\ (a_1 \to \mathsf{Set}\ (a_2 \to \ldots \to \mathsf{Set}\ (a_n \to a) \ldots)) \end{aligned}$$

For any concrete $n \in \mathbb{N}$ the function wrap_n can be defined within SaLT. For example $\mathit{wrap}_1\, f = \{\lambda x \to \mathit{wrap}_0\ (f\ x)\} = \{\lambda x \to f\ x\} = \{f\}$.
Finally, fun^T is the translation of the CuMin function symbol *fun*. A function definition

$$\mathit{fun} :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau_1 \to \ldots \to \tau_n \to \tau; \mathit{fun}\ \overline{x_n} = e$$

is translated to:

$$\mathit{fun}^T :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \psi(\tau_1) \to \ldots \to \psi(\tau_n) \to \mathsf{Set}\ (\psi(\tau)); \mathit{fun}^T\ \overline{x_n} = \psi(e)$$

When translating, we change the names of all function symbols (*fun* becomes fun^T) to avoid name clashes. For example *double* is a reasonable function in both CuMin and SaLT, but the translation of CuMin's *double* function is not SaLT's *double* function. Instead *double* is renamed to double^T when translating, so we can distinguish it from the native *double*.
The translation of a program consists of the translations of all the function definitions and the definitions of all wrap_n functions that are used.

The following lemma formalizes the claim that the translation preserves the denotational semantics.

Lemma 4.6.3. Let $\Gamma = \overline{\alpha_m}, \overline{\mathsf{Data}\ \alpha_{i_j}}, \overline{x_n :: \tau_n}$ be some CuMin context. Define a context $\Gamma' = \overline{\alpha_m}, \overline{\mathsf{Data}\ \alpha_{i_j}}, \overline{x_n :: \psi(\tau_n)}$.

1. If τ is a CuMin type within Γ, then $\psi(\tau)$ is a SaLT type within Γ'.

2. If $\Gamma \vdash \tau \in \mathsf{Data}$ holds, then so does $\Gamma' \vdash \psi(\tau) \in \mathsf{Data}$.

3. If $\Gamma \vdash e :: \tau$ holds for some CuMin expression e and type τ, then so does $\Gamma' \vdash \psi(e) :: \mathsf{Set}\ (\psi(\tau))$.

4. If θ, σ are a type and term environment for Γ, then they are also a type and term environment for Γ'.

5. $[\![\tau]\!]_\theta = [\![\psi(\tau)]\!]_\theta$

6. $[\![e]\!]^\infty_{\theta,\sigma} = [\![\psi(e)]\!]^\infty_{\theta,\sigma}$

In the last two equations the left hand side uses the CuMin denotational semantics, while the right hand side uses the SaLT denotational semantics. The proof is done by straightforward induction and we do not carry it out in detail.

The translation of functions given here differs from the one given in [Mehner et al., 2014]. There, the translation of an n-ary function with type $\tau_1 \to \ldots \to \tau_n \to \tau$ is $\mathsf{Set}\ (\psi(\tau_1) \to \mathsf{Set}\ (\ldots \to \mathsf{Set}\ (\psi(\tau_n) \to \mathsf{Set}\ (\psi(\tau)))\ldots))$, which introduces many additional set wrappers. The term is then given as a singleton set containing a lambda abstraction, which again produces a singleton set containing a lambda abstraction and so on. The sets are singleton sets by construction, so we know the set types to be an unnecessary complication. No nondeterminism can occur before applying the function n times, so taking the function's arity into account, we can also give the translation a simpler type, which motivates the change. Because of the lacking set wrappers, the translation of a function now has a function type and can thus be given by a rule with variables on the left hand side. This is the reason SaLT now allows defining functions like this, while in [Mehner et al., 2014] there was no reason to have this syntactic possibility. The simplified translation for function definitions however comes with a price: The translation has to add the *wrap* function whenever a function is used. A handy equation for dealing with them is

$$\mathit{wrap}_{n+1}\, f \ggg\!= \lambda f' \to f'\ x = \mathit{wrap}_n\ (f\ x) \tag{4.11}$$

where f is any function-typed SaLT expression. The proof is some easy equational reasoning:

$$\begin{array}{ll}
& \mathit{wrap}_{n+1}\, f \ggg\!= \lambda f' \to f'\ x \\
= & \langle \text{ definition of } \mathit{wrap}. \rangle \\
& \{\lambda y \to \mathit{wrap}_n\ (f\ y)\} \ggg\!= \lambda f' \to f'\ x \\
= & \langle \text{ first monad law (4.5) } \rangle \\
& (\lambda y \to \mathit{wrap}_n\ (f\ y))\ x \\
= & \langle \text{ beta equivalence } \rangle \\
& \mathit{wrap}_n\ (f\ x)
\end{array}$$

The formula is used to prove the following lemma, which gives an alternative translation for partially and fully applied function symbols.

Lemma 4.6.4. Let *fun* be an n-ary CuMin function and $k \leq n$. Then the following holds for CuMin expressions $e_1 \ldots e_k$:

$$\psi(\mathit{fun}_{\overline{\tau_m}}\ e_1 \ldots e_k) = \psi(e_1) \ggg\!= \lambda x_1 \to \ldots \psi(e_k) \ggg\!= \lambda x_k \to \mathit{wrap}_{n-k}\ ((\mathit{fun}^T{}_{\overline{\psi(\tau_m)}})\ x_1 \ldots x_k)$$

In particular, if $k = n$ the wrap_0 can be dropped.

Proof. The proof is by induction on k, where in the base case ($k = 0$) the claim

$$\psi(\mathit{fun}) = \mathit{wrap}_n\ \mathit{fun}^T$$

is simply the definition of the translation of function symbols. For the induction step let $k < n$ be given.

$$\begin{array}{ll} & \psi(\mathit{fun}\ e_1 \dots e_{k+1}) \\ = & \langle \text{ translation of function application and monad commutativity (4.8) } \rangle \\ & \psi(e_{k+1}) \ggg\!= \lambda x_{k+1} \to \psi(\mathit{fun}\ e_1 \dots e_k) \ggg\!= \lambda f \to f\ x_{k+1} \\ = & \langle \text{ induction hypothesis } \rangle \\ & \psi(e_{k+1}) \ggg\!= \lambda x_{k+1} \to \psi(e_1) \ggg\!= \lambda x_1 \to \dots \psi(e_k) \ggg\!= \lambda x_k \to \\ & \quad \mathit{wrap}_{n-k}\ (\mathit{fun}^T\ x_1 \dots x_k) \ggg\!= \lambda f \to f\ x_{k+1} \\ = & \langle \text{ set monad is commutative (4.8) } \rangle \\ & \psi(e_1) \ggg\!= \lambda x_1 \to \dots \psi(e_{k+1}) \ggg\!= \lambda x_{k+1} \to \\ & \quad \mathit{wrap}_{n-k}\ (\mathit{fun}^T\ x_1 \dots x_k) \ggg\!= \lambda f \to f\ x_{k+1} \\ = & \langle \text{ equation (4.11) } \rangle \\ & \psi(e_1) \ggg\!= \lambda x_1 \to \dots \psi(e_{k+1}) \ggg\!= \lambda x_{k+1} \to \mathit{wrap}_{n-(k+1)}\ (\mathit{fun}^T\ x_1 \dots x_k\ x_{k+1}) \end{array}$$

□

When applying the lemma, we will often have the special case of one of the e_i being a variable y_i already. Then, a further simplification is possible: Since $\psi(y_i) = \{y_i\}$, the $\psi(y_i) \ggg\!= \lambda x_i \to$ can be omitted using the first monad law (4.5).

4.6.4 Example

As an example, consider the following CuMin function definition:

$$\begin{array}{l} \mathit{map} :: \forall a\ b.(a \to b) \to [a] \to [b] \\ \mathit{map}\ f\ l = \textbf{case}\ l\ \textbf{of}\ \{\mathsf{Nil} \to \mathsf{Nil}_{\,b}\,; \mathsf{Cons}\ x\ \mathit{xs} \to \mathsf{Cons}_{\,b}\ (f\ x)\ (\mathit{map}_{\ a,b}\, f\ \mathit{xs})\} \end{array}$$

The type signature of the translated function will be

$$\mathit{map}^T :: \forall a\ b.(a \to \mathsf{Set}\ b) \to [a] \to \mathsf{Set}\ [b]$$

so the first argument of map^T can be a nondeterministic function and the result can also be nondeterministic.
The expression $\mathit{map}_{\ a,b}\, f\ \mathit{xs}$ can be translated using lemma 4.6.4, which yields:

$$\psi(f) \ggg\!= \lambda f' \to \psi(\mathit{xs}) \ggg\!= \lambda \mathit{xs}' \to \mathit{map}^T{}_{a,b}\, f'\ \mathit{xs}'$$

We can simplify further:

$$\begin{array}{ll} & \psi(f) \ggg\!= \lambda f' \to \psi(\mathit{xs}) \ggg\!= \lambda \mathit{xs}' \to \mathit{map}^T{}_{a,b}\, f'\ \mathit{xs}' \\ = & \langle \text{ translation of variables } \rangle \\ & \{f\} \ggg\!= \lambda f' \to \{\mathit{xs}\} \ggg\!= \lambda \mathit{xs}' \to \mathit{map}^T{}_{a,b}\, f'\ \mathit{xs}' \end{array}$$

$$\begin{array}{ll}
= & \langle \text{ first monad law (4.5) } \rangle \\
 & map^T{}_{a,b}\, f\ xs
\end{array}$$

The lemma does not cover the application $f\ x$, because f is a function-typed variable. However, we can still use the first monad law:

$$\begin{array}{ll}
 & \psi(f\ x) \\
= & \langle \text{ translation of application } \rangle \\
 & \psi(f) \gg\!= \lambda f' \rightarrow \psi(x) \gg\!= \lambda x' \rightarrow f'\ x' \\
= & \langle \text{ translation of variables } \rangle \\
 & \{f\} \gg\!= \lambda f' \rightarrow \{x\} \gg\!= \lambda x' \rightarrow f'\ x' \\
= & \langle \text{ first monad law (4.5) } \rangle \\
 & f\ x
\end{array}$$

The translation of $\mathsf{Cons}_b\,(f\ x)\ (map\ {}_{a,b} f\ xs)$ can be simplified analogously to the lemma:

$$\begin{array}{ll}
 & \psi(\mathsf{Cons}_b\,(f\ x)\ (map\ {}_{a,b} f\ xs)) \\
= & \langle \text{ analogous to lemma 4.6.4 } \rangle \\
 & \psi(f\ x) \gg\!= \lambda h \rightarrow \psi(map\ {}_{a,b} f\ xs) \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_b\ h\ t\} \\
= & \langle \text{ see above } \rangle \\
 & f\ x \gg\!= \lambda h \rightarrow map^T{}_{a,b} f\ xs \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_b\ h\ t\}
\end{array}$$

In this case, we cannot use the first monad law anymore, because we do not find further singleton sets.
Now we can give the overall translation of the rule's body:

$$\begin{array}{ll}
 & \psi(\ \mathbf{case}\ l\ \mathbf{of}\ \{\mathsf{Nil} \rightarrow \mathsf{Nil}_b\,; \mathsf{Cons}\ x\ xs \rightarrow \mathsf{Cons}_b\,(f\ x)\ (map\ {}_{a,b} f\ xs)\}) \\
= & \langle \text{ translation of } \mathbf{case} \rangle \\
 & \{l\} \gg\!= \lambda l' \rightarrow \mathbf{case}\ l'\ \mathbf{of} \\
 & \quad \mathsf{Nil} \rightarrow \psi(\mathsf{Nil}_b) \\
 & \quad \mathsf{Cons}\ x\ xs \rightarrow \psi(\mathsf{Cons}_b\,(f\ x)\ (map\ {}_{a,b} f\ xs)) \\
= & \langle \text{ first monad law (4.5) } \rangle \\
 & \mathbf{case}\ l\ \mathbf{of} \\
 & \quad \mathsf{Nil} \rightarrow \psi(\mathsf{Nil}_b) \\
 & \quad \mathsf{Cons}\ x\ xs \rightarrow \psi(\mathsf{Cons}_b\,(f\ x)\ (map\ {}_{a,b} f\ xs)) \\
= & \langle \text{ see above } \rangle \\
 & \mathbf{case}\ l\ \mathbf{of} \\
 & \quad \mathsf{Nil} \rightarrow \{\mathsf{Nil}_b\} \\
 & \quad \mathsf{Cons}\ x\ xs \rightarrow f\ x \gg\!= \lambda h \rightarrow map^T{}_{a,b,} f\ xs \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_b\ h\ t\}
\end{array}$$

Thus the full definition of the translated function is:

$$\begin{array}{l}
map^T :: \forall a\ b.(a \rightarrow \mathsf{Set}\ b) \rightarrow [a] \rightarrow \mathsf{Set}\ [b] \\
map^T\ f\ l = \mathbf{case}\ l\ \mathbf{of} \\
\quad \mathsf{Nil} \qquad\quad\ \rightarrow \{\mathsf{Nil}_b\} \\
\quad \mathsf{Cons}\ x\ xs \rightarrow f\ x \gg\!= \lambda h \rightarrow map^T{}_{a,b} f\ xs \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_b\ h\ t\}
\end{array}$$

To take the example a bit further, note that the definition of the CuMin function *map* given above is also valid SaLT code and defines a SaLT function *map*. The two SaLT functions *map* and map^T satisfy the following equation:

$$map^T{}_{\mathsf{A,B}}\,(\lambda x \rightarrow \{f\ x\})\ xs = \{\,map_{\mathsf{A,B}}\,f\ xs\,\} \tag{4.12}$$

This can be checked by equational reasoning in SaLT, where we split cases on l. We start with the base case:

$$\begin{array}{ll}
& map^T{}_{\mathsf{A,B}}\,(\lambda x \rightarrow \{f\ x\})\ \mathsf{Nil}_{\mathsf{A}} \\
= & \langle \text{ definition of } map^T \ \rangle \\
& \{\mathsf{Nil}_{\mathsf{B}}\} \\
= & \langle \text{ definition of } map \ \rangle \\
& \{\,map_{\mathsf{A,B}}\,f\ \mathsf{Nil}_{\mathsf{A}}\,\}
\end{array}$$

In the induction step we will assume the claim to be true for xs already:

$$\begin{array}{ll}
& map^T{}_{\mathsf{A,B}}\,(\lambda y \rightarrow \{f\ y\})\ (\mathsf{Cons}_{\mathsf{A}}\ x\ xs) \\
= & \langle \text{ definition of } map^T \ \rangle \\
& (\lambda y \rightarrow \{f\ y\})\ x \gg\!= \lambda h \rightarrow map^T{}_{\mathsf{A,B}}\,(\lambda y \rightarrow \{f\ y\})\ xs \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_{\mathsf{B}}\ h\ t\} \\
= & \langle \text{ beta equivalence } \rangle \\
& \{f\ x\} \gg\!= \lambda h \rightarrow map^T{}_{\mathsf{A,B}}\,(\lambda y \rightarrow \{f\ y\})\ xs \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_{\mathsf{B}}\ h\ t\} \\
= & \langle \text{ first monad law (4.5) } \rangle \\
& map^T{}_{\mathsf{A,B}}\,(\lambda y \rightarrow \{f\ y\})\ xs \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_{\mathsf{B}}\,(f\ x)\ t\} \\
= & \langle \text{ induction hypothesis } \rangle \\
& \{\,map_{\mathsf{A,B}}\,f\ xs\,\} \gg\!= \lambda t \rightarrow \{\mathsf{Cons}_{\mathsf{B}}\,(f\ x)\ t\} \\
= & \langle \text{ first monad law (4.5) } \rangle \\
& \{\mathsf{Cons}_{\mathsf{B}}\,(f\ x)\ (map_{\mathsf{A,B}}\,f\ xs)\} \\
= & \langle \text{ definition of } map \ \rangle \\
& \{\,map_{\mathsf{A,B}}\,f\ (\mathsf{Cons}_{\mathsf{A}}\ x\ xs)\}
\end{array}$$

4.7 Parametricity for SaLT

The idea of parametricity goes back to Reynolds [1983] and later became the technical machinery behind free theorems [Wadler, 1989]. In the original setup a language with two different denotational semantics is considered, which are to be compared somehow. Given some expression, it has an interpretation in either semantics. Now we would like to know whether both interpretations coincide, but this question is generally meaningless: The semantic domains of the two semantics do not have to coincide, so we would be comparing apples and oranges.

So instead of asking whether both interpretations are equal, we ask whether they are *related* via some suitable relation called *logical relation.* To do so, for every type of the language such a relation between its two interpretations has to be defined. The claim

$$
\begin{aligned}
[\![\alpha]\!]_\rho &= \rho(\alpha)\\
[\![\tau \to \tau']\!]_\rho &= \Big\{(\mathbf{f}^-, \mathbf{f}^+) \mid \forall(\mathbf{x}^-, \mathbf{x}^+) \in [\![\tau]\!]_\rho . \forall t \in \mathbb{N}.(\mathbf{f}^-(\mathbf{x}^-, t), \mathbf{f}^+(\mathbf{x}^+, t)) \in [\![\tau']\!]_\rho\Big\}\\
[\![\mathsf{Bool}]\!]_\rho &= \{(\mathbf{b}^-, \mathbf{b}^+) \mid \mathbf{b}^-, \mathbf{b}^+ \in [\![\mathsf{Bool}]\!], \mathbf{b}^- \sqsubseteq \mathbf{b}^+\}\\
[\![\mathsf{Nat}]\!]_\rho &= \{(\mathbf{n}^-, \mathbf{n}^+) \mid \mathbf{n}^-, \mathbf{n}^+ \in [\![\mathsf{Nat}]\!], \mathbf{n}^- \sqsubseteq \mathbf{n}^+\}\\
[\![[\tau]]\!]_\rho &= \Big\{(\mathbf{x}_1^- : \ldots : \mathbf{x}_n^- : [], \mathbf{x}_1^+ : \ldots : \mathbf{x}_n^+ : []) \mid 0 \le n, (\mathbf{x}_i^-, \mathbf{x}_i^+) \in [\![\tau]\!]_\rho\Big\}\\
&\quad \cup \Big\{(\mathbf{x}_1^- : \ldots : \mathbf{x}_{n^-}^- : \bot, \mathbf{x}_1^+ : \ldots : \mathbf{x}_{n^+}^+ : \mathbf{e})\\
&\qquad \mid 0 \le n^- \le n^+, \forall i \le n^-.(\mathbf{x}_i^-, \mathbf{x}_i^+) \in [\![\tau]\!]_\rho, \mathbf{e} \in \{\bot, []\}\Big\}\\
[\![(\tau, \tau')]\!]_\rho &= \Big\{(\bot, \mathbf{p}^+) \mid \mathbf{p}^+ \in [\![(\tau, \tau')]\!]_{\theta^+, \sigma^+}\Big\}\\
&\quad \cup \Big\{((\mathbf{l}^-, \mathbf{r}^-), (\mathbf{l}^+, \mathbf{r}^+)) \mid (\mathbf{l}^-, \mathbf{l}^+) \in [\![\tau]\!]_\rho, (\mathbf{r}^-, \mathbf{r}^+) \in [\![\tau']\!]_\rho\Big\}\\
[\![\mathsf{Set}\ \tau]\!]_\rho &= \Big\{(A^-, A^+) \mid \forall \mathbf{a}^- \in A^-.\exists \mathbf{x}^- \in A^-, \mathbf{x}^+ \in A^+.\mathbf{a}^- \sqsubseteq \mathbf{x}^- \wedge (\mathbf{x}^-, \mathbf{x}^+) \in [\![\tau]\!]_\rho\Big\}
\end{aligned}
$$

Figure 4.19: Definition of the logical relation

then is, that for every expression of any type, the two interpretations of the expression are related via the relation associated to the type.
Wadler [1989] used an important special case: Instead of having two different denotational semantics, the same denotational semantics can be used twice. The case is not trivial, because picking the relations remains a degree of freedom. In particular, functions of the language itself (or rather their semantics) can be used as relations. This way 'being related' becomes a concept expressible in the language's own syntax.

In this section, we prove a parametricity theorem for SaLT (theorem 4.7.6). We will use the denotational semantics given in section 4.6 twice. For all closed types, both interpretations will be the same, so we could use equality as the relation associated to closed types. Yet following Johann and Voigtländer [2006] and others, we will use the definedness relation $\sqsubseteq$ instead. This will allow us to also deduce inequational free theorems. The definition of the (family of) relations is given in figure 4.19.
An equational version of the parametricity theorem appeared in [Mehner et al., 2014] already, which we shall repeat here. While in the paper a direct proof has been given, here the equational version is a consequence of the inequational version.

Let us take a closer look at the definition of the relations, in particular for set types. In essence, the definition tells us that two sets A^- and A^+ are related, if for every element $\mathbf{a}^- \in A^-$ there is a related element in A^+. However, the sets we are dealing with are closed, i.e., for every element of the set all less defined values also have to be elements. To accommodate for elements that have only been added because of this closure property, we only require the existence of a possibly more defined value $\mathbf{x}^-$ that is related to an element of the other set. If $\mathbf{a}^-$ is a maximal element of A^-, it has to be related to some element of A^+ itself, for there is no other choice but $\mathbf{a}^- = \mathbf{x}^-$.

If the relation $[\![\tau]\!]_\rho$ is a function $\mathbf{r} : [\![\tau]\!]_{\theta-} \to [\![\tau]\!]_{\theta+}$, the situation becomes easier. In this case, $\mathbf{x}^+$ is determined by $\mathbf{x}^-$ already, so the condition of the two being in relation may be stated as $\mathbf{r}(\mathbf{x}^-) \in A^+$ instead. If $\mathbf{r}$ is monotonous, which is the case for the semantics of syntactic functions, we can also omit $\mathbf{x}^-$. If $\mathbf{r}(\mathbf{a}^-) \in A^+$ holds, $\mathbf{x}^- = \mathbf{a}^-$ is a suitable choice. Conversely if $\mathbf{r}(\mathbf{a}^-)$ is not an element of A^+, then neither is any $\mathbf{x}^- \sqsupseteq \mathbf{a}^-$. Thus two sets A^- and A^+ are related if for all $\mathbf{a}^- \in A^-$ it holds that $\mathbf{r}(\mathbf{a}^-) \in A^+$ or equivalently

$$\bigcup_{\mathbf{a}^- \in A^-} {\downarrow}\mathbf{r}(\mathbf{a}^-) \subseteq A^+. \tag{4.13}$$

If the relation $[\![\tau]\!]_\rho$ is given by a function $\mathbf{r} : [\![\tau]\!]_{\theta+} \to [\![\tau]\!]_{\theta-}$ in the opposite direction, a similar situation arises. Now $\mathbf{x}^-$ is determined by $\mathbf{x}^+$, so the existence of $\mathbf{x}^+ \in A^+$ with $\mathbf{a}^- \sqsubseteq \mathbf{r}(\mathbf{x}^+)$ is required. Two sets A^- and A^+ are thus related if

$$A^- \subseteq \bigcup_{\mathbf{a}^+ \in A^+} {\downarrow}\mathbf{r}(\mathbf{a}^+). \tag{4.14}$$

Definition 4.7.1. A relation R between two posets is called *strict* if $(\bot, \bot) \in R$.

Lemma 4.7.2. If all relations $\rho(\alpha)$ are strict, then all relations $[\![\tau]\!]_\rho$ are strict.

Proof. The proof is by induction on types and splitting cases. For type variables, strictness of the relation is just the condition of the claim. For function types, $\tau \to \tau'$ strictness of the relation $[\![\tau \to \tau']\!]_\rho$ follows from strictness of the relation $[\![\tau']\!]_\rho$. For base types, lists and pairs strictness is immediate from the definition of the relation. Finally, for set types, we have to show $\{\bot\}$ is related to $\{\bot\}$, which also follows immediately from the definition of the relations. □

Lemma 4.7.3. For all closed types τ the relation $[\![\tau]\!]$ is the definedness relation.

Proof. The proof is by induction on types and splitting cases. Type variables do not have to be taken into account. For functions and data types, the claim directly follows from the definition in figure 4.19.

For set types we argue as follows. Let A^- and A^+ be related. Thus for every $\mathbf{a} \in A^-$ there are $\mathbf{x}^- \in A^-$ and $\mathbf{x}^+ \in A^+$ with $\mathbf{a} \sqsubseteq \mathbf{x}^- \sqsubseteq \mathbf{x}^+$, since by assumption, the relation on elements is the definedness relation. Since A^+ is down-closed, $\mathbf{a} \in A^+$ holds. Since $\mathbf{a}$ has been arbitrary, $A^- \subseteq A^+$.

Now conversely, let $A^- \subseteq A^+$ be given. The two sets are related, since for every $\mathbf{a} \in A^-$, we can set $\mathbf{x}^- = \mathbf{x}^+ = \mathbf{a}$ and both $\mathbf{a} \sqsubseteq \mathbf{x}^-$ and $\mathbf{x}^- \sqsubseteq \mathbf{x}^+$ hold. □

Definition 4.7.4. A relation R between two posets P^- and P^+ is called *left-whole*, if for every $\mathbf{a}^- \in P^-$ there is some $\mathbf{a}^- \sqsubseteq \mathbf{x}^- \in P^-$ and some $\mathbf{x}^+ \in P^+$, such that $\mathbf{x}^-$ and $\mathbf{x}^+$ are related via R. A relation R is called *right-whole* if its inverse is left-whole and called *whole* if it is both left- and right-whole.

Lemma 4.7.5. If all relations $\rho(\alpha)$ are left-whole, then for all data types τ the relation $\llbracket\tau\rrbracket_\rho$ is left-whole.

Proof. The proof is by induction on the rules for deriving $\Gamma \vdash \tau \in \mathsf{Data}$. For type variables $\llbracket\alpha\rrbracket_\rho = \rho(\alpha)$, so the claim is true. Bool and Nat are closed types, so because of lemma 4.7.3 the relations $\llbracket\mathsf{Bool}\rrbracket_\rho$ and $\llbracket\mathsf{Nat}\rrbracket_\rho$ are the respective definedness relations. For those relations we can prove being left-whole by setting $\mathbf{a}^- = \mathbf{x}^- = \mathbf{x}^+$.
Let some $\mathbf{a}_1^- : \ldots : \mathbf{a}_n^- : \mathbf{e} \in \llbracket[\tau]\rrbracket_{\theta^-}$ with $\mathbf{e} \in \{\bot, []\}$ be given. Since we assume $\llbracket\tau\rrbracket_\rho$ to be left-whole, there is a series of $\mathbf{a}_i^- \sqsubseteq \mathbf{x}_i^- \in \llbracket\tau\rrbracket_{\theta^-}$ and a series of $\mathbf{x}_i^+ \in \llbracket\tau\rrbracket_{\theta^+}$ such that $\mathbf{x}_i^-$ and $\mathbf{x}_i^+$ are related. Then $\mathbf{a}_1^- : \ldots : \mathbf{a}_n^- : \mathbf{e} \sqsubseteq \mathbf{x}_1^- : \ldots : \mathbf{x}_n^- : \mathbf{e}$ and $\mathbf{x}_1^- : \ldots : \mathbf{x}_n^- : \mathbf{e}$ is related to $\mathbf{x}_1^+ : \ldots : \mathbf{x}_n^+ : \mathbf{e}$.
The case for tuple types is shown analogously to lists. □

Theorem 4.7.6. Let Γ be a SaLT context. Let θ^-, σ^- and θ^+, σ^+ be two type and term environments for Γ. Let $\rho(\alpha)$ be a strict relation between $\theta^-(\alpha)$ and $\theta^+(\alpha)$ for all type variables in Γ that is also left-whole for all type variables α with $\alpha \in \mathsf{Data}$ in Γ. If $(\sigma^-(x), \sigma^+(x)) \in \llbracket\tau\rrbracket_\rho$ for all $x :: \tau$ in Γ, then

$$\left(\llbracket e\rrbracket^t_{\theta^-,\sigma^-}, \llbracket e\rrbracket^t_{\theta^+,\sigma^+}\right) \in \llbracket\tau\rrbracket_\rho$$

for all valid judgments $\Gamma \vdash e :: \tau$ and $t \in \mathbb{N}$.

Proof. The proof is by induction on t and over the expression e. Since the proof is well-known for all kinds of typed lambda calculi, we only do some of the relevant cases. Because of $\llbracket\mathbf{failed}_\tau\rrbracket^t_{\theta^\pm,\sigma^\pm} = \bot$ and lemma 4.7.2, the interpretations of $\mathbf{failed}_\tau$ are always related.

For singleton sets, we have

$$\llbracket\{\, e \,\}\rrbracket^{t+1}_{\theta^\pm,\sigma^\pm} = \downarrow\llbracket e\rrbracket^t_{\theta^\pm,\sigma^\pm}.$$

For every $\mathbf{v} \in \downarrow\llbracket e\rrbracket^t_{\theta^-,\sigma^-}$, by definition, $\mathbf{v} \sqsubseteq \llbracket e\rrbracket^t_{\theta^-,\sigma^-}$ holds and $\llbracket e\rrbracket^t_{\theta^-,\sigma^-}$ is related to $\llbracket e\rrbracket^t_{\theta^+,\sigma^+} \in \downarrow\llbracket e\rrbracket^t_{\theta^+,\sigma^+}$ by the induction hypothesis. Thus $\llbracket\{\, e \,\}\rrbracket^{t+1}_{\theta^-,\sigma^-}$ is related to $\llbracket\{\, e \,\}\rrbracket^{t+1}_{\theta^+,\sigma^+}$.

For $e_1 \ggg\!= e_2$ by the defintion of the semantics:

$$\llbracket e_1 \ggg\!= e_2\rrbracket^{t+1}_{\theta^\pm,\sigma^\pm} = \bigcup_{\mathbf{x}^\pm \in \llbracket e_1\rrbracket^t_{\theta^\pm,\sigma^\pm}} \llbracket e_2\rrbracket^t_{\theta^\pm,\sigma^\pm}(\mathbf{x}^\pm, t)$$

So let any $\mathbf{v} \in \llbracket e_1 \ggg\!= e_2\rrbracket^{t+1}_{\theta^\pm,\sigma^\pm}$ be given. Pick a suitable $\mathbf{x} \in \llbracket e_1\rrbracket^t_{\theta^-,\sigma^-}$ such that $\mathbf{v} \in \llbracket e_2\rrbracket^t_{\theta^-,\sigma^-}(\mathbf{x}, t)$. By the first induction hypothesis, $\llbracket e_1\rrbracket^t_{\theta^-,\sigma^-}$ is related to $\llbracket e_1\rrbracket^t_{\theta^+,\sigma^+}$, so there is some $\mathbf{x} \sqsubseteq \mathbf{y}^- \in \llbracket e_1\rrbracket^t_{\theta^-,\sigma^-}$ and some $\mathbf{y}^+ \in \llbracket e_1\rrbracket^t_{\theta^+,\sigma^+}$ such that $\mathbf{y}^-$ and $\mathbf{y}^+$ are related. Because of monotonicity $\llbracket e_2\rrbracket^t_{\theta^-,\sigma^-}(\mathbf{x}, t) \subseteq \llbracket e_2\rrbracket^t_{\theta^-,\sigma^-}(\mathbf{y}^-, t)$, so $\mathbf{v} \in \llbracket e_2\rrbracket^t_{\theta^-,\sigma^-}(\mathbf{y}^-, t)$.
Because of the second induction hypothesis, $\llbracket e_2\rrbracket^t_{\theta^-,\sigma^-}$ is related to $\llbracket e_2\rrbracket^t_{\theta^+,\sigma^+}$ and by definition this means sending related arguments to related results. Thus also $\llbracket e_2\rrbracket^t_{\theta^-,\sigma^-}(\mathbf{y}^-, t)$

and $[\![e_2]\!]^t_{\theta^+,\sigma^+}(\mathbf{y}^+, t)$ are related. So for $\mathbf{v} \in [\![e_2]\!]^t_{\theta^-,\sigma^-}(\mathbf{y}^-, t)$ there has to be some $\mathbf{v} \sqsubseteq \mathbf{w}^- \in [\![e_2]\!]^t_{\theta^-,\sigma^-}(\mathbf{y}^-, t)$ which is related to some $\mathbf{w}^+ \in [\![e_2]\!]^t_{\theta^+,\sigma^+}(\mathbf{y}^+, t)$. These are in fact the elements we are looking for, since

$$\mathbf{w}^\pm \in [\![e_2]\!]^t_{\theta^\pm,\sigma^\pm}(\mathbf{y}^\pm, t) \subseteq \bigcup_{\mathbf{x}^\pm \in [\![e_1]\!]^t_{\theta^\pm,\sigma^\pm}} [\![e_2]\!]^t_{\theta^\pm,\sigma^\pm}(\mathbf{x}^\pm, t) = [\![e_1 \ggg e_2]\!]^{t+1}_{\theta^\pm,\sigma^\pm}.$$

□

Using the parametricity theorem 4.7.6 we can derive free theorems for SaLT. This is fully analogous to free theorems in Haskell, as both languages are deterministic. The logical relation can be instantiated with the denotation of a syntactic function. This function has to be strict, since SaLT has a **failed** primitive. More precisely, a function-typed term g can be used as the logical relation if g **failed** $=$ **failed** holds.

A function is automatically left-whole when viewed as a relation. Therefore, the logical relation in the theorem can always be instantiated to the denotation of a syntactic function, even if a Data constraint occurs. This corresponds to the fact that g **unknown** is always a subset of **unknown**.

If we want to instantiate the logical relation to the inverse of a function, we have to prove right-wholeness instead. A function $\mathbf{g}$ from a poset P^- to a poset P^+ is right-whole, if for every $\mathbf{a}^+ \in P^+$ there is some $\mathbf{a}^+ \sqsubseteq \mathbf{x}^+ \in P^+$ and some $\mathbf{x}^- \in P^-$, such that $\mathbf{g}(\mathbf{x}^-) = \mathbf{x}^+$. Since $\mathbf{x}^+$ is determined by $\mathbf{g}$ and $\mathbf{x}^-$, this is equivalent to the following:

Definition 4.7.7. A function $\mathbf{g}$ from a poset P^- to a poset P^+ is called *whole*, if for every $\mathbf{x} \in P^+$ there is some $\mathbf{x}^- \in P^-$ with $\mathbf{g}(\mathbf{x}^-) \sqsupseteq \mathbf{x}^+$.

Since all functions are automatically left-whole as relations, being right-whole and being whole is the same for functions. This terminology agrees with [Mehner et al., 2014], where we did not take inequations into account.

When working with sets, the following insight is very useful. Suppose the relation between two types A and B is given by a function $g :: \mathsf{A} \to \mathsf{B}$, i.e., x and y are related if $g\ x = y$. Then the lifted relation between $\mathsf{Set\ A}$ and $\mathsf{Set\ B}$ is also given by a function, which is $\mathit{smap}\ g :: \mathsf{Set\ A} \to \mathsf{Set\ B}$. Where smap

$$\mathit{smap} :: \forall a, b :: (a \to b) \to \mathsf{Set}\ a \to \mathsf{Set}\ b$$
$$\mathit{smap}\ g\ s = s \ggg \lambda x \to \{g\ x\}$$

applies a function to every element of a set. This follows directly from equations (4.13) and (4.14).

4.8 Proving Free Theorems for CuMin

4.8.1 Side Conditions

Before we can prove any free theorems for CuMin, we need ways to deal with the side conditions. These conditions will always restrict the function that takes the place of the logic relation.

In a language with failure, this function has to be strict, i.e., send failure to failure [Wadler, 1989]. We use the same definition for CuMin:

Definition 4.8.1. A CuMin expression $g :: \mathsf{A} \to \mathsf{B}$ is called *strict*, if

$$g\ \mathbf{failed}_{\mathsf{A}} = \mathbf{failed}_{\mathsf{B}}$$

holds.

An alternative to requiring strictness is to use inequational-style free theorems [Johann and Voigtländer, 2006]. Obviously, $g\ \mathbf{failed}_{\mathsf{A}} \sqsupseteq \mathbf{failed}_{\mathsf{B}}$ holds for any g, where we write $e_1 \sqsupseteq e_2$ if $[\![e_1]\!]_{\theta,\sigma} \sqsupseteq [\![e_2]\!]_{\theta,\sigma}$ for all environments θ, σ.

We also have to restrict nondeterminism, which will be done using the translation to SaLT. Take some function-typed CuMin term $g :: \mathsf{A} \to \mathsf{B}$. The translation $\psi(g)$ has type $\mathsf{Set}\ (\psi(\mathsf{A}) \to \mathsf{Set}\ (\psi(\mathsf{B})))$. We want to compare g to a function-typed term $\hat{g} :: \psi(\mathsf{A}) \to \psi(\mathsf{B})$, which is deterministic in the sense that its type does not permit nondeterminism. The SaLT terms $\psi(g)$ and $\hat{g}$ cannot be semantically equivalent, because they do not even have the same type. However, we can add singleton set wrappers to 'convert' $\hat{g}$ to the type of $\psi(g)$:

$$\{\lambda x :: \psi(\mathsf{A}) \to \{\hat{g}\ x\}\} :: \mathsf{Set}\ (\psi(\mathsf{A}) \to \mathsf{Set}\ (\psi(\mathsf{B})))$$

The type of the expression contains the set type constructor twice, so type-wise it has to be considered effectful. Yet, since we know that all sets are singleton sets, the above expression is deterministic in the sense that no real choice occurs. If a $\hat{g}$ exists, such that $\psi(g) = \{\lambda x :: \psi(\mathsf{A}) \to \{\hat{g}\ x\}\}$, we call g *deterministic* and $\hat{g}$ a *witness*.
Christiansen et al. [2010] observed, that in many situations it is enough to only restrict the inner layer. The following definition captures this restriction in a style similar to the above considerations.

Definition 4.8.2. A CuMin expression $g :: \mathsf{A} \to \mathsf{B}$ is called *multi-deterministic*, if there is a SaLT term $\hat{g} :: \mathsf{Set}\ (\psi(\mathsf{A}) \to \psi(\mathsf{B}))$, such that

$$\psi(g) = \hat{g} \ggg\!= \lambda \hat{g}' \to \{\lambda x \to \{\hat{g}'\ x\}\}$$

holds semantically.

In this situation, we also call $\hat{g}$ a witness (of g being multi-deterministic).
Let us take a look at an example, relying on the following CuMin function definitions:

$id :: \forall a. a \to a$
$id\ x = x$

$inc :: \mathsf{Nat} \to \mathsf{Nat}$
$inc\ n = n + 1$

$mayInc0 :: \mathsf{Nat} \to \mathsf{Nat}$
$mayInc0 = id\ ?\ inc$

$$\begin{array}{l} \mathit{mayInc1} :: \mathsf{Nat} \rightarrow \mathsf{Nat} \\ \mathit{mayInc1}\ x = x \mathbin{?} x + 1 \end{array}$$

The expression *inc* is deterministic, which is witnessed by the SaLT expression $\lambda n \rightarrow n + 1$. Indeed:

$$\begin{array}{ll}
 & \psi(\mathit{inc}) \\
= & \langle \text{ definition of the translation } \rangle \\
 & \mathit{wrap}_1\ \mathit{inc}^T \\
= & \langle \text{ definition of } \mathit{wrap} \ \rangle \\
 & \{\lambda x \rightarrow \mathit{wrap}_0\ \mathit{inc}^T\ x\} \\
= & \langle \text{ definition of } \mathit{wrap} \ \rangle \\
 & \{\lambda x \rightarrow \mathit{inc}^T\ x\} \\
= & \langle \text{ definition of } \mathit{inc}^T \ \rangle \\
 & \{\lambda x \rightarrow \{x + 1\}\}
\end{array}$$

Both *mayInc0* and *mayInc1* use nondeterminism and in fact neither of them is deterministic in the above sense. Yet, *mayInc0* is at least multi-deterministic:

$$\begin{array}{ll}
 & \psi(\mathit{mayInc0}) \\
= & \langle \text{ definition of the translation } \rangle \\
 & \mathit{mayInc0}^T \\
= & \langle \text{ definition of } \mathit{mayInc0}^T \ \rangle \\
 & \psi(\mathit{id} \mathbin{?} \mathit{inc}) \\
= & \langle \text{ translation of (?) } \rangle \\
 & \psi(\mathit{id})\ `\mathit{union}`\ \psi(\mathit{inc}) \\
= & \langle \text{ as above } \rangle \\
 & \{\lambda x \rightarrow \{x\}\}\ `\mathit{union}`\ \{\lambda x \rightarrow \{x + 1\}\} \\
= & \langle \text{ beta equivalence } \rangle \\
 & \{\lambda x \rightarrow \{(\lambda n \rightarrow n)\ x\}\}\ `\mathit{union}`\ (\{\lambda x \rightarrow \{(\lambda n \rightarrow n + 1)\ x\}\}) \\
= & \langle \text{ first monad law (4.5) } \rangle \\
 & (\{\lambda n \rightarrow n\} \ggg\!= \lambda \hat{g}' \rightarrow \{\lambda x \rightarrow \{\hat{g}'\ x\}\}) \\
 & \quad `\mathit{union}`\ (\{\lambda n \rightarrow n + 1\} \ggg\!= \lambda \hat{g}' \rightarrow \{\lambda x \rightarrow \{\hat{g}'\ x\}\}) \\
= & \langle \text{ distributivity law (4.9) } \rangle \\
 & (\{\lambda n \rightarrow n\}\ `\mathit{union}`\ \{\lambda n \rightarrow n + 1\}) \ggg\!= \lambda \hat{g}' \rightarrow \{\lambda x \rightarrow \{\hat{g}'\ x\}\}
\end{array}$$

The function *mayInc1* is not multi-deterministic, as we will prove later. Yet, it can be shown that $\psi(\mathit{mayInc1}) = \{\lambda n \rightarrow \{n\}\ `\mathit{union}`\ \{n + 1\}\}$ using equational reasoning as above. From this representation we already see that nondeterminism occurs inside the function.

4.8.2 The Standard Example

We start with the same example that Wadler [1989] uses. Consider a polymorphic unary CuMin function

$$\mathit{fun} :: \forall a.[a] \rightarrow [a]$$

and some strict and multi-deterministic function-typed expression

$$g :: \mathsf{A} \to \mathsf{B}$$

from one concrete type to some other concrete type. The equation we want to prove is

$$\mathit{map}_{\mathsf{A},\mathsf{B}}\; g\; (\mathit{fun}_{\mathsf{A}}\; l) = \mathit{fun}_{\mathsf{B}}\; (\mathit{map}_{\mathsf{A},\mathsf{B}}\; g\; l) \tag{4.15}$$

where $l :: [\mathsf{A}]$ is some variable. We have already seen the function *map* in section 4.6.4.

Before we prove the claim, we first show that the restriction of g being multi-deterministic cannot be dropped. A counter example is given by

$$\mathit{fun}\; l = \textbf{case}\; l\; \textbf{of}\; \{[\,] \to [\,]; x : xs \to [x, x]\},$$

$g = \mathit{mayInc1}$ from the previous section and $l = [0]$ (we use list syntax instead of proper CuMin syntax, but the example should be clear nevertheless). The expression *map mayInc1* (*fun* [0]) results in the four lists $[0, 0], [0, 1], [1, 0], [1, 1]$. Here, the number 0 is first duplicated and then every copy is either increased or not increased. On the other hand, the expression *fun* (*map mayInc1* [0]) only produces $[0, 0]$ and $[1, 1]$. Here, the number 0 is either increased or not and the result is duplicated.
If *mayInc0* is used instead of *mayInc1*, both sides produce $[0, 0]$ and $[1, 1]$ only. This also shows that *mayInc0* and *mayInc1* do not coincide, even though *mayInc0 n* and *mayInc1 n* do for every n. The difference between both functions only becomes visible when both are used as an argument to a higher-order function like *map*.

The first step of the proof is to translate both sides of the claim (4.15) into SaLT. Both *fun* and *map* are functions, so we can use lemma 4.6.4 for the saturated applications. The left hand side translates to:

$$\begin{aligned}
&\psi(\mathit{map}_{\mathsf{A},\mathsf{B}}\; g\; (\mathit{fun}_{\mathsf{A}}\; l))\\
=\quad &\langle \text{ lemma 4.6.4 for } \mathit{map} \;\rangle\\
&\psi(g) \ggg\!= \lambda g' \to \psi(\mathit{fun}_{\mathsf{A}}\; l) \ggg\!= \lambda xs \to \mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\; g'\; xs\\
=\quad &\langle \text{ lemma 4.6.4 for } \mathit{fun} \;\rangle\\
&\psi(g) \ggg\!= \lambda g' \to \mathit{fun}^T{}_{\psi(\mathsf{A})}\; l \ggg\!= \lambda xs \to \mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\; g'\; xs
\end{aligned}$$

The right hand side translates to:

$$\begin{aligned}
&\psi(\mathit{fun}_{\mathsf{B}}\; (\mathit{map}_{\mathsf{A},\mathsf{B}}\; g\; l))\\
=\quad &\langle \text{ lemma 4.6.4 for } \mathit{fun} \;\rangle\\
&\psi(\mathit{map}_{\mathsf{A},\mathsf{B}}\; g\; l) \ggg\!= \lambda ys \to \mathit{fun}^T{}_{\psi(\mathsf{B})}\; ys\\
=\quad &\langle \text{ lemma 4.6.4 for } \mathit{map} \;\rangle\\
&\psi(g) \ggg\!= \lambda g' \to \mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\; g'\; l \ggg\!= \lambda ys \to \mathit{fun}^T{}_{\psi(\mathsf{B})}\; ys
\end{aligned}$$

So we are left with two SaLT expression we want to prove equivalent.

Now, we make use of the side condition by assuming some $\hat{g}$ to be given, such that

$$\psi(g) = \hat{g} \gg= \lambda\hat{g}' \rightarrow \{\lambda x \rightarrow \{\hat{g}'\ x\}\}$$

holds. Replacing $\psi(g)$ on both sides, the claim reduces to:

$$\begin{aligned}
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \{\lambda x \rightarrow \{\hat{g}'\ x\}\} \gg= \lambda g' \rightarrow \mathit{fun}^T{}_{\psi(\mathsf{A})}\ l \gg= \lambda xs \rightarrow \mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ g'\ xs \\
= \quad & \langle \text{ claim } \rangle \\
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \{\lambda x \rightarrow \{\hat{g}'\ x\}\} \gg= \lambda g' \rightarrow \mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ g'\ l \gg= \lambda ys \rightarrow \mathit{fun}^T{}_{\psi(\mathsf{B})}\ ys
\end{aligned}$$

Using the first monad law (4.5):

$$\begin{aligned}
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \mathit{fun}^T{}_{\psi(\mathsf{A})}\ l \gg= \lambda xs \rightarrow \mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ (\lambda x \rightarrow \{\hat{g}'\ x\})\ xs \\
= \quad & \langle \text{ claim } \rangle \\
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ (\lambda x \rightarrow \{\hat{g}'\ x\})\ l \gg= \lambda ys \rightarrow \mathit{fun}^T{}_{\psi(\mathsf{B})}\ ys
\end{aligned}$$

Now we use

$$\mathit{map}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ (\lambda x \rightarrow \{f\ x\})\ xs = \{\mathit{map}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ f\ xs\} \tag{4.12}$$

from section 4.6.4 to get:

$$\begin{aligned}
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \mathit{fun}^T{}_{\psi(\mathsf{A})}\ l \gg= \lambda xs \rightarrow \{\mathit{map}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}'\ xs\} \\
= \quad & \langle \text{ claim } \rangle \\
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \{\mathit{map}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}'\ l\} \gg= \lambda ys \rightarrow \mathit{fun}^T{}_{\psi(\mathsf{B})}\ ys
\end{aligned}$$

On the right hand side, we can use the first monad law again. The left hand side can be written more concisely using *smap*, which reduces the claim to:

$$\begin{aligned}
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \mathit{smap}_{[\psi(\mathsf{A})],[\psi(\mathsf{B})]}\ (\mathit{map}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}')\ (\mathit{fun}^T{}_{\psi(\mathsf{A})}\ l) \\
= \quad & \langle \text{ claim } \rangle \\
& \hat{g} \gg= \lambda\hat{g}' \rightarrow \mathit{fun}^T{}_{\psi(\mathsf{B})}\ (\mathit{map}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}'\ l)
\end{aligned}$$

At this point we want to use the free theorem for $\mathit{fun}^T :: \forall a.[a] \rightarrow \mathsf{Set}\ [a]$:

$$\mathit{smap}_{[\psi(\mathsf{A})],[\psi(\mathsf{B})]}\ (\mathit{map}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}')\ (\mathit{fun}^T{}_{\psi(\mathsf{A})}\ l) = \mathit{fun}^T{}_{\psi(\mathsf{B})}\ (\mathit{map}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}'\ l)$$

We have to check the side condition, which is $\hat{g}'$ being strict. However, $\hat{g}'$ is only a variable and has no meaning outside the lambda abstractions $\lambda\hat{g}' \rightarrow \ldots$. This requires a little detour.
We know that g is strict, i.e., $g\ \mathbf{failed}_{\mathsf{A}} = \mathbf{failed}_{\mathsf{B}}$. This translates to:

$$\psi(g) \gg= \lambda g' \rightarrow g'\ \mathbf{failed}_{\psi(\mathsf{A})} = \{\mathbf{failed}_{\psi(\mathsf{B})}\}$$

On the left we can use again that g is multi-deterministic:

$$\begin{aligned}
& \psi(g) \gg= \lambda g' \rightarrow g'\ \mathbf{failed}_{\psi(\mathsf{A})} \\
= \quad & \langle\ g \text{ is multi-deterministic } \rangle
\end{aligned}$$

$$
\begin{array}{ll}
& \hat{g} \ggg\!= \lambda\hat{g}' \to \{\,\lambda x \to \{\,\hat{g}'\ x\,\}\,\} \ggg\!= \lambda g' \to g'\ \mathbf{failed}_{\psi(\mathsf{A})} \\
= & \langle \text{ first monad law (4.5) } \rangle \\
& \hat{g} \ggg\!= \lambda\hat{g}' \to (\lambda x \to \{\,\hat{g}'\ x\,\})\ \mathbf{failed}_{\psi(\mathsf{A})} \\
= & \langle \text{ beta equivalence } \rangle \\
& \hat{g} \ggg\!= \lambda\hat{g}' \to \{\,\hat{g}'\ \mathbf{failed}_{\psi(\mathsf{A})}\,\}
\end{array}
$$

So, g being strict implies:

$$\hat{g} \ggg\!= \lambda\hat{g}' \to \{\,\hat{g}'\ \mathbf{failed}_{\psi(\mathsf{A})}\,\} = \{\,\mathbf{failed}_{\psi(\mathsf{B})}\,\}$$

On the semantic level this means

$$\bigcup_{\hat{\mathbf{g}}' \in [\![\hat{g}]\!]} \{\hat{\mathbf{g}}'(\bot)\} = \{\bot\}$$

and therefore every element $\hat{\mathbf{g}}'$ of $[\![\hat{g}]\!]$ is a strict function, i.e., satisfies $\hat{\mathbf{g}}'(\bot) = \bot$. Now, we can finish the proof. Because of the free theorem for $\mathit{fun}^T :: \forall a.[\,a\,] \to \mathsf{Set}\ [\,a\,]$, we have

$$[\![\mathit{smap}\ (\mathit{map}\ \hat{g}')\ (\mathit{fun}^T{}_{\psi(\mathsf{A})}\ l)]\!]_{\sigma[\hat{g}' \mapsto \hat{\mathbf{g}}']} = [\![\mathit{fun}^T{}_{\psi(\mathsf{B})}\ (\mathit{map}\ \hat{g}'\ l)]\!]_{\sigma[\hat{g}' \mapsto \hat{\mathbf{g}}']}$$

for every $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]$. So, we can take the union on both sides:

$$
\begin{array}{rcl}
\bigcup_{\hat{\mathbf{g}}' \in [\![\hat{g}]\!]} [\![\mathit{smap}\ (\mathit{map}\ \hat{g}')\ (\mathit{fun}^T{}_{\psi(\mathsf{A})}\ l)]\!]_{\sigma[\hat{g}' \mapsto \hat{\mathbf{g}}']} & = & \bigcup_{\hat{\mathbf{g}}' \in [\![\hat{g}]\!]} [\![\mathit{fun}^T{}_{\psi(\mathsf{B})}\ (\mathit{map}\ \hat{g}'\ l)]\!]_{\sigma[\hat{g}' \mapsto \hat{\mathbf{g}}']} \\
[\![\hat{g} \ggg\!= \lambda\hat{g}' \to \mathit{smap}\ (\mathit{map}\ \hat{g}')\ (\mathit{fun}^T{}_{\psi(\mathsf{A})}\ l)]\!]_{\sigma} & = & [\![\hat{g} \ggg\!= \lambda\hat{g}' \to \mathit{fun}^T{}_{\psi(\mathsf{B})}\ (\mathit{map}\ \hat{g}'\ l)]\!]_{\sigma}
\end{array}
$$

This is the last form the claim was given in.

This derivation is quite long, but most of it is generic: Multi-determinism is always exploited in the same way, i.e., replacing $\psi(g)$ by $\hat{g} \ggg\!= \lambda\hat{g}' \to \{\,\lambda x \to \{\,\hat{g}'\ x\,\}\,\}$ and using the first monad law (4.5). Also, the part about strictness of $\hat{\mathbf{g}}'$ resp. $\hat{g}'$ is true in general. In future proofs we will simply call the syntactic variable $\hat{g}'$ strict without descending to the semantic level again.

4.8.3 Second Example

Let $\mathit{fun} :: \forall a.a \to a \to (a, a)$ be a binary, polymorphic function, $g :: \mathsf{A} \to \mathsf{B}$ a strict and multi-deterministic function-typed term and $x, y :: \mathsf{A}$ two variables. Then

$$\mathit{pmap}_{\,\mathsf{A},\mathsf{B}}\ g\ (\mathit{fun}_{\,\mathsf{A}}\ x\ y) \quad = \quad \mathbf{let}\ g' = g\ \mathbf{in}\ \mathit{fun}_{\,\mathsf{B}}\ (g'\ x)\ (g'\ y) \qquad (4.16)$$

holds, where pmap is the following function:

$$
\begin{array}{l}
\mathit{pmap} :: \forall a\ b.(a \to b) \to (a, a) \to (b, b) \\
\mathit{pmap}\ f\ p = \mathbf{case}\ p\ \mathbf{of}\ \{\,(x, y) \to (f\ x, f\ y)\,\}
\end{array}
$$

The proof is very similar to the previous one. After translating the claim to SaLT, it reads:

$$\begin{array}{ll} & \psi(g) \gg\!= \lambda g' \to \mathit{fun}^T{}_{\psi(\mathsf{A})}\ x\ y \gg\!= \lambda p \to \mathit{pmap}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ g'\ p \\ = & \langle \text{ claim } \rangle \\ & \psi(g) \gg\!= \lambda g' \to g'\ x \gg\!= \lambda u \to g'\ y \gg\!= \lambda v \to \mathit{fun}^T{}_{\psi(\mathsf{B})}\ u\ v \end{array}$$

We use $\psi(g) = \hat{g} \gg\!= \lambda \hat{g}' \to \{\lambda z \to \{\hat{g}'\ z\}\}$ again and replace g':

$$\begin{array}{ll} & \hat{g} \gg\!= \lambda \hat{g}' \to \mathit{fun}^T{}_{\psi(\mathsf{A})}\ x\ y \gg\!= \lambda p \to \mathit{pmap}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ (\lambda z \to \{\hat{g}'\ z\})\ p \\ = & \langle \text{ claim } \rangle \\ & \hat{g} \gg\!= \lambda \hat{g}' \to \{\hat{g}'\ x\} \gg\!= \lambda u \to \{\hat{g}'\ y\} \gg\!= \lambda v \to \mathit{fun}^T{}_{\psi(\mathsf{B})}\ u\ v \end{array}$$

Using the same definition as above, *pmap* also is a SaLT function and

$$\mathit{pmap}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ (\lambda z \to \{f\ z\})\ p = \{\mathit{pmap}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ f\ p\} \tag{4.17}$$

holds (the proof is analogous to the proof of equation (4.12)). This and the first monad law can be used to simplify further:

$$\begin{array}{ll} & \hat{g} \gg\!= \lambda \hat{g}' \to \mathit{fun}^T{}_{\psi(\mathsf{A})}\ x\ y \gg\!= \lambda p \to \{\mathit{pmap}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}'\ p\} \\ = & \langle \text{ claim } \rangle \\ & \hat{g} \gg\!= \lambda \hat{g}' \to \mathit{fun}^T{}_{\psi(\mathsf{B})}\ (\hat{g}'\ x)\ (\hat{g}'\ y) \end{array}$$

Ignoring the $\hat{g} \gg\!= \lambda \hat{g}' \to$ on both sides, this simply is the free theorem for $\mathit{fun}^T :: \forall a.a \to a \to \mathsf{Set}\ (a, a)$. We can apply the free theorem 'inline' since $\hat{g}'$ is strict in the sense of only having strict denotations. So, the proof is finished.

In this example, the right hand side of equation (4.16) features a **let** binding introducing the variable g'. In SaLT (or Haskell), the free theorem for the type $\forall a.a \to a \to (a, a)$ simply is

$$\mathit{pmap}_{\mathsf{A},\mathsf{B}}\ g\ (\mathit{fun}_{\mathsf{A}}\ x\ y) \quad = \quad \mathit{fun}_{\mathsf{B}}\ (g\ x)\ (g\ y)$$

and introducing an additional variable g' is not necessary. This simpler equation does not hold in CuMin and $\mathit{fun} = \mathsf{Pair}$ and $g = \mathit{mayInc0}$ form a counter example. On the left hand side, g is shared automatically because it is an argument of *pmap*. On the right hand side, sharing is not automatic and has to be ensured artificially (like in equation (4.16)).

The free theorem remains true if x and y are arbitrary expressions instead of variables. The question of sharing is irrelevant here because they appear only once on either side of equation (4.16). The same is true for the function *fun*.

4.8.4 Handling the Data Class

When deriving free theorems for functions with Data class constraints, an additional complication arises. As an example, take functions

$$\mathit{fun} :: \forall a.\mathsf{Data}\ a \Rightarrow (a, a)$$

which can use logic variables of type a. Without the constraint, there is no function of that type other than $\mathit{fun} = \textbf{failed}_{(a,a)}$ or $\mathit{fun} = (\textbf{failed}_a, \textbf{failed}_a)$. With the constraint, functions like $\mathit{fun} = \textbf{let}\ x, y\ \textbf{free in}\ (x, y)$ and $\mathit{fun} = \textbf{let}\ x\ \textbf{free in}\ (x, x)$ become possible.

One way to deal with the constraint is to switch to inequational free theorems [Johann and Voigtländer, 2006]. For functions of the above type and strict and multi-deterministic terms $g :: \mathsf{A} \to \mathsf{B}$ the inequation

$$\mathit{pmap}\ g\ \mathit{fun}_{\mathsf{A}} \sqsubseteq \mathit{fun}_{\mathsf{B}}$$

holds.

In order to get equality, we have to add yet another side condition and require g to be multi-onto.

Definition 4.8.3. A CuMin term g::$\mathsf{A} \to \mathsf{B}$ is called *multi-onto* if it is multi-deterministic and there is a witness $\hat{g}$ satisfying:

$$\hat{g} \ggg\!= \lambda\hat{g}' \to \{(\hat{g}', \mathit{smap}\ \hat{g}'\ \textbf{unknown}_{\psi(\mathsf{A})})\} = \hat{g} \ggg\!= \lambda\hat{g}' \to \{(\hat{g}', \textbf{unknown}_{\psi(\mathsf{B})})\} \quad (4.18)$$

In particular, being multi-onto implies

$$\psi(g) \ggg\!= \lambda g' \to \textbf{unknown}_{\psi(\mathsf{A})} \ggg\!= g' = \textbf{unknown}_{\psi(\mathsf{B})} \quad (4.19)$$

which is the translation of the CuMin equivalence $g\ \textbf{unknown}_{\mathsf{A}} = \textbf{unknown}_{\mathsf{B}}$. This can be interpreted as being surjective.

Yet, being multi-onto is a strictly stronger property. Intuitively, it means that every element of $\hat{g}$ is surjective. For example, $\{\lambda x \to \mathsf{True}\}$ '*union*' $\{\lambda x \to \mathsf{False}\}$ is not the witness of a multi-onto function and equation (4.18) is not satisfied. The set consists of two functions, neither of which is surjective by itself. Both constituents together cover the type Bool, so equation (4.19) is satisfied.

Here, we face the same problem again we have already seen when considering strict constituents of $\hat{g}$: We cannot really talk about 'being an element of $\hat{g}$' because $\hat{g}$ is a syntactic set. In the case of strictness, we switched to the semantic level and used the notion of strictness of a semantic function. Here, we do something similar, but have to deal with an additional complication: The denotation of $\hat{g}$ always contains functions that are not whole in any sense, like $\lambda\mathbf{x}.\bot$, the function sending everything to $\bot$. At least the following is true:

Lemma 4.8.4. Let $g :: \mathsf{A} \to \mathsf{B}$ be a multi-onto CuMin term, whose multi-determinism is witnessed by $\hat{g}$. Then for every element $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]$ there is an element $\hat{\mathbf{g}}' \sqsubseteq \hat{\mathbf{g}}'' \in [\![\hat{g}]\!]$ which is whole.

Proof. We take the denotation of either side of equation (4.18):

$$\bigcup_{\hat{\mathbf{g}}' \in [\![\hat{g}]\!]} \downarrow\!(\hat{\mathbf{g}}', [\![\mathit{smap}\ \hat{g}'\ \textbf{unknown}_{\psi(\mathsf{A})}]\!]_{[\hat{g}' \mapsto \hat{\mathbf{g}}']}) = \bigcup_{\hat{\mathbf{g}}' \in [\![\hat{g}]\!]} \downarrow\!(\hat{\mathbf{g}}', [\![\textbf{unknown}_{\psi(\mathsf{B})}]\!])$$

For every $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]$ the tuple $(\hat{\mathbf{g}}', [\![\mathbf{unknown}_{\psi(\mathsf{B})}]\!])$ is an element of the right hand side of the above equation. So, it also has to be an element of the left hand side. This means there is some $\hat{\mathbf{g}}'' \in [\![\hat{g}]\!]$ with:

$$\left(\hat{\mathbf{g}}'', [\![\mathit{smap}\ \hat{g}'\ \mathbf{unknown}_{\psi(\mathsf{A})}]\!]_{[\hat{g}' \mapsto \hat{\mathbf{g}}'']}\right) \sqsupseteq (\hat{\mathbf{g}}', [\![\mathbf{unknown}_{\psi(\mathsf{B})}]\!])$$

Comparing the left entries gives $\hat{\mathbf{g}}' \sqsubseteq \hat{\mathbf{g}}''$ and comparing the right entries gives

$$[\![\mathit{smap}\ \hat{g}'\ \mathbf{unknown}_{\psi(\mathsf{A})}]\!]_{[\hat{g}' \mapsto \hat{\mathbf{g}}'']} \supseteq [\![\mathbf{unknown}_{\psi(\mathsf{B})}]\!]$$

since $\sqsubseteq$ is $\subseteq$ for sets. The above is equivalent to $\hat{\mathbf{g}}''$ being whole. □

Using the lemma, the proof is again analogous to the ones we have seen already. We translate the claim $\mathit{pmap}_{\mathsf{A},\mathsf{B}}\ g\ \mathit{fun}_{\mathsf{A}} = \mathit{fun}_{\mathsf{B}}$ into SaLT:

$$\psi(g) \ggg\!= \lambda g' \to \mathit{fun}^T{}_{\psi(\mathsf{A})} \ggg\!= \lambda p \to \mathit{pmap}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ g'\ p = \mathit{fun}^T{}_{\psi(\mathsf{B})}$$

Use multi-determinism:

$$\hat{g} \ggg\!= \lambda \hat{g}' \to \mathit{fun}^T{}_{\psi(\mathsf{A})} \ggg\!= \lambda p \to \mathit{pmap}^T{}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ (\lambda x \to \{\hat{g}'\ x\})\ p = \mathit{fun}^T{}_{\psi(\mathsf{B})}$$

Use equation (4.17):

$$\begin{aligned} \hat{g} \ggg\!= \lambda \hat{g}' \to \mathit{fun}^T{}_{\psi(\mathsf{A})} \ggg\!= \lambda p \to \{\mathit{pmap}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}'\ p\} &= \mathit{fun}^T{}_{\psi(\mathsf{B})} \\ \hat{g} \ggg\!= \lambda \hat{g}' \to \mathit{smap}\ (\mathit{pmap}_{\psi(\mathsf{A}),\psi(\mathsf{B})}\ \hat{g}')\ (\mathit{fun}^T{}_{\psi(\mathsf{A})}) &= \mathit{fun}^T{}_{\psi(\mathsf{B})} \end{aligned}$$

The free theorem for $\mathit{fun}^T :: \forall a.\mathsf{Data}\ a \Rightarrow \mathsf{Set}\ (a, a)$ is in fact

$$\mathit{smap}\ (\mathit{pmap}\ \hat{g}')\ \mathit{fun}^T = \mathit{fun}^T$$

for strict and whole $\hat{g}'$. The last step is to apply the semantics function and build the union over all $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]$:

$$\bigcup_{\hat{\mathbf{g}}' \in [\![\hat{g}]\!]} [\![\mathit{smap}\ (\mathit{pmap}\ \hat{g}')\ \mathit{fun}^T]\!]_{[\hat{g}' \mapsto \hat{\mathbf{g}}']} = [\![\mathit{fun}^T]\!]$$

In order to prove the claim, the union has to be taken over all $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]$. Yet, the free theorem only applies for the ones which are whole. This is the point where lemma 4.8.4 is needed. Since for every $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]$ there is a more defined, whole $\hat{\mathbf{g}}'' \in [\![\hat{g}]\!]$, it is enough to build the union over whole constituents.

4.9 Fold/Build Fusion for CuMin

4.9.1 Deterministic Case

We start with a very easy example of what fold/build fusion does in a deterministic setting. Suppose we want to compute the sum $n + (n-1) + \cdots + 0$ for any given n in

SaLT (we could use Haskell equally well). The direct approach is the following function, where we assume a minus primitive to be given:

$$\begin{array}{l}
\mathit{sumFrom} :: \mathsf{Nat} \to \mathsf{Nat} \\
\mathit{sumFrom}\ n = \mathbf{case}\ n == 0\ \mathbf{of} \\
\quad \mathsf{True}\ \ \to 0 \\
\quad \mathsf{False} \to n + \mathit{sumFrom}\ (n - 1)
\end{array}$$

An alternative approach would be to split the generation and processing of numbers:

$$\begin{array}{l}
\mathit{downFrom} :: \mathsf{Nat} \to [\mathsf{Nat}] \\
\mathit{downFrom}\ n = \mathbf{case}\ n == 0\ \mathbf{of} \\
\quad \mathsf{True}\ \ \to \mathsf{Nil}_{\,\mathsf{Nat}} \\
\quad \mathsf{False} \to \mathsf{Cons}_{\,\mathsf{Nat}}\ n\ (\mathit{downFrom}\ (n - 1)) \\
\mathit{sum} :: [\mathsf{Nat}] \to \mathsf{Nat} \\
\mathit{sum}\ l = \mathbf{case}\ l\ \mathbf{of} \\
\quad \mathsf{Nil} \qquad\quad \to 0 \\
\quad \mathsf{Cons}\ x\ \mathit{xs} \to x + \mathit{sum}\ \mathit{xs}
\end{array}$$

The expressions *sumFrom n* and *sum* (*downFrom n*) are semantically equivalent, so either gives the right result. The appeal of the latter approach is the reusability of the individual functions *sum* and *downFrom*. On the other hand, it requires more runtime than the direct approach because building and inspecting the list takes additional time. Fold/build fusion will allow us to write programs in a highly compositional way without making the programs slower. Instead the compiler will recognize the list to be a temporary data structure and omit it, creating essentially the directly recursive function *sumFrom*. Thus the advantages of both of the above are combined.

For the compiler to be able to detect the optimization potential, the functions have to be given in appropriate forms. We start with the consumer, which has to be given as a fold. Using the function

$$\begin{array}{l}
\mathit{foldr} :: \forall a\ b.(a \to b \to b) \to b \to [a] \to b \\
\mathit{foldr}\ f\ z\ l = \mathbf{case}\ l\ \mathbf{of}\ \{\mathsf{Nil} \to z; \mathsf{Cons}\ x\ \mathit{xs} \to f\ x\ (\mathit{foldr}_{\,a,b}\ f\ z\ \mathit{xs})\}
\end{array}$$

we can redefine *sum* equivalently as:

$$\begin{array}{l}
\mathit{sum} :: [\mathsf{Nat}] \to \mathsf{Nat} \\
\mathit{sum}\ l = \mathit{foldr}_{\,\mathsf{Nat},\mathsf{Nat}}\ (+)\ 0\ l
\end{array}$$

The function *downFrom* also has to be given in a special form. The idea is to replace the result type (i.e., [Nat]) by a type variable everywhere. Since the list constructors cannot be used anymore, surrogates have to be given:

$$\begin{array}{l}
\mathit{downFrom}' :: \forall b.\mathsf{Nat} \to (\mathsf{Nat} \to b \to b) \to b \to b \\
\mathit{downFrom}'\ n\ \mathit{cons}\ \mathit{nil} = \mathbf{case}\ n == 0\ \mathbf{of}
\end{array}$$

```
        True  → $nil$
        False → $cons\ n\ (downFrom'_{b}\ (n-1)\ cons\ nil)$
```

From the definition we see that:

$$downFrom'_{[\mathsf{Nat}]}\ n\ \mathsf{Cons}_{\mathsf{Nat}}\ \mathsf{Nil}_{\mathsf{Nat}} = downFrom\ n$$

We can thus redefine *downFrom* to rely on *downFrom′* without changing the meaning. In GHC, a function *build* is used:

```
    $build :: \forall a.(\forall b.(a \to b \to b) \to b \to b) \to [a]$
    $build\ g = g\ (:)\ [\,]$
```

The type variable b is only in scope within the type of the argument and forces g to have a type polymorphic in b. In contrast, a is a type variable of *build* and can be a concrete type in the function passed as g.
In Curry, CuMin and SaLT, there is no function *build*, because there are no higher rank types (i.e., types with ∀ quantifiers in argument types). The application of a function to Cons and Nil can of course be spelled out in full. Thus the condition of g being polymorphic is not enforced by the type of *build* anymore, but has to be required explicitly.
This condition is crucial because it enables us to instantiate g in two different ways. In our example, another way to call *downFrom′* is $downFrom'_{\mathsf{Nat}}\ n\ (+)\ 0$, which essentially is the directly recursive function *sumFrom*. This can be seen by comparing the two definitions.
In order to compute a sum, we can use $sum\ (downFrom\ n)$, which is equivalent to

$$foldr_{\mathsf{Nat},\mathsf{Nat}}\ (+)\ 0\ (downFrom'_{[\mathsf{Nat}]}\ n\ \mathsf{Cons}_{\mathsf{Nat}}\ \mathsf{Nil}_{\mathsf{Nat}}) \tag{4.20}$$

by inlining the functions *sum* and *downFrom*. In Haskell, the inlining is done automatically. Afterwards, an optimization rule fires and replaces the above by the equivalent but faster

$$downFrom'_{\mathsf{Nat}}\ n\ (+)\ 0. \tag{4.21}$$

For this optimization to be used, some preparation is necessary because only functions given via fold or build are suited. The GHC Prelude is tailored to provide many opportunities for fusing and by using these functions a lot, potential speedup is generated. Thus the only thing the ordinary programmer has to do to profit from fold/build fusion, is to avoid writing recursive functions on lists by hand and instead rely on provided functionality. On the other hand, the compiler needs an intricate mechanism to know when to inline, replace and fuse.

The theoretical foundation behind the optimization strategy is the following. Let A and B be concrete types and $c{::}\mathsf{A} \to \mathsf{B} \to \mathsf{B}$ and $n{::}\mathsf{B}$ terms. Let $g{::}\forall b.(\mathsf{A} \to b \to b) \to b \to b$ be a polymorphic function. Then the following equation holds:

$$foldr_{\mathsf{A},\mathsf{B}}\ c\ n\ (g_{[\mathsf{A}]}\ \mathsf{Cons}_{\mathsf{A}}\ \mathsf{Nil}_{\mathsf{A}}) = g_{\mathsf{B}}\ c\ n \tag{4.22}$$

Expressions (4.20) and (4.21) are equivalent as together they are an example of (4.22). This justifies the optimization in the case we discussed. The general idea is to use the right hand side instead of the left because the right hand side is thought to be faster. The proof of (4.22) is by using the free theorem for the type of g [Johann, 2003]. The function we want to use as a relation is $\mathit{foldr}_{\mathsf{A},\mathsf{B}}\ c\ n :: [\mathsf{A}] \to \mathsf{B}$, which is a strict function (resp. relation) because of the pattern matching in the function's definition.

First, we show that $\mathsf{Nil}_{\mathsf{A}} :: [\mathsf{A}]$ is related to $n :: \mathsf{B}$. Indeed, by the definition of *foldr* we have $\mathit{foldr}_{\mathsf{A},\mathsf{B}}\ c\ n\ \mathsf{Nil}_{\mathsf{A}} = n$. Second, we claim $\mathsf{Cons}_{\mathsf{A}} :: \mathsf{A} \to [\mathsf{A}] \to [\mathsf{A}]$ and $c :: \mathsf{A} \to \mathsf{B} \to \mathsf{B}$ to be related via $[\![\mathsf{A} \to b \to b]\!]_{[b \mapsto [\![\mathit{foldr}_{\mathsf{A},\mathsf{B}}\, c\ n]\!]]}$. We show this by applying both functions to related arguments. Any $a :: \mathsf{A}$ is clearly related only to itself and any $as :: [\mathsf{A}]$ is only related to $\mathit{foldr}_{\mathsf{A},\mathsf{B}}\ c\ n\ as$. Thus for $\mathsf{Cons}_{\mathsf{A}}$ and c to be related, only, $\mathsf{Cons}_{\mathsf{A}}\ a\ as :: [\mathsf{A}]$ and $c\ a\ (\mathit{foldr}_{\mathsf{A},\mathsf{B}}\ c\ n\ as) :: \mathsf{B}$ have to be related. This is true since

$$\mathit{foldr}_{\mathsf{A},\mathsf{B}}\ c\ n\ (\mathsf{Cons}_{\mathsf{A}}\ a\ as) = c\ a\ (\mathit{foldr}_{\mathsf{A},\mathsf{B}}\ c\ n\ as)$$

holds by the definition of *foldr*.

By parametricity, $g_{[\mathsf{A}]}$ and g_{B} are related via

$$[\![(\mathsf{A} \to b \to b) \to b \to b]\!]_{[b \mapsto [\![\mathit{foldr}_{\mathsf{A},\mathsf{B}}\, c\ n]\!]]}.$$

This means $g_{[\mathsf{A}]}$ and g_{B} produce related results for related arguments. We already know two pairs of related arguments, i.e., $\mathsf{Nil}_{\mathsf{A}}$ resp. $\mathsf{Cons}_{\mathsf{A}}$ and n resp. c. Thus $g_{[\mathsf{A}]}\ \mathsf{Cons}_{\mathsf{A}}\ \mathsf{Nil}_{\mathsf{A}} :: [\mathsf{A}]$ and $g_{\mathsf{B}}\ c\ n :: \mathsf{B}$ are related, which is what (4.22) states.

4.9.2 Counter-Example to Naive Approach

A naive approach to fold/build fusion in CuMin is to ask whether equation (4.22) is also true in CuMin, given that *foldr* is defined the same way it is defined in SaLT. The following counter-example shows that this is not the case. We define three functions:

$$\begin{aligned}
&\mathit{inc} :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{inc}\ n = n + 1\\
&\mathit{idOrInc} :: \mathsf{Bool} \to \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{idOrInc}\ b = \mathit{id}_{\mathsf{Nat}}\ ?\ \mathit{inc}\\
&\mathit{weird} :: \forall b.(\mathsf{Bool} \to b \to b) \to b \to b\\
&\mathit{weird}\ c\ n = \mathbf{let}\ h = c\ \mathsf{False}\ \mathbf{in}\ h\ (h\ n)
\end{aligned}$$

We will show that

$$\mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ (\mathit{weird}_{[\mathsf{Bool}]}\ \mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{Nil}_{\mathsf{Bool}}) \neq \mathit{weird}_{\mathsf{Nat}}\ \mathit{idOrInc}\ 0.$$

We rely on equational reasoning here despite having warned against doing so in CuMin earlier. The reason we deem this method apt here is that we do not actually need a proof. Since we compare two closed expressions, one can simply evaluate both in an interpreter. However, to give some impression of what happens without unfolding either of the formal semantics, equational reasoning seems fine.

We first calculate the left side, starting with the term in parentheses:

$$\begin{array}{ll}
& \mathit{weird}_{[\mathsf{Bool}]}\ \mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{Nil}_{\mathsf{Bool}} \\
= & \langle \text{ definition of } \mathit{weird} \ \rangle \\
& \mathbf{let}\ h = \mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathbf{in}\ h\ (h\ \mathsf{Nil}_{\mathsf{Bool}}) \\
= & \langle \text{ inlining } h \ \rangle \\
& \mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})
\end{array}$$

Here inlining h is fine, since it is given as a partial application. Then we can calculate the left hand side:

$$\begin{array}{ll}
& \mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})) \\
= & \langle \text{ definition of } \mathit{foldr} \text{ and pattern matching } \rangle \\
& \mathit{idOrInc}\ \mathsf{False}\ (\mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})) \\
= & \langle \text{ definition of } \mathit{idOrInc} \ \rangle \\
& (\mathit{id}_{\mathsf{Nat}}\ ?\ \mathit{inc})\ (\mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})) \\
= & \langle \text{ sharing of argument } \rangle \\
& \mathbf{let}\ n = \mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})\ \mathbf{in}\ (\mathit{id}_{\mathsf{Nat}}\ ?\ \mathit{inc})\ n \\
= & \langle \text{ application distributes over nondeterminism } \rangle \\
& \mathbf{let}\ n = \mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})\ \mathbf{in}\ \mathit{id}_{\mathsf{Nat}}\ n\ ?\ \mathit{inc}\ n \\
= & \langle \text{ definition of } \mathit{id} \text{ and } \mathit{inc} \ \rangle \\
& \mathbf{let}\ n = \mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})\ \mathbf{in}\ n\ ?\ n+1 \\
= & \langle \text{ definition of } \mathit{foldr} \text{ and pattern matching } \rangle \\
& \mathbf{let}\ n = \mathit{idOrInc}\ \mathsf{False}\ (\mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ \mathsf{Nil}_{\mathsf{Bool}})\ \mathbf{in}\ n\ ?\ n+1 \\
= & \langle \text{ definition of } \mathit{idOrInc} \ \rangle \\
& \mathbf{let}\ n = (\mathit{id}_{\mathsf{Nat}}\ ?\ \mathit{inc})\ (\mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ \mathsf{Nil}_{\mathsf{Bool}})\ \mathbf{in}\ n\ ?\ n+1 \\
= & \langle \text{ sharing of argument } \rangle \\
& \mathbf{let}\ m = \mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ \mathsf{Nil}_{\mathsf{Bool}}\,;\, n = (\mathit{id}_{\mathsf{Nat}}\ ?\ \mathit{inc})\ m\ \mathbf{in}\ n\ ?\ n+1 \\
= & \langle \text{ application distributes over nondeterminism } \rangle \\
& \mathbf{let}\ m = \mathit{foldr}_{\mathsf{Bool},\mathsf{Nat}}\ \mathit{idOrInc}\ 0\ \mathsf{Nil}_{\mathsf{Bool}}\,;\, n = m\ ?\ m+1\ \mathbf{in}\ n\ ?\ n+1 \\
= & \langle \text{ definition of } \mathit{foldr} \text{ and pattern matching } \rangle \\
& \mathbf{let}\ m = 0; n = m\ ?\ m+1\ \mathbf{in}\ n\ ?\ n+1 \\
= & \langle \text{ inlining } m \text{ and addition } \rangle \\
& \mathbf{let}\ n = 0\ ?\ 1\ \mathbf{in}\ n\ ?\ n+1 \\
= & \langle \text{ distributing local binding } \rangle \\
& (\mathbf{let}\ n = 0\ \mathbf{in}\ n\ ?\ n+1)\ ?\ (\mathbf{let}\ n = 1\ \mathbf{in}\ n\ ?\ n+1) \\
= & \langle \text{ inlining } n \text{ and addition } \rangle \\
& (0\ ?\ 1)\ ?\ (1\ ?\ 2) \\
= & \langle \text{ associativity and idempotence of } (?) \ \rangle \\
& 0\ ?\ 1\ ?\ 2
\end{array}$$

Now for the right hand side:

$$\begin{array}{ll}
& \mathit{weird}_{\mathsf{Nat}}\ \mathit{idOrInc}\ 0 \\
= & \langle \text{ definition of } \mathit{weird} \ \rangle
\end{array}$$

$$
\begin{array}{ll}
& \textbf{let}\ h = \mathit{idOrInc}\ \mathsf{False}\ \textbf{in}\ h\ (h\ 0) \\
= & \langle\ \text{definition of}\ \mathit{idOrInc}\ \rangle \\
& \textbf{let}\ h = \mathit{id}_{\mathsf{Nat}}\ ?\ \mathit{inc}\ \textbf{in}\ h\ (h\ 0) \\
= & \langle\ \text{distributing local binding}\ \rangle \\
& (\textbf{let}\ h = \mathit{id}_{\mathsf{Nat}}\ \textbf{in}\ h\ (h\ 0))\ ?\ (\textbf{let}\ h = \mathit{inc}\ \textbf{in}\ h\ (h\ 0)) \\
= & \langle\ \text{inlining}\ h\ \rangle \\
& \mathit{id}_{\mathsf{Nat}}\ (\mathit{id}_{\mathsf{Nat}})\ 0\ ?\ \mathit{inc}\ (\mathit{inc}\ 0) \\
= & \langle\ \text{and so on}\ \rangle \\
& 0\ ?\ 2
\end{array}
$$

So on the left hand side 1 is a possible result, while on the right hand side it is not. This disproves validity of equation (4.22) in CuMin.

This counter-example is admittedly rather far-fetched. If we wanted to abstract

$$\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ (\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{False}\ \mathsf{Nil}_{\mathsf{Bool}})$$

to enable fold/build fusion, we would probably go with

$\mathit{natural} :: \forall b.(\mathsf{Bool} \to b \to b) \to b \to b$
$\mathit{natural}\ c\ n = c\ \mathsf{False}\ (c\ \mathsf{False}\ n)$

rather than *weird*. In fact *natural* does not admit a counter-example – this will be a consequence of the positive claim in the next section. At this point, we can at least note that *natural* and *idOrInc* do not constitute a counter-example: On the right hand side, *idOrInc* would also be evaluated twice, thus 1 would become a result of this side as well. The second thing being rather odd in the above construction is the function *idOrInc*, which is a unary function. If *idOrInc* was defined as a binary function (i.e., $\mathit{idOrInc}\ b\ n = n\ ?\ n + 1$), the counter-example would not work either. In this case the right hand side would also admit 1 as a result.
Only the combination of *idOrInc*, which produces nondeterminism already after being applied once, and *weird*, which explicitly shares such an application, leads to the discrepancy between both sides. The refined approach will inhibit exactly this interaction.

4.9.3 Statement and Proof

In Haskell and SaLT, the types $\mathsf{X} \to \mathsf{Y} \to \mathsf{Z}$ and $(\mathsf{X}, \mathsf{Y}) \to \mathsf{Z}$ are nearly isomorphic via the functions:[7]

$\mathit{curry} :: \forall a\ b\ c.((a, b) \to c) \to a \to b \to c$
$\mathit{curry}\ f\ x\ y = f\ (x, y)$

[7] Here we encounter an unfortunate name clash. The language Curry (as well as the language Haskell) is named after Haskell Brooks Curry. Also named after him is the process of currying, which means turning a function in several variables into a unary function-valued function. This is what the function *curry* does, which can be found in the preludes of (among others) the languages Curry and Haskell. So the function *curry* is not specific to the language Curry and both having the same name can be understood as coincidental.

$$\begin{aligned}&\mathit{uncurry} :: \forall a\ b\ c.(a \to b \to c) \to (a, b) \to c\\&\mathit{uncurry}\ f\ p = \textbf{case}\ p\ \textbf{of}\ \{(x, y) \to f\ x\ y\}\end{aligned}$$

Here and in the following, we will write (x, y) instead of $\mathsf{Pair}_{\mathsf{A,B}}\, x\ \ y$ to denote the elements of pair types.
The two types $(\mathsf{X}, \mathsf{Y}) \to \mathsf{Z}$ and $\mathsf{X} \to \mathsf{Y} \to \mathsf{Z}$ are only nearly isomorphic, because a function of the former type can distinguish between (**failed**, **failed**) and **failed**. In Curry however, $\mathsf{X} \to \mathsf{Y} \to \mathsf{Z}$ contains many functions without a counterpart in $(\mathsf{X}, \mathsf{Y}) \to \mathsf{Z}$, for example *idOrInc*.
The idea now is to replace some types by their uncurried versions to exclude functions like *idOrInc*. To this end we redefine *foldr* with a different type and an uncurried version of Cons:

$$\begin{aligned}&\mathit{foldr} :: \forall a\ b.((a, b) \to b) \to b \to [a] \to b\\&\mathit{foldr}\ f\ z\ l = \textbf{case}\ l\ \textbf{of}\ \{\mathsf{Nil} \to z; \mathsf{Cons}\ x\ \mathit{xs} \to f\ (x, \mathit{foldr}_{\,a,b}\, f\ z\ \mathit{xs})\}\\&\mathit{uncCons} :: \forall a.(a, [a]) \to [a]\\&\mathit{uncCons}\ p = \textbf{case}\ p\ \textbf{of}\ \{(a, \mathit{as}) \to \mathsf{Cons}_{\,a}\ a\ \mathit{as}\}\end{aligned}$$

Now we can state the claim: For concrete types A and B, terms $c :: (\mathsf{A}, \mathsf{B}) \to \mathsf{B}$, $n :: \mathsf{B}$ and a polymorphic function

$$g :: \forall b.((\mathsf{A}, b) \to b) \to b \to b$$

the equation

$$\mathit{foldr}_{\,\mathsf{A,B}}\ c\ n\ (g_{\,[\mathsf{A}]}\ \mathit{uncCons}_{\,\mathsf{A}}\ \mathsf{Nil}_{\,\mathsf{A}}) = g_{\,\mathsf{B}}\ c\ n \tag{4.23}$$

holds.
Note that c can still be an nondeterministic function. Being given in uncurried form only forces both arguments to be given at once. Thus, no nondeterminism can occur after applying c to only one argument. Yet, when both arguments are given, c *can* branch.
Instead of changing the types, c can be restricted via its arity. The original, curried version of the statement also holds for CuMin, if c is required to be an actual function (and not just function-typed) and have arity at least two.

The proof works slightly differently than the proofs in section 4.8. After the usual translation into SaLT and some simplifications, a result by Ghani and Johann [2007] is invoked: monadic fold/build fusion.
We translate *foldr* and *uncCons*:

$$\begin{aligned}&\mathit{foldr}^T :: \forall a\ b.((a, b) \to \mathsf{Set}\ b) \to b \to [a] \to \mathsf{Set}\ b\\&\mathit{foldr}^T\ f\ z\ l = \textbf{case}\ l\ \textbf{of}\ \{\mathsf{Nil} \to \{z\}; \mathsf{Cons}\ x\ \mathit{xs} \to \mathit{foldr}^T{}_{a,b}\, f\ z\ \mathit{xs} \ggg\!= \lambda y \to f\ (x, y)\}\\&\mathit{uncCons}^T :: \forall a.(a, [a]) \to \mathsf{Set}\ [a]\\&\mathit{uncCons}^T\ p = \textbf{case}\ p\ \textbf{of}\ \{(a, \mathit{as}) \to \{\mathsf{Cons}_{\,a}\ a\ \mathit{as}\}\}\end{aligned}$$

We do not know whether g is a function symbol or a polymorphic expression, so we do not use g^T. Instead we define a function *gee*, that serves the same purpose but only relies on the translation of the expression $g_{\,b}$:

$$\begin{aligned}&gee :: \forall b.((\psi(\mathsf{A}), b) \rightarrow \mathsf{Set}\ b) \rightarrow b \rightarrow \mathsf{Set}\ b\\&gee\ c\ n = \psi(g_{\ b}) \ggg\!= \lambda g' \rightarrow g'\ c \ggg\!= \lambda h \rightarrow h\ n\end{aligned}$$

The translation of the claim's right hand side is:

$$\begin{aligned}&\psi(g_{\ \mathsf{B}}\ c\ n)\\=\ &\\&\psi(g_{\ \mathsf{B}}\ c) \ggg\!= \lambda h \rightarrow h\ n\\=\ &\\&\psi(g_{\ \mathsf{B}}) \ggg\!= \lambda g' \rightarrow g'\ c \ggg\!= \lambda h \rightarrow h\ n\\=\ &\\&gee_{\ \psi(\mathsf{B})}\ c\ n\end{aligned}$$

For the left hand side we find:

$$\begin{aligned}&\psi(foldr_{\ \mathsf{A,B}}\ c\ n\ (g_{\ [\mathsf{A}]}\ uncCons_{\ \mathsf{A}}\ \mathsf{Nil}_{\ \mathsf{A}}))\\=\ &\\&\psi(g_{\ [\mathsf{A}]}\ uncCons_{\ \mathsf{A}}\ \mathsf{Nil}_{\ \mathsf{A}}) \ggg\!= \lambda xs \rightarrow foldr^{T}{}_{\ \psi(\mathsf{A}),\psi(\mathsf{B})}\ c\ n\ xs\\=\ &\\&\psi(g_{\ [\mathsf{A}]}\ uncCons_{\ \mathsf{A}}\ \mathsf{Nil}_{\ \mathsf{A}}) \ggg\!= foldr^{T}{}_{\ \psi(\mathsf{A}),\psi(\mathsf{B})}\ c\ n\\=\ &\\&\psi(g_{\ [\mathsf{A}]}) \ggg\!= \lambda g' \rightarrow wrap_1\ (uncCons^{T}{}_{\ \psi(\mathsf{A})}) \ggg\!= \lambda u \rightarrow g'\ u \ggg\!= \lambda h \rightarrow\\&\quad h\ \mathsf{Nil}_{\ \psi(\mathsf{A})} \ggg\!= foldr^{T}{}_{\ \psi(\mathsf{A}),\psi(\mathsf{B})}\ c\ n\\=\ &\\&\psi(g_{\ [\mathsf{A}]}) \ggg\!= \lambda g' \rightarrow \{uncCons^{T}{}_{\ \psi(\mathsf{A})}\} \ggg\!= \lambda u \rightarrow g'\ u \ggg\!= \lambda h \rightarrow\\&\quad h\ \mathsf{Nil}_{\ \psi(\mathsf{A})} \ggg\!= foldr^{T}{}_{\ \psi(\mathsf{A}),\psi(\mathsf{B})}\ c\ n\\=\ &\\&\psi(g_{\ [\mathsf{A}]}) \ggg\!= \lambda g' \rightarrow g'\ (uncCons^{T}{}_{\ \psi(\mathsf{A})}) \ggg\!= \lambda h \rightarrow\\&\quad h\ \mathsf{Nil}_{\ \psi(\mathsf{A})} \ggg\!= foldr^{T}{}_{\ \psi(\mathsf{A}),\psi(\mathsf{B})}\ c\ n\\=\ &\\&gee_{\ [\psi(\mathsf{A})]}\ uncCons^{T}{}_{\ \psi(\mathsf{A})}\ \mathsf{Nil}_{\ \psi(\mathsf{A})} \ggg\!= foldr^{T}{}_{\ \psi(\mathsf{A}),\psi(\mathsf{B})}\ c\ n\end{aligned}$$

So it is sufficient to prove

$$gee_{\ [\psi(\mathsf{A})]}\ uncCons^{T}{}_{\ \psi(\mathsf{A})}\ \mathsf{Nil}_{\ \psi(\mathsf{A})} \ggg\!= foldr^{T}{}_{\ \psi(\mathsf{A}),\psi(\mathsf{B})}\ c\ n = gee_{\ \psi(\mathsf{B})}\ c\ n$$

for the function *gee*. This equation is known to hold and can be found in [Ghani and Johann, 2007].

Chapter 5

Conclusion and Outlook

Idiomatic Traversals

The case of idiomatic traversals, studied in chapter 3, seems to be closed for the most part. The correspondence between traversable data structures and finitary containers connects interface and intuition. We now have a clear picture of which structures are traversable and which methods are suitable for doing so. When using effectful traversals in programs, we can reason about them by using concepts usually associated to finitary containers, like shape, contents, and the order of the entries. When implementing effectful traversals we have an equally clear picture of what we are trying to achieve and whether functions will satisfy the traversal laws.

This situation is reflected in the fact that idioms and traversals are now part of the Haskell prelude, where the Traversable class is also given a set of laws. Not only have the classes been moved there, they also have become an integral part of the prelude: Many functions that only used to work for lists are now abstracted over the container and work for all foldable or all traversable type constructors. Applicative appears both as a superclass of Monad and as a typeclass constraint in the class definition of Traversable.

Yet, a gap remains between theory and practice: We have assumed the language to be total, which does not apply to Haskell. The difference does not matter as long as only total values are involved. On the other hand, reasoning about infinite lists and trees would be very useful. Then again, this point is not specific to the discussion of idiomatic traversals, but applies to equational reasoning in general.

While the specific object of study, idiomatic traversals, are well-understood, questions remain about related concepts. The Foldable class still does not have any laws, though there have been discussions whether it should. Conversely, we can think of structures with a richer interface than Traversable, like containers that also allow to add and delete entries. Are there useful abstractions for these data structures and if so, how can we reason about them? Yet broader, what other patterns for effectful programming in Haskell would we like to understand better?

Functional-Logic Programming

Chapter 4 features a number of different contributions. The functional-style denotational semantics for Curry has been simplified substantially. More importantly, a formal connection to other semantics, like the operational semantics [Albert et al., 2005] or the rewriting semantics [González-Moreno et al., 1999] has been established. The language SaLT has been introduced as a tool for reasoning about Curry. Using the translation to SaLT, equational reasoning about Curry is facilitated. Parametricity [Reynolds, 1983] and free theorems [Wadler, 1989] can be derived for Curry. In particular, the necessary side conditions given in [Christiansen et al., 2010] have been formalized. Finally, short cut fusion [Gill et al., 1993] has been proved to hold for Curry under very weak side conditions.
This last point is of particular interest, as it leads towards an application of type based reasoning about Curry for the first time. There seems to be no working implementation yet, but I would be delighted to see short cut fusion in action in future versions of Curry.

Yet, a lot remains to be done. Most notably, there is a remaining gap between actual Curry and the simplified version CuMin. Three features stand out: Unrestricted use of logic variables, recursive let bindings and encapsulated search.
Type based reasoning for Curry and free theorems in particular rely on restricting the use of logic variables and constraints. Free theorems can still be proved if logic variables for arbitrary types are allowed. Yet, the side conditions that become necessary (multiontoness in particular) are rarely satisfied and hard to check in any automated way.
The alternative is to introduce a Data typeclass into actual Curry and thereby restrict the use of logic variables to types which have a suitable generator function. This requires extending the type checker as well as changing the libraries, which will break existing code. However, both is already necessary to introduce typeclasses other than Data (e.g., Eq or Num). So, the best opportunity for introducing Data is to add it simultaneously.
Recursive let bindings entered the discussion for a very different reason – a compromise due to the method. Fully compositional, functional-style denotational semantics are incapable of handling recursive let bindings correctly [Christiansen et al., 2011b]. So, they were excluded from the whole development. Many of the results that rely on this denotational semantics may still hold for a language with recursive let bindings, but the presented methods are not able to prove that. There seems to be no easy way to handle this discrepancy. Of course, one could switch to a different overall approach and reconstruct the whole development in an operational or rewriting context, but this would require a lot of additional effort.
The third main difference between Curry and CuMin is encapsulated search. This feature has been left out of the discussion for now, but it is at least conceivable that it could be added. The functional-style denotational semantics by Christiansen et al. [2013] can be considered a first step. Yet, in order to reproduce the rest of the development (proving the semantics to be equivalent to some other semantics and deriving and applying parametricity) additional effort is again required.

The translation from CuMin to SaLT has proved to be a valuable tool for studying nondeterminism in Curry. Yet, translating back and forth is tedious. A more convenient solution could be an annotated version of Curry, that unites the additional information SaLT provides with the original Curry syntax. In particular, this could lead to a simpler formalization of multi-determinism.

List of Figures

Bibliography

M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *FOSSACS, Proceedings*, volume 2620 of *LNCS*, pages 23–38. Springer-Verlag, 2003.

S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Oxford University Press, 1994.

E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *ICLP, Proceedings*, volume 4079 of *LNCS*, pages 87–101. Springer-Verlag, 2006.

J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *ESOP, Proceedings*, volume 6012 of *LNCS*, pages 125–144. Springer-Verlag, 2010.

R. Bird, J. Gibbons, S. Mehner, T. Schrijvers, and J. Voigtländer. Understanding idiomatic traversals backwards and forwards. In *Haskell Symposium, Proceedings*, pages 25–36. ACM Press, 2013.

M. Böhm. Erweiterung von Curry um Typklassen. Master's thesis, University of Kiel, 2013.

B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *APLAS, Proceedings*, volume 4807 of *LNCS*, pages 122–138. Springer-Verlag, 2007.

B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming functional logic programs into monadic functional programs. In *WFLP 2010, Revised Selected Papers*, volume 6559 of *LNCS*, pages 30–47. Springer-Verlag, 2011.

J. Breitner. The correctness of Launchbury's natural semantics for lazy evaluation. *Computing Research Repository*, abs/1405.3099, 2014.

P. Capriotti and A. Kaposi. Free applicative functors. In *MSFP, Proceedings*, volume 153 of *EPTCS*, pages 2–30. Open Publishing Association, 2014.

J. Christiansen, D. Seidel, and J. Voigtländer. Free theorems for functional logic programs. In *PLPV, Proceedings*, pages 39–48. ACM Press, 2010.

J. Christiansen, D. Seidel, and J. Voigtländer. An adequate, denotational, functional-style semantics for Typed FlatCurry. In *WFLP 2010, Revised Selected Papers*, volume 6559 of *LNCS*, pages 119–136. Springer-Verlag, 2011a.

J. Christiansen, D. Seidel, and J. Voigtländer. An adequate, denotational, functional-style semantics for Typed FlatCurry without Letrec. Technical report, University of Bonn, 2011b. `http://www.iai.uni-bonn.de/~jv/IAI-TR-2011-1.pdf`.

J. Christiansen, M. Hanus, F. Reck, and D. Seidel. A semantics for weakly encapsulated search in functional logic programs. In *PPDP, Proceedings*, pages 49–60. ACM Press, 2013.

A. Colmerauer and P. Roussel. The birth of Prolog. In *History of Programming languages*, volume 2, pages 331–367. ACM Press, 1996.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *TYPES, Revised Selected Papers*, volume 3085 of *LNCS*, pages 210–225. Springer-Verlag, 2003.

N. Ghani and P. Johann. Monadic augment and generalised short cut fusion. *Journal of Functional Programming*, 17(6):731–776, 2007.

J. Gibbons and R. Hinze. Just do it: Simple monadic equational reasoning. In *ICFP, Proceedings*, pages 2–14. ACM Press, 2011.

J. Gibbons and B. Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19(3–4):377–402, 2009.

A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *FPCA, Proceedings*, pages 223–232. ACM Press, 1993.

J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2).* 2006. Available at `http://curry-language.org`.

M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics — Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 123–168. Springer-Verlag, 2013.

M. Hanus and B. Peemöller. A partial evaluator for Curry. In *WFLP, Proceedings*, pages 55–71. University of Halle-Wittenberg, 2014.

P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *HOPL, Proceedings*, pages 12-1–12-55. ACM Press, 2007.

G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. 2008. URL `http://www.cs.nott.ac.uk/~gmh/effects-extended.pdf`.

M. Jaskelioff and R. O'Connor. A representation theorem for second-order functionals. *Journal of Functional Programming*, 25, 2015.

M. Jaskelioff and O. Rypacek. An investigation of the laws of traversals. In *MSFP, Proceedings*, volume 76 of *EPTCS*, pages 40–49. Open Publishing Association, 2012.

P. Johann. Short cut fusion is correct. *Journal of Functional Programming*, 13(4): 797–814, 2003.

P. Johann and J. Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.

J. Launchbury. A natural semantics for lazy evaluation. In *POPL, Proceedings*, pages 144–154. ACM Press, 1993.

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL, Proceedings*, pages 333–343. ACM Press, 1995.

F.J. López-Fraguas and J. Sánchez-Hernández. *TOY*: A multiparadigm declarative system. In *RTA, Proceedings*, volume 1631 of *LNCS*, pages 244–247. Springer-Verlag, 1999.

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Equivalence of two formal semantics for functional logic programs. In *PROLE 2006, Proceedings*, volume 188 of *ENTCS*, pages 117–142. Elsevier, 2007.

K. Matsuda and M. Wang. "Bidirectionalization for free" for monomorphic transformations. *Science of Computer Programming*, 111(P1):79–109, 2015.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

S. Mehner, D. Seidel, L. Straßburger, and J. Voigtländer. Parametricity and proving free theorems for functional-logic languages. In *PPDP, Proceedings*, pages 19–30. ACM Press, 2014.

R.E. Møgelberg and A. Simpson. Relational parametricity for computational effects. *Logical Methods in Computer Science*, 5(3), 2009.

E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1): 55–92, 1991.

E. Moggi, G. Bellè, and C. Barry Jay. Monads, shapely functors, and traversals. In *CTCS, Proceedings*, volume 29 of *ENTCS*, pages 187–208. Elsevier, 1999.

R. Paterson. Constructing applicative functors. In *MPC, Proceedings*, volume 7342 of *LNCS*, pages 300–323. Springer-Verlag, 2012.

S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2002.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000. Reprint of lecture notes from 1967.

F. Thorand. Implementing denotational semantics of a functional-logic language and a functional language with sets. Bachelor's thesis, University of Bonn, 2015.

J. Voigtländer. Bidirectionalization for free! In *POPL, Proceedings*, pages 165–176. ACM Press, 2009.

J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, 388(1–3):290–318, 2007.

P. Wadler. How to replace failure by a list of successes. In *FPCA, Proceedings*, pages 113–128. Springer-Verlag, 1985.

P. Wadler. Theorems for free! In *FPCA, Proceedings*, pages 347–359. ACM Press, 1989.

P. Wadler. The essence of functional programming. In *POPL, Proceedings*, pages 1–14. ACM Press, 1992.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL, Proceedings*, pages 60–76. ACM Press, 1989.

F. Zaiser. Implementing an operational semantics and nondeterminism analysis for a functional-logic language. Bachelor's thesis, University of Bonn, 2015.