Rheinische Friedrich-Wilhelms-Universität Bonn

Bachelor Thesis

Implementing an Operational Semantics and Nondeterminism Analysis for a Functional-Logic Language

Fabian Moritz Frank Zaiser
Student number: XXXXXX
STREET NUMBER
D-53121 Bonn

November 16, 2015

Supervisor: Jun.-Prof. Dr. rer. nat. habil. Janis Voigtländer

Institute for Computer Science, Department III
Römerstraße 164
D-53117 Bonn

Eigenständigkeitserklärung

Hiermit erkläre ich, Fabian Moritz Frank Zaiser, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Ort, Datum Unterschrift

Abstract

Functional-logic languages like Curry aim to combine the strengths of both functional and logic languages. In order to establish free theorems for such languages, the authors of [11] have focused their attention on a sublanguage they call *CuMin*, which can be translated to another language called *SaLT* where logic features like nondeterminism are more explicit.

In this thesis, I will give a detailed exposition of the two languages. Moreover, I will document the implementation of an operational semantics for CuMin and of a translation algorithm that transforms CuMin programs into equivalent SaLT programs. I will describe how the generated SaLT code can be simplified and how it can be used to analyze the nondeterminism in the original CuMin program.

Contents

Chapter 1

Introduction

In the field of declarative programming languages, there are two important paradigms. On the one hand, functional programming languages like *Haskell* are based on the lambda calculus and commonly include features like first-class functions, algebraic data types, and powerful type systems. On the other hand, logic programming languages like *Prolog* are based on first-order logic and allow computing with partial information, constraint solving, and nondeterministic search for solutions.

The declarative programming language *Curry* [6] aims to combine the central features of both paradigms in one language. Curry is based on (a subset of) Haskell but integrates logic variables and nondeterministic search. This functional-logic language is well-known, actively researched and has various implementations, such as PAKCS (Portland Aachen Kiel Curry System) [3] or KiCS2 (Kiel Curry System) [5]. Therefore, it makes sense to use it as an introduction to this paradigm.

I will then describe the languages CuMin (Curry Minor) and SaLT (Set and Lambda Terms), both of which were introduced in [11]. The former is a sublanguage of Curry including its characteristic features but with fewer syntactic constructs, which simplifies the study of the language. The latter is essentially a lambda-calculus where, in contrast to CuMin, nondeterminism is made explicit using an abstract set type. It behaves more like a functional language and makes it easier, for example, to derive *free theorems* [17], which was the original motivation to introduce these two languages in [11].[1]

In the later sections of this chapter, I present an example-based overview of these languages. A more formal description can be found in chapter 2. But first, let me give a brief introduction to Haskell, for one thing because it is the language that Curry is built upon, and for another because the implementation part of this thesis was carried out in Haskell.

1.1 Haskell

Haskell is a purely functional, statically typed, lazy language. This means that Haskell functions are more like mathematical functions than procedures in imperative languages, in that they always produce the same output for the same input, and side-effects are restricted. Every expression is typed and the type checker ensures that the types match up, thus catching potential bugs at compile time. Evaluation of terms is delayed by default until the value is needed, enabling, for example, the use of infinite data structures.

[1] Free theorems are theorems about a function that can be derived solely from its type signature.

1.1.1 Functions

A function definition in Haskell consists of an (optional) type signature and one or more defining equations. The following examples should illustrate the concept.

```
double :: Int → Int
double x = 2 ∗ x

factorial :: Int → Int
factorial 0 = 1
factorial n = n ∗ factorial (n − 1)

min :: Int → Int → Int
min x y = if x < y then x else y
```

One can look at *min* as a function that takes two arguments of type Int and returns an integer. But actually, *min* is a function that takes one argument of type Int and returns a function of type Int → Int. This makes *min* a *curried* function. It becomes more apparent when *min* is written with an explicit lambda abstraction.

```
min :: Int → (Int → Int)
min x = λy → if x < y then x else y
```

One can now use *min* 0 :: Int → Int as a cut-off function that returns its argument if it is negative and otherwise returns 0.

Functions can also take other functions as their arguments. In this case, they are called *higher-order functions*. The following definition is an example.

```
applyTwice :: (Int → Int) → Int → Int
applyTwice f x = f (f x)
```

Applying this function to *double* results in a new function which applies *double* to its argument twice, i.e. multiplies its argument by four.

```
quadruple :: Int → Int
quadruple = applyTwice double
```

1.1.2 Polymorphism

Actually, it is not necessary for *applyTwice* to have its type specialized to Int. It should work for any type. Indeed, Haskell allows us to write the following more general definition.

```
applyTwice :: (a → a) → a → a
```

Here, *a* is a type variable and can be instantiated with any type. When using this function with *double* as the argument, *a* will be instantiated to the type Int in order to match the type of *double*.

Polymorphic functions are very common in Haskell. Another simple example is the identity function.

```
id :: a → a
id x = x
```

As a matter of fact, it is essentially the only function with this type signature. (This is a consequence of the free theorem for *id*.) Another very common example is function composition.

$$
\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to a \to c \\
&(\circ)\; f\; g\; x = f\; (g\; x)
\end{aligned}
$$

Here, $(\circ)$ is a higher-order function that composes its two function arguments. It can also be written as an infix operator: $f \circ g$ stands for $(\circ)\; f\; g$. Hence, one can write *applyTwice* by composing the given function with itself.

$$
\mathit{applyTwice}\; f = f \circ f
$$

1.1.3 Algebraic Data Types

Non-primitive data types in Haskell take the form of *algebraic data types (ADTs).* They are defined by giving a name to the new type and listing all its constructors, separated by vertical bars. Each constructor has a name and takes zero or more arguments, whose types have to be specified.

```
data Bool = False | True
data IntTree = Leaf Int | Node IntTree IntTree
```

The first data type has two nullary constructors, False and True. These constitute its only values. The second data type specifies a binary tree whose leafs are annotated with an integer. Its values include, for instance, Leaf 0 and Node (Leaf 10) (Node (Leaf 7) (Leaf 2)).

Data types can also be polymorphic. To this end, type variables can be added after the name of the ADT and the types on the right-hand side can use them.

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

Using this definition, Tree Int is equivalent to IntTree from above. Singly-linked lists can also be represented as ADTs:

```
data List a = Nil | Cons a (List a)
```

Here, Nil represents the empty list and Cons prepends one element to another list. As an example, the list 1,2,3 is represented by Cons 1 (Cons 2 (Cons 3 Nil)) :: List Int. In Haskell, there is special syntax for lists: List a is written $[a]$, Nil means $[\,]$ and Cons x xs corresponds to $x : xs$.

1.1.4 Pattern Matching

Constructors of algebraic data types give a way to create values of ADTs. Conversely, pattern matching is used to find out which constructor a value was built with and what its arguments are. In the simplest form, this is achieved with a **case** expression.

```
map :: (a → b) → List a → List b
map f list = case list of
  Nil         → Nil
  Cons x rest → Cons (f x) (map f rest)
```

The function *map* applies a function to every element of a list. It inspects the list by matching it against all possible constructors. If it is the empty list Nil, the result is also Nil. If it begins with an element x, the function is applied to x and recursively to the rest of the list. Haskell has special syntactic sugar for pattern matches on function arguments, which allows *map* to be written as follows.

```
map f Nil           = Nil
map f (Cons x rest) = Cons (f x) (map f rest)
```

1.1.5 Lazy Evaluation

As mentioned above, Haskell's evaluation strategy is lazy. That means it evaluates expressions only as much as is necessary. Broadly speaking, the only way to evaluate something is by pattern matching on it. Consider the following program.

```
zeros :: List Int
zeros = Cons 0 zeros

take :: Int → List a → List a
take 0 _             = Nil
take n (Cons x xs) = Cons x (take (n − 1) xs)
```

It defines *zeros*, an infinite list of zeros, and *take*, a function returning the first elements of a list, which can be used like this.

```
take 2 (Cons 1 (Cons 2 (Cons 3 Nil))) ≡ Cons 1 (Cons 2 Nil)
```

But even an expression like *take* 2 *zeros* is evaluated in finite time and works as desired because of lazy evaluation.

```
take 2 zeros
 ≡ take 2 (Cons 0 zeros)
 ≡ Cons 0 (take 1 zeros)
 ≡ Cons 0 (Cons 0 (take 0 zeros)
 ≡ Cons 0 (Cons 0 Nil)
```

Since *take* does not require the value of *zeros* in the last step because *take* 0 does not pattern match on the list, therefore *zeros* is not evaluated further, thus avoiding an infinite loop.

1.2 Curry

Curry is almost a superset of Haskell but also incorporates nondeterministic functions and logic variables. As an example, the following choice function can return any of its two arguments.

```
choose x y = x
choose x y = y
```

With this definition, *choose* 0 1 has two values, 0 and 1. A Curry interpreter will display both of them, when asked for all solutions (similarly to Prolog). As another application, consider this definition of a nondeterministic list insertion and permutation function. (Here, the conventional list syntax with [ ] and : is used instead of the custom data type from above with Nil and Cons.)

```
insert x [ ]              = [x]
insert x (first : rest) = choose (x : first : rest) (first : insert x rest)

permute [ ]              = [ ]
permute (first : rest) = insert first (permute rest)
```

To insert an object at any place in a list, *insert* puts it at the beginning or recursively inserts it later in the list. The function call *insert* 0 $[3, 4]$ results in $[0, 3, 4]$, $[3, 0, 4]$ or $[3, 4, 0]$. Moreover, *permute* uses this function to insert the first element in the recursively permuted rest. Thus it nondeterministically computes all permutations of a list.

Another feature of Curry are logic variables. Such variables are not assigned a value but are instead declared with the keyword **free**. The interpreter then searches for suitable assignments that satisfy the given constraints.

```
append []       ys = ys
append (x : xs) ys = x : append xs ys
last list | list =:= append init [e] = e where init, e free
```

The *append* function simply concatenates two lists and does not make use of nondeterminism. To retrieve the last element of a list, *last* specifies that this last element e must satisfy the constraint *list* =:= *append init* $[e]$, i.e. that appending $[e]$ to some list *init* must yield the original list. When this constraint is satisfied, e is returned.

1.3 CuMin

CuMin (Curry Minor) is a simple Curry dialect that lacks most of the syntactic sugar in Curry programs like **where** clauses and lambda abstractions. It essentially has only one logic programming feature: logic variables. However, this is enough to achieve a lot of what is possible in Curry, but at the expense of conciseness. For instance, the *choose* function from above can be translated to CuMin like this.

```
choose :: ∀a.a → a → a
choose x y = let choice :: Bool free in
  case choice of
    False → x
    True → y
```

The **let**. . . **free** binding introduces a logic variable *choice* that nondeterministically assumes all boolean values. As a consequence, both case alternatives can be evaluated the function can nondeterministically return either of its arguments. The following CuMin functions correspond to the Curry functions from above.

```
insert :: ∀a.a → List a → List a
insert x list = case list of
  Nil             → Cons_a x Nil_a
  Cons first rest → choose_a
    (Cons_a x list)
    (Cons_a first (insert_a x rest))
permute :: ∀a.List a → List a
permute list = case list of
  Nil             → Nil_a
  Cons first rest → insert_a first (permute_a rest)
append :: ∀a.List a → List a → List a
append xs ys = case xs of
  Nil             → ys
  Cons first rest → Cons_a first (append_a rest ys)
last :: ∀a.Data a ⇒ List a → a
last list =
  let e :: a free in
  let init :: List a free in
  case $append_a$ $init$ ($\mathsf{Cons}_a$ $e$ $\mathsf{Nil}_a$) $\equiv$ $list$ of
    True $\to$ $e$
    False $\to$ $\mathbf{failed}_a$
```

The syntax, in particular the type signatures, will be explained in detail later. Notable differences to Curry include explicit type variable instantiation for polymorphic functions, mandatory type signatures, and pattern matching exclusively with **case** expressions.

1.4 SaLT

SaLT makes the nondeterminism of CuMin programs explicit in the type. Nondeterministic expressions and functions have set types, written $\{\cdot\}$. Unlike CuMin, it does not have the **let** . . . **free** construct. Instead, there is a special keyword **unknown**. For a certain class of types, including typical ADTs, it represents the set of all values of that type. For example, $\mathbf{unknown}_{\mathsf{Bool}}$ represents the set $\{\mathsf{False}, \mathsf{True}\}$ and corresponds to the CuMin expression **let** $x :: \mathsf{Bool}$ **free in** x. There is only one combinator for sets, namely $\ggg\!=$, pronounced "bind". It represents an indexed union. More precisely, $e \ggg\!= f$ where $e :: \{\tau\}$ and $f :: \tau \to \{\tau'\}$, for some types τ, τ', represents the set $\bigcup_{x\in e}(f\ x)$. This can be used to the same effect as **let** . . . **free** bindings in CuMin, as the following translation of the *choose* function demonstrates.

```
$choose :: a \to a \to \{a\}$
$choose = \lambda x :: a \to \lambda y :: a \to \mathbf{unknown}_{\mathsf{Bool}}\ \ggg\!=$
  $\lambda choice :: \mathsf{Bool} \to$ case $choice$ of
    False $\to \{x\}$
    True $\to \{y\}$
```

The lambda abstraction after $\ggg\!=$ replaces False by x and True by y in the set $\{\mathsf{False}, \mathsf{True}\}$, thus forming the set $\{x, y\}$. This means that, while the CuMin function *choose* nondeterministically returns either of its arguments, the SaLT version returns all these results as a set. The CuMin example functions from above can be expressed in SaLT as follows.

```
$append :: \forall a.\mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a$
$append = \lambda xs :: \mathsf{List}\ a \to \lambda ys :: \mathsf{List}\ a \to$ case $xs$ of
  Nil              $\to ys$
  Cons $first\ rest \to \mathsf{Cons}_a\ first\ (append_a\ rest\ ys)$

$last :: \forall a.\mathsf{Data}\ a \Rightarrow \mathsf{List}\ a \to \{a\}$
$last = \lambda list :: \mathsf{List}\ a \to$
  $\mathbf{unknown}_a\ \ggg\!= \lambda e \to$
  $\mathbf{unknown}_{\mathsf{List}\ a}\ \ggg\!= \lambda init \to$
  case $append_a\ init\ (\mathsf{Cons}_a\ e\ \mathsf{Nil}_a) \equiv list$ of
    True $\to \{e\}$
    False $\to \mathbf{failed}_{\{a\}}$
```

Since the type signature of *append* does not contain any set types, one can immediately know for sure that the function is deterministic. On the other hand, *last* uses **unknown** internally, so it must have set type. The code also demonstrates some other differences from CuMin, such as mandatory lambda abstractions instead of function argument notation.

1.5 Overview and Goals of the Thesis

Why do we concern ourselves with all these languages? As said before, Haskell was discussed because it is the basis for all the other languages that this thesis deals with. Curry is included in the introduction since it is one of the most well-known functional-logic languages.

In the following, however, the thesis will be dealing with CuMin and SaLT. There are technical reasons for restricting ourselves to these two languages instead of Curry, which are given in [11]. CuMin is better suited for studying the semantics and SaLT is a means of better understanding nondeterminism in CuMin. The two languages did not have an implementation before, so the first goal is to implement a semantics. There are two different kinds of semantics, namely a *denotational* and an *operational* semantics. The former gives a mathematical meaning to programs, the latter describes a program's execution more directly. As part of this thesis, I implemented an operational semantics (chapter 3) for a variant of CuMin that is more general than in [11]. For instance, it supports general algebraic data types. At the same time, Fabian Thorand developed an implementation of a denotational semantics for both CuMin and SaLT, as part of his bachelor thesis. This way, we were able to test whether the operational and denotational semantics are consistent with each other. Additionally, the authors of [11] claim that the semantics of CuMin is compatible with real Curry implementations, i.e. Curry programs expressed in CuMin should exhibit an equivalent behavior.

Moreover, CuMin programs can be translated to SaLT. I implemented this translation, adapted from [11], as well as certain ways to simplify the generated SaLT code (chapter 4). It creates an interesting connection between the two languages since Fabian Thorand implemented semantics for both of them. However, the main goal of the translation and simplifications is to better understand the nondeterminism in CuMin since it is made more explicit in SaLT. How one can analyze the nondeterministic behavior of programs this way, is explained by way of examples in chapter 5. But first, the languages CuMin and SaLT have to be properly specified, which is the purpose of the next chapter.

Chapter 2

Preliminaries

In this chapter, I will give a more precise description of CuMin and SaLT, as well as some remarks on the implementation of parsers and type checkers. This part of the implementation was done together with Fabian Thorand, who wrote his bachelor thesis, which is concerned with other aspects of CuMin and SaLT, over the same period of time.

2.1 Notation and Conventions

In the following, a sequence of objects $z_1 \dots z_n$ will be denoted $\overline{z_n}$. An empty sequence, i.e. $n = 0$, is allowed. If the index is unclear, it is written $(\overline{z_n})_n$, or even $(\overline{z_n})_{n \in S}$ where S denotes the range of n. The notation $e\,[y/x]$ means replacing every unbound occurrence of x in e with y. By convention, α, β denote type variables; ρ, τ denote types; x, y denote variables; m, n denote natural numbers; p denotes a pattern in pattern matches; e denotes an expression; f denotes a function; C denotes a constructor; A denotes an algebraic data type; and Γ denotes a context, unless otherwise stated.

When specifying the well-formedness, evaluation or other *judgments* about programs, types or expressions, the following notation is widely used.

$$\frac{\text{Assumption 1} \qquad \text{Assumption 2} \qquad \dots}{\text{Consequence}}$$

This means that the consequence is a valid judgment if all the assumptions can be shown to be valid judgments. Oftentimes, these judgments will only make sense in a certain *context*, denoted Γ. The judgment is then written with the turnstile symbol: $\Gamma \vdash$ judgment. More details on the presentation of type systems can be found in [13].

2.2 Types in CuMin and SaLT

Types can either be a type variable or a type constructor applied to zero or more types, whose number must match the arity of the type constructor. The following list contains all valid type constructors.

- The name of an algebraic data type. Its arity is the number of type parameters in the definition.
- The function type constructor $\to$, with two arguments. It associates to the right.

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

$$\frac{\Gamma, \alpha \text{ context}}{\Gamma, \alpha, \alpha \in \mathsf{Data} \text{ context}} \qquad \frac{\Gamma \text{ context} \qquad I \subset \mathbb{N}}{\Gamma, \mathsf{A} \in \mathsf{Data}^I \text{ context}} \text{ for an ADT } \mathsf{A}$$

Figure 2.1: Context formation rules

$$\Gamma, \alpha \vdash \alpha \in \mathsf{Type} \qquad \Gamma \vdash \mathsf{Nat} \in \mathsf{Type} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Type} \qquad \Gamma \vdash \tau' \in \mathsf{Type}}{\Gamma \vdash \tau \to \tau' \in \mathsf{Type}}$$

$$\frac{\overline{\Gamma \vdash \tau_l \in \mathsf{Type}}}{\Gamma \vdash \mathsf{A}\ \overline{\tau_l} \in \mathsf{Type}} \text{ for an ADT } \mathsf{A} \text{ with } l \text{ parameters} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Type}}{\Gamma \vdash \{\tau\} \in \mathsf{Type}} \text{ in SaLT}$$

Figure 2.2: Type formation rules

- Nat, a primitive type for natural numbers.
- Only in SaLT: $\{\cdot\}$ to create set types, with one argument.

To make this exact, one first needs to describe what valid type contexts look like (fig. 2.1). These are the allowed contexts in typing judgments. The above description of types can then be formalized and is shown in fig. 2.2.

Typing judgments are always made with respect to a given program P. After all, they can depend on the ADTs defined in P and, as we will see later, on the function definitions in P, too. Because of that, we would theoretically have to index all typing judgments by that program. However, we omit the index for the sake of readability.

While there are only three built-in algebraic data types in [11], namely lists, pairs and booleans, we considered this to be too limiting and decided to augment the languages with general ADTs. Such an algebraic data type is defined like it is in Haskell. It has a name A, is parameterized by zero or more type variables $\overline{\alpha_l}$, has one or more constructors $\overline{\mathsf{C}_m}$, each of which is specified by its name and its argument types $\overline{\tau_{mn_m}}$.

$$\textbf{data}\ \mathsf{A}\ \alpha_1 \ldots \alpha_l = \mathsf{C}_1\ \tau_{11} \ldots \tau_{1n_1} \mid \cdots \mid \mathsf{C}_m\ \tau_{m1} \ldots \tau_{mn_m}$$

According to the above conventions, I will often abbreviate it like this:

$$\textbf{data}\ \mathsf{A}\ \overline{\alpha_l} = \overline{\mathsf{C}_m\ \overline{\tau_{mn_m}}}$$

The only type variables allowed in the types $\overline{\tau_{mn_m}}$ are $\overline{\alpha_l}$. Higher-kinded type variables are not supported, which means that a type variable cannot be applied to other types. As a consequence, the following data type is invalid in CuMin and SaLT although it is permitted in Haskell.

$$\textbf{data}\ \mathsf{D}\ f\ a = \mathsf{D}\ (f\ a)$$

Logic variables in CuMin and the **unknown** primitive in SaLT cannot have an arbitrary type; only so called Data types are permitted. This is because values of logic variables have to be

$$\Gamma, \alpha, \alpha \in \mathsf{Data} \vdash \alpha \in \mathsf{Data} \qquad \Gamma, \alpha, \mathsf{A} \in \mathsf{Data}^I \vdash \mathsf{A} \in \mathsf{Data}^I$$

$$\Gamma \vdash \mathsf{Nat} \in \mathsf{Data} \qquad \frac{\Gamma \vdash \mathsf{A} \in \mathsf{Data}^I \qquad (\overline{\Gamma \vdash \tau_i \in \mathsf{Data}})_{i \in I}}{\Gamma \vdash \mathsf{A}\ \overline{\tau_l} \in \mathsf{Data}}$$

$$\frac{\left(\overline{\Gamma, \mathsf{A} \in \mathsf{Data}^I, \overline{\alpha_l}, (\overline{\alpha_l \in \mathsf{Data}})_{l \in I} \ \vdash \tau_{mn_m} \in \mathsf{Data}}\right)_{mn_m}}{\Gamma \vdash \mathsf{A} \in \mathsf{Data}^I} \quad \text{where } \mathbf{data}\ \mathsf{A}\ \overline{\alpha_l} = \overline{\mathsf{C}_m\ \overline{\tau_{mn_m}}}$$

Figure 2.3: Rules for Data types

enumerable. As a counterexample, values of function types do not have a natural structure for enumeration. Most algebraic data types, however, are structurally enumerable, for instance Bool, Nat, $\mathsf{List}\ \tau$ or $\mathsf{Tree}\ \tau$ for any enumerable τ. They can be enumerated because all constructors can be listed and their arguments are enumerable.

In order to formalize this notion, we introduce another judgment $\mathsf{A} \in \mathsf{Data}^I$ for an ADT A with l type parameters where I has to be a subset of $\{1, \ldots, l\}$. It states that $\mathsf{A}\ \overline{\tau_l}$ is a Data type if, for each $i \in I$, τ_i is a Data type. Essentially, an ADT is a Data type if the types of the arguments of all constructors are Data types. The rules in fig. 2.3 capture this notion. Note that this is another deviation from [11], which only specifies simple rules for the three built-in ADTs. As we allow general algebraic data types, these more intricate rules are necessary.

Let us take a look at some examples. As all constructors of Bool are nullary, the deduction of $\mathsf{Bool} \in \mathsf{Data}$ is quite simple.

$$\frac{\Gamma \vdash \mathsf{Bool} \in \mathsf{Data}^{\emptyset}}{\Gamma \vdash \mathsf{Bool} \in \mathsf{Data}}$$

One can also deduce $\mathsf{List} \in \mathsf{Data}^{\{1\}}$, using the abbreviation $\Gamma' := \Gamma, \mathsf{List} \in \mathsf{Data}^{\{1\}}, a, a \in \mathsf{Data}$.

$$\frac{\Gamma' \vdash a \in \mathsf{Data} \qquad \dfrac{\Gamma' \vdash \mathsf{List} \in \mathsf{Data}^{\{1\}} \qquad \Gamma' \vdash a \in \mathsf{Data}}{\Gamma' \vdash \mathsf{List}\ a \in \mathsf{Data}}}{\Gamma \vdash \mathsf{List} \in \mathsf{Data}^{\{1\}}}$$

Putting these two results together, we can show that $\mathsf{List\ Bool} \in \mathsf{Data}$ holds, too.

$$\frac{\Gamma \vdash \mathsf{List} \in \mathsf{Data}^{\{1\}} \qquad \Gamma \vdash \mathsf{Bool} \in \mathsf{Data}}{\Gamma \vdash \mathsf{List\ Bool} \in \mathsf{Data}}$$

If a type variable is used in no constructor, it does not have to be a Data type. As an example, consider the phantom type

$$\mathbf{data}\ \mathsf{Phantom}\ a = \mathsf{Phantom}$$

where the type parameter a only occurs on the left-hand side. $\mathsf{Phantom}\ \tau$ is a Data type for any τ since its only value is $\mathsf{Phantom}$. So, even for function types like $\mathsf{Nat} \to \mathsf{Nat}$, we have $\mathsf{Phantom}\ (\mathsf{Nat} \to \mathsf{Nat}) \in \mathsf{Data}$.

$$\frac{\Gamma \vdash \mathsf{Phantom} \in \mathsf{Data}^{\emptyset}}{\Gamma \vdash \mathsf{Phantom}\ \tau \in \mathsf{Data}}$$

$$
\begin{aligned}
e ::= {} & x \mid n \mid f_{\overline{\tau_m}} \mid \mathsf{C}_{\overline{\tau_m}} \\
& \mid e_1\ e_2 \mid e_1 + e_2 \mid e_1 \equiv e_2 \mid \textbf{failed}_\tau \\
& \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{let}\ x :: \tau\ \textbf{free in}\ e \\
& \mid \textbf{case}\ e\ \textbf{of}\ \{\overline{\mathsf{C}_i\ \overline{x_{i_j}} \to e_i;}\} \\
& \mid \textbf{case}\ e\ \textbf{of}\ \{\overline{\mathsf{C}_i\ \overline{x_{i_j}} \to e_i;}\ x \to e'\} \\
n ::= {} & 0 \mid 1 \mid \ldots
\end{aligned}
$$

Figure 2.4: Syntax of CuMin expressions

2.3 CuMin Syntax and Typing

2.3.1 Syntax

CuMin programs consist of algebraic data type declarations and function definitions. A function f is defined by giving its *type signature*, a list of variables that constitute the arguments, and the expression it computes, depending on the arguments.

$$
\begin{aligned}
& f :: \forall\overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau_1 \to \cdots \to \tau_n \to \tau \\
& f\ x_1 \ldots x_n = e
\end{aligned}
$$

The type signature consists of *type variables* to allow polymorphism, a *type class context* which specifies that certain type variables have to be instantiated to Data types, and finally the actual function type. One does not need to write the empty context $() \Rightarrow$ if the function has no Data constraints, and if there are no type variables, the $\forall.$ may be dropped, as well.

The shape of CuMin expressions is shown in fig. 2.4. As might be expected, the syntax includes variables and literals for natural numbers. Polymorphic functions and constructors have to be given type instantiations at the call site, denoted by subscripts. In principle, these could be inferred automatically, but this complicates type checking. For the sake of simplicity, these annotations are mandatory. Function application is written by juxtaposition, it associates to the left and has the highest binding precedence. The supported primitive operations are addition for natural numbers and equality checks for Data types, in particular natural numbers. Comparison for equality requires a certain structure on the values, thus the Data requirement. After all, checking whether two functions are equal is undecidable in general. As usual, the operator $+$ binds more tightly than $\equiv$. The former associates to the left and the latter is not associative. Parentheses can be used for structuring expressions. The keyword **failed** signifies that the computation does not yield a result. It can be used to “cut off” unwanted computation branches. Let bindings allow using the result of a computation more than once in an expression by binding it to a variable x. Recursive let bindings are not allowed, i.e. x must not occur in e_1. The construct **let**...**free** introduces logic variables; it is the only logic feature in the language. Case expressions examine the value of an expression, called the *scrutinee*. Its value is matched with the constructor patterns $\mathsf{C}_i\ x_{i_j}$. The constructors C_i that are matched on must be pairwise different and there must be at least one constructor pattern. There may or may not be a catch-all variable pattern x at the end. It only matches if none of the constructors before did.

We modified the CuMin syntax from [11] in the following points. It was already mentioned that we allow general ADTs. As a consequence, the syntax for constructors and case expressions has to be generalized as well. Additionally, we lift the requirement that constructors have to be fully applied. Moreover, the type class context in function signatures is written differently. In the original paper, the type variables with a Data requirement where indicated by tagging

Mathematical notation	Plain text
$\forall a.$	`forall a.`
$\rightarrow$	`->`
$\Rightarrow$	`=>`
f_t	`f<:t:>`
$\equiv$	`==`

Figure 2.5: Plain text representation of CuMin

the quantifier $\forall$ with a $*$. Another discrepancy are case expressions. While the original syntax expects one pattern for each constructor, we permit a catch-all variable pattern at the end and do not require every constructor to be matched. Furthermore, the primitive $\textbf{anything}_\tau$ from [11] (corresponding to **let** $x :: \tau$ **free in** x) is removed in favor of the **let**... **free** construct. Finally, the keyword **failure** is renamed to **failed**.

Besides the mathematical notation, there is also a plain-text representation of CuMin code. The straightforward correspondence is shown in fig. 2.5. As in Haskell, code can also be written using indentation instead of braces and semicolons:

$$\begin{array}{l}\textbf{case } e \textbf{ of}\\ \quad \mathsf{C}_1 \cdots \rightarrow \ldots\\ \quad \mathsf{C}_2 \cdots \rightarrow \ldots\end{array}$$

There are some data types and functions that are so common and useful that we decided to put them in a so-called *prelude*, which is copied to the top of every program. It defines data types like lists and booleans, as well as functions that operate on them. The types of the definitions of the prelude are listed in fig. 2.6.

Finally, there is some syntactic sugar to make programs easier to read and write. List literals can be written in the natural way $[e_1, \ldots, e_n]_\tau$, which is desugared to the expression $\mathsf{Cons}_\tau\ e_1\ (\ldots(\mathsf{Cons}_\tau\ e_n\ \mathsf{Nil}_\tau\ )\ldots)$.

2.3.2 Typing

CuMin expressions have to be well-typed. For example, the term $\mathsf{True} + 1$ does not make sense, because the primitive operator $+$ only accepts natural numbers as arguments. The typing rules are given in fig. 2.7. They are based on [11] but take our modifications and generalizations of the language into account. Another deviation from the original paper, which is does not affect the syntax, can be seen, as well. While the original paper restricts equality checks to natural numbers, we allow all Data types. In order to type check functions, recall the shape of their definitions.

$$\begin{array}{l} f :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau\\ f\ x_1 \ldots x_n = e\end{array}$$

Such a CuMin function f is well-typed if the following judgment holds.

$$\overline{\alpha_m}, \overline{\alpha_{i_j} \in \mathsf{Data}}, \overline{x_n :: \tau_n} \vdash e :: \tau$$

A CuMin program is well-typed if each of its ADTs and functions is well-typed.

$\mathbf{data}\ \mathsf{Pair}\ a\ b = \mathsf{Pair}\ a\ b$
$\mathbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)$
$\mathbf{data}\ \mathsf{Maybe}\ a = \mathsf{Nothing} \mid \mathsf{Just}\ a$
$\mathbf{data}\ \mathsf{Either}\ a\ b = \mathsf{Left}\ a \mid \mathsf{Right}\ b$
$\mathbf{data}\ \mathsf{Bool} = \mathsf{False} \mid \mathsf{True}$

$\mathit{and} :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$
$\mathit{choose} :: \forall a.a \to a \to a$
$\mathit{const} :: \forall a\ b.a \to b \to a$
$\mathit{either} :: \forall a\ b\ c.(a \to c) \to (b \to c) \to \mathsf{Either}\ a\ b \to c$
$\mathit{filter} :: \forall a.(a \to \mathsf{Bool}) \to \mathsf{List}\ a \to \mathsf{List}\ a$
$\mathit{flip} :: \forall a\ b\ c.(a \to b \to c) \to b \to a \to c$
$\mathit{foldr} :: \forall a\ b.(a \to b \to b) \to b \to \mathsf{List}\ a \to b$
$\mathit{fst} :: \forall a\ b.\mathsf{Pair}\ a\ b \to a$
$\mathit{guard} :: \forall a.\mathsf{Bool} \to a \to a$
$\mathit{id} :: \forall a.a \to a$
$\mathit{length} :: \forall a.\mathsf{List}\ a \to \mathsf{Nat}$
$\mathit{map} :: \forall a\ b.(a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b$
$\mathit{maybe} :: \forall a\ b.b \to (a \to b) \to \mathsf{Maybe}\ a \to b$
$\mathit{not} :: \mathsf{Bool} \to \mathsf{Bool}$
$\mathit{or} :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$
$\mathit{snd} :: \forall a\ b.\mathsf{Pair}\ a\ b \to b$

Figure 2.6: The CuMin Prelude

2.3.3 Examples

As an example, consider the following program.

$\mathit{choose} :: \forall a.a \to a \to a$
$\mathit{choose}\ x\ y = \mathbf{let}\ c :: \mathsf{Bool}\ \mathbf{free\ in\ case}\ c\ \mathbf{of}$
$\quad \mathsf{True} \to x$
$\quad \mathsf{False} \to y$

$\mathit{map} :: \forall a\ b.(a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b$
$\mathit{map}\ f\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}$
$\quad \mathsf{Nil} \qquad\quad\ \to \mathsf{Nil}_b$
$\quad \mathsf{Cons}\ y\ \mathit{ys} \to \mathsf{Cons}_b\ (f\ y)\ (\mathit{map}_{a,b}\ f\ \mathit{ys})$

To prove that *choose* is well-typed, consider the following deduction. Let $\Gamma := a, x :: a, y :: a$.

$$\dfrac{\dfrac{\Gamma, c :: \mathsf{Bool} \vdash c :: \mathsf{Bool} \qquad \Gamma, c :: \mathsf{Bool} \vdash x :: a \qquad \Gamma, c :: \mathsf{Bool} \vdash y :: a}{\Gamma, c :: \mathsf{Bool} \vdash \mathbf{case}\ c\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to y\} :: a} \qquad \dfrac{\dots}{\Gamma \vdash \mathsf{Bool} \in \mathsf{Data}}}{\Gamma \vdash \mathbf{let}\ c :: \mathsf{Bool}\ \mathbf{free\ in\ case}\ c\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to y\} :: a}$$

The judgment $\Gamma \vdash \mathsf{Bool} \in \mathsf{Data}$ has been proven before. Similarly, one can derive that *map* is well-typed. Let $\Gamma := f :: a \to b, \mathit{xs} :: \mathsf{List}\ a$ and $\Gamma' := \Gamma, y :: a, \mathit{ys} :: \mathsf{List}\ a$.

$$\dfrac{\Gamma \vdash \mathit{xs} :: \mathsf{List}\ a \qquad \Gamma \vdash \mathsf{Nil}_b\ :: \mathsf{List}\ b \qquad \dfrac{\dfrac{\dots}{\Gamma' \vdash \mathsf{Cons}_b\ \dots :: \mathsf{List}\ b \to \mathsf{List}\ b} \qquad \dfrac{\dots}{\Gamma' \vdash \mathit{map}_{a,b}\ \dots :: \mathsf{List}\ b}}{\Gamma, y :: a, \mathit{ys} :: \mathsf{List}\ a \vdash \mathsf{Cons}_b\ (f\ y)\ (\mathit{map}_{a,b}\ f\ \mathit{ys}) :: \mathsf{List}\ b}}{\Gamma \vdash \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{\mathsf{Nil} \to \mathsf{Nil}_b\ ; \mathsf{Cons}\ y\ \mathit{ys} \to \mathsf{Cons}_b\ (f\ y)\ (\mathit{map}_{a,b}\ f\ \mathit{ys})\} :: \mathsf{List}\ b}$$

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash n :: \mathsf{Nat} \qquad \Gamma \vdash \mathbf{failed}_\tau :: \tau \qquad \frac{\Gamma \vdash e_1 :: \tau' \to \tau \qquad \Gamma \vdash e_2 :: \tau'}{\Gamma \vdash e_1\ e_2 :: \tau}$$

$$\frac{\overline{\Gamma \vdash \tau_{i_j} \in \mathsf{Data}}}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau\ [\tau_m/\alpha_m]} \quad \text{where } f :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau$$

$$\Gamma \vdash \mathsf{C}_{\overline{\rho_m}} :: (\tau_1 \to \dots \tau_n \to \mathsf{A}\ \overline{\alpha_m})\ [\overline{\rho_m/\alpha_m}] \text{ for every } \mathbf{data}\ \mathsf{A}\ \overline{\alpha_m} = \dots \mid \mathsf{C}\ \tau_1 \dots \tau_n \mid \dots$$

$$\frac{\Gamma \vdash e_1 :: \tau' \qquad \Gamma, x :: \tau' \vdash e_2 :: \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :: \tau} \quad \frac{\Gamma, x :: \tau' \vdash e :: \tau \qquad \Gamma \vdash \tau' \in \mathsf{Data}}{\Gamma \vdash \mathbf{let}\ x :: \tau'\ \mathbf{free\ in}\ e :: \tau}$$

$$\frac{\Gamma \vdash e_1 :: \mathsf{Nat} \qquad \Gamma \vdash e_2 :: \mathsf{Nat}}{\Gamma \vdash e_1 + e_2 :: \mathsf{Nat}} \quad \frac{\Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \tau \qquad \Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash e_1 \equiv e_2 :: \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e :: \mathsf{A}\ \overline{\rho_m} \qquad \overline{\Gamma, \overline{x_{i_j} :: \tau_{i_j}\ [\overline{\rho_m/\alpha_m}]} \vdash e_i :: \tau}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{C}_i\ \overline{x_{i_j}} \to e_i;}\} :: \tau} \quad \text{for every } \mathbf{data}\ \mathsf{A}\ \overline{\alpha_m} = \overline{\mathsf{C}_k\ \overline{\tau_{k_j}}}$$

$$\frac{\dots\text{(as above)} \qquad \Gamma, x :: \mathsf{A}\ \overline{\rho_m} \vdash e' :: \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{C}_i\ \overline{x_{i_j}} \to e_i;}\ x \to e';\} :: \tau} \quad \text{for every } \mathbf{data}\ \mathsf{A}\ \overline{\alpha_m} = \overline{\mathsf{C}_k\ \overline{\tau_{k_j}}}$$

Figure 2.7: Typing rules for CuMin expressions

The missing subderivations look like this.

$$\frac{\Gamma' \vdash \mathsf{Cons}_b :: b \to \mathsf{List}\ b \to \mathsf{List}\ b \qquad \dfrac{\Gamma' \vdash f :: a \to b \qquad \Gamma' \vdash y :: a}{\Gamma' \vdash (f\ y) :: b}}{\Gamma' \vdash \mathsf{Cons}_b\ (f\ y) :: \mathsf{List}\ b \to \mathsf{List}\ b}$$

$$\frac{\dfrac{\Gamma' \vdash map_{a,b} :: (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b \qquad \Gamma' \vdash f :: \mathsf{List}\ a}{\Gamma' \vdash map_{a,b}\ f :: \mathsf{List}\ a \to \mathsf{List}\ b} \qquad \Gamma' \vdash ys :: \mathsf{List}\ a}{\Gamma' \vdash map_{a,b}\ f\ ys :: \mathsf{List}\ b}$$

2.4 SaLT Syntax and Typing

The syntax of SaLT is quite similar to CuMin (fig. 2.8). However, it replaces the **let**...**free** construct with the keyword $\mathbf{unknown}_\tau$, which represents the set of values of the Data type τ. As mentioned in the introduction, other primitives for sets are $\{\cdot\}$ for creating singleton sets and $\ggg\!=$ for indexed unions. This operator binds least tightly and associates to the left. As SaLT has explicit lambda abstractions, there is no need for function arguments. Hence, function definitions are simpler than in CuMin.

$$\begin{aligned}&f :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau\\ &f = e\end{aligned}$$

The SaLT syntax deviates slightly from [11], as well. Except for the **let**...**free** construct, the modifications of CuMin also apply to SaLT. Additionally, the **anything** primitive from the

$$
\begin{aligned}
e ::=\ & x \mid n \mid f_{\overline{\tau_m}} \mid \mathsf{C}_{\overline{\tau_m}} \mid \lambda x :: \tau \to e \\
& \mid e_1\ e_2 \mid e_1 + e_2 \mid e_1 \equiv e_2 \mid \mathbf{failed}_\tau \\
& \mid \mathbf{unknown}_\tau \mid \{e\} \mid e_1 \ggg\!= e_2 \\
& \mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{C}_i\ \overline{x_{i_j}} \to e_i;}\} \\
& \mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{C}_i\ \overline{x_{i_j}} \to e_i;}\ x \to e'\} \\
n ::=\ & 0 \mid 1 \mid \ldots
\end{aligned}
$$

Figure 2.8: Syntax of SaLT expressions

Mathematical notation	plain text
$\{t\}$ in types	`Set t`
$\{e\}$ in expressions	`{e}`
λ	`\`
$\ggg\!=$	`>>=`

Figure 2.9: Plain text representation of SaLT syntax

original paper is renamed to **unknown**. Furthermore, the syntax for indexed unions is different. The original paper writes $e_1 \ni x \bigcup e_2$ for what we denote by $e_1 \ggg\!= \lambda x :: \tau \to e_2$. Our syntax is more liberal because we do not restrict ourselves to lambda abstractions on the right-hand side.

SaLT has the same plain text representation as CuMin. The additional symbols are written as shown in fig. 2.9. Since `Set` is a reserved type constructor, it cannot be the name of an ADT.

As for CuMin, we created a SaLT prelude with useful definitions. It is mainly a manual translation of the CuMin prelude. There are only two major differences.

$$
\begin{aligned}
&\mathit{choose} :: \forall a.\{a\} \to \{a\} \to \{a\} \\
&\mathit{sMap} :: \forall a\ b.(a \to b) \to \{a\} \to \{b\}
\end{aligned}
$$

The SaLT version of *choose* operates on sets by forming their union. There is also a new function *sMap* which acts on sets like *map* acts on lists. It yields a set where the given function has been applied to every element of the original one.

There is also an alternative prelude that is generated by the translation method described in chapter 4. It behaves the same but due to the nature of the translation, its functions contain more sets than necessary, for example, *id* is translated to $\mathit{id} :: \forall a.\{a \to \{a\}\}$.

The SaLT typing rules are similar to those of CuMin. The ones for let bindings are now unnecessary. Instead, there are rules for lambda abstractions, and most importantly, for handling sets (fig. 2.10). A function in the shape given above is well-typed if the following judgment is correct.

$$\overline{\alpha_m}, \overline{\alpha_{i_j} \in \mathsf{Data}} \vdash e :: \tau$$

Having specified the SaLT syntax and typing rules, let us take a look at some examples. It is instructive to translate the above CuMin programs to SaLT.

$$
\begin{aligned}
&\mathit{choose} :: \forall a.\{a\} \to \{a\} \to \{a\} \\
&\mathit{choose} = \lambda x :: \{a\} \to \lambda y :: \{a\} \to \mathbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda c :: \mathsf{Bool} \to \\
&\quad \mathbf{case}\ c\ \mathbf{of} \\
&\qquad \mathsf{True} \to x \\
&\qquad \mathsf{False} \to y
\end{aligned}
$$

$$\dfrac{\Gamma, x :: \tau' \vdash e :: \tau}{\Gamma \vdash (\lambda x :: \tau' \to e) :: \tau' \to \tau}$$

$$\dfrac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash \mathbf{unknown}_\tau \; :: \{\tau\}} \qquad \dfrac{\Gamma \vdash e :: \tau}{\Gamma \vdash \{e\} :: \{\tau\}} \qquad \dfrac{\Gamma \vdash e_1 :: \{\tau\} \qquad \Gamma \vdash e_2 :: \tau \to \{\tau'\}}{\Gamma \vdash e_1 \mathbin{>\!\!>\!\!=} e_2 :: \{\tau'\}}$$

Figure 2.10: SaLT-specific typing rules

$$\begin{aligned}
&\mathit{map} :: \forall a\ b.(a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b\\
&\mathit{map} = \lambda f :: (a \to b) \to \lambda \mathit{xs} :: \mathsf{List}\ a \to \mathbf{case}\ \mathit{xs}\ \mathbf{of}\\
&\quad \mathsf{Nil} \qquad\qquad \to \mathsf{Nil}_b\\
&\quad \mathsf{Cons}\ y\ \mathit{ys} \to \mathsf{Cons}_b\ (f\ y)\ (\mathit{map}_{a,b}\ f\ \mathit{ys})
\end{aligned}$$

Proving that *choose* is well-typed works similarly as above. Let $\Gamma := a, x :: \{a\}, y :: \{a\}$ and $\Gamma' := \Gamma, c :: \mathsf{Bool}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma \vdash \mathsf{Bool} \in \mathsf{Data}}}{\Gamma \vdash \mathbf{unknown}_{\mathsf{Bool}} \; :: \{\mathsf{Bool}\}} \qquad \dfrac{\dfrac{\Gamma' \vdash c :: \mathsf{Bool} \qquad \Gamma' \vdash x :: \{a\} \qquad \Gamma' \vdash y :: \{a\}}{\Gamma' \vdash \mathbf{case}\ c\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to y\} :: \{a\}}}{\Gamma \vdash \lambda c :: \mathsf{Bool} \to \mathbf{case}\ c\ \mathbf{of}\ \{\dots\} :: \mathsf{Bool} \to \{a\}}}{\Gamma \vdash \mathbf{unknown}_{\mathsf{Bool}} \; \mathbin{>\!\!>\!\!=} \lambda c :: \mathsf{Bool} \to \mathbf{case}\ c\ \mathbf{of}\ \{\dots\} :: \{a\}}}{a, x :: \{a\} \vdash \lambda y :: \{a\} \to \mathbf{unknown}_{\mathsf{Bool}} \; \mathbin{>\!\!>\!\!=} \lambda c :: \mathsf{Bool} \to \mathbf{case}\ c\ \mathbf{of}\ \{\dots\} :: \{a\} \to \{a\}}}{a \vdash \lambda x :: \{a\} \to \lambda y :: \{a\} \to \mathbf{unknown}_{\mathsf{Bool}} \; \mathbin{>\!\!>\!\!=} \lambda c :: \mathsf{Bool} \to \mathbf{case}\ c\ \mathbf{of}\ \{\dots\} :: \{a\} \to \{a\} \to \{a\}}$$

The proof of well-typedness of *map* very similar to the one in CuMin, so it can be safely left out. Instead let us understand the function *sMap*.

$$\begin{aligned}
&\mathit{sMap} :: \forall a\ b.(a \to b) \to \{a\} \to \{b\}\\
&\mathit{sMap} = \lambda f :: (a \to b) \to \lambda \mathit{xs} :: \{a\} \to \mathit{xs} \mathbin{>\!\!>\!\!=} \lambda x :: a \to \{f\ x\}
\end{aligned}$$

This function applies f to every element of xs. As the only combinator for sets is the indexed union with $\mathbin{>\!\!>\!\!=}$, *sMap* constructs the singleton set $\{f\ x\}$ for every element x of xs and forms the union of these sets, yielding the set $\{(f\ x) | x \in \mathit{xs}\}$, as desired. The relevant expression $\mathit{xs} \mathbin{>\!\!>\!\!=} \lambda x :: a \to \{f\ x\}$ is perhaps more suggestive when written in the notation of [11], namely as $\mathit{xs} \ni x \bigcup \{f\ x\}$. To check type correctness, let $\Gamma := a, b, f :: a \to b, \mathit{xs} :: \{a\}$.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathit{xs} :: \{a\} \qquad \dfrac{\dfrac{\dfrac{\Gamma, x :: a \vdash f :: a \to b \qquad \Gamma, x :: a \vdash x :: a}{\Gamma, x :: a \vdash f\ x :: b}}{\Gamma, x :: a \vdash \{f\ x\} :: \{b\}}}{\Gamma \vdash \lambda x :: a \to \{f\ x\} :: a \to \{b\}}}{\Gamma \vdash \mathit{xs} \mathbin{>\!\!>\!\!=} \lambda x :: a \to \{f\ x\} :: \{b\}}}{a, b, f :: (a \to b) \vdash \lambda \mathit{xs} :: \{a\} \to \mathit{xs} \mathbin{>\!\!>\!\!=} \lambda x :: a \to \{f\ x\} :: \{a\} \to \{b\}}}{a, b \vdash \lambda f :: (a \to b) \to \lambda \mathit{xs} :: \{a\} \to \mathit{xs} \mathbin{>\!\!>\!\!=} \lambda x :: a \to \{f\ x\} :: (a \to b) \to \{a\} \to \{b\}}$$

2.5 Implementation

Before implementing a semantics or translation method involving the two languages, some groundwork had to be done. CuMin and SaLT programs have to be parsed into an *abstract* *syntax tree (AST)*. A type checker for these ASTs is needed and a pretty-printer will be useful as well. This functionality was implemented together with Fabian Thorand, whose bachelor thesis is also concerned with CuMin and SaLT.

The functionality is split into three packages: `funlogic-common`, `language-cumin` and `language-salt`. The first one contains common functionality for both CuMin and SaLT, like the representation of types. The other two packages deal with the two languages specifically, each one providing a parser, pretty-printer and type checker.

All implementations in this thesis were developed in Haskell using the standard tools, namely the *Glasgow Haskell Compiler* (GHC)[1], version 7.8.3, and the *cabal* build system[2], version 1.18. The dependencies were installed from *Hackage*.[3] Detailed Installation instructions can be found on the attached CD.

2.5.1 Abstract Syntax Tree

The objects in CuMin and SaLT are straightforwardly modeled as algebraic data types in Haskell. For instance, a type is represented like this.

$$\textbf{data}\ \mathsf{Type} = \mathsf{TVar}\ \mathsf{TVName} \mid \mathsf{TCon}\ \mathsf{TyConName}\ [\mathsf{Type}]$$

A type is either a type variable or a type constructor applied to a list of types. Hence, TCon "->" [TVar "a", TCon "Set" [TCon "Bool" []]] is the representation of the SaLT type $a \to \{\mathsf{Bool}\}$. The representation of expressions is similarly given by introducing one constructor for each kind of expression from the previous sections.

One notorious problem in writing compilers is the representation of bound variables. In our implementation, they are simply represented by their names. Generally, this can cause a number of problems with substitution because free variables may be captured: Consider the two lambda terms $\lambda x \to y$ and $\lambda y \to z$. Blindly substituting the former term for z yields $\lambda y \to \lambda x \to y$, which is incorrect as the variable y is not free anymore. For the correct solution $\lambda y' \to \lambda x \to y$, variables have to be renamed. There are more elaborate representations for capture avoidance, like the one I explain in section 4.3.1, but we opted against that complexity in the common packages since the kind of substitution needed for type checking, namely instantiating type variables on function invocations, can never lead to unwanted capturing. That is because type variable bindings are always on the top level, introduced by $\forall$, and cannot be nested.

2.5.2 Parser

Programs are given in the plain text format specified above. This textual representation needs to be parsed and converted to an AST. Instead of writing a parser by hand, we took the usual approach in the Haskell community and used a parser combinator library [8]. The most well-known one is `parsec` but we chose `trifecta`[4] by Edward Kmett because it has more readable and (subjectively) better error messages. As we wanted indentation-sensitive parsing, we used the library `indentation` [1], which builds on top of `trifecta`. Parser combinators are a very readable and concise way of defining parsers. For instance, the parser for lambda abstractions `\x :: t -> e` in SaLT looks like this. (slightly modified for clarity)

```
lambdaE  =  ELam
            <$> (symbol "\\" *> varIdent)
```

[1] https://www.haskell.org/ghc/
[2] https://www.haskell.org/cabal/
[3] The Haskell package archive, http://hackage.haskell.org/
[4] http://hackage.haskell.org/package/trifecta

```
    <*> (symbol "::" *> complexType)
    <*> (symbol "->" *> expression)
```

The combinator $*\!>$ combines two parsers by running the first one and then the second one, returning the result of the second one. In this way, *lambdaE* first parses the bound variable, then its type, and then the expression. ELam constructs an expression AST from this data. The combinators <$> and <*> "lift" this operation to an operation on parsers, such that *lambdaE* is itself a parser that returns an expression. Parsers for other kinds of expressions look similar.

After having collected all top-level and data type definitions, they have to be checked for duplicates. Giving different functions the same name is obviously ambiguous and such programs have to be rejected. Similarly, ADT names must be unique among each other and the same applies to constructor names.

2.5.3 Pretty-printer

Pretty-printing is in a way the opposite of parsing: It means transforming an AST into human-readable code. As for parsing, we use a combinator library for pretty-printing, called `ansi-wl-pprint`[5]. It is based on [15] but with some extensions. This particular library allows colored output, which makes syntax highlighting in the terminal possible. The pretty-printer is aware of operator precedence, so it only uses parentheses where necessary. It is used for automatically generated programs, debugging and in the translation program (see chapter 4).

2.5.4 Type Checker

The type checker is essentially a direct implementation of the typing rules. It runs as a monadic computation [16] to keep track of the type variables and bound variables that are in scope.[6] The syntax tree can simply be checked from the bottom up, composing the types of smaller expressions to larger ones, according to the typing rules. No complicated inference is needed since the expressions have all the necessary type annotations to determine their type locally. (In contrast, a Haskell type inference would have to take global constraints into account.) As soon as an inconsistency is found, a type error is reported.

The only thing that deserves a special remark is the inference of Data types. The inference of the $\mathsf{A} \in \mathsf{Data}^I$ judgments is done once, at the beginning of type checking, and the result is stored. Closely following the typing rules to determine the index set I of type variables that need to be Data types would be inefficient because we would have to try all possible sets I. Instead, the following fixpoint iteration is used. We keep track of which ADTs are Data types and if so, which type variables have to be Data types for the ADT to be one. Let I_{A} be this set of constraints for an ADT A. It corresponds to the judgment $\mathsf{A} \in \mathsf{Data}^I$. In the beginning, every ADT A is assumed to be a Data type without any constraints, i.e. $I = \emptyset$. In each iteration, for each ADT A, and each constructor C, type variables in the argument type of C are added to this set. For argument types that are type constructors D applied to other types, we first check that the D is a Data type, according to the current knowledge. If not, A itself cannot be a Data type. Otherwise, we require the types to which the D is applied to be Data types, but only those with an index in the constraint set I_{D}.

In this way, more necessary Data constraints are accumulated until a fixed point is reached. Then, the constraints are also sufficient. Such a fixed point is reached since the syntactic nesting level of data types is bounded. When the maximum nesting level has been explored, there are

[5] `http://hackage.haskell.org/package/ansi-wl-pprint`

[6] Monads can be used to thread a context through a computation, among other things. Explaining monads properly is beyond the scope of this thesis, however.

no new constraints to be discovered. This procedure gives the same result as the formal typing rules because it requires no more than the necessary Data constraints. To illustrate the method, consider the following definitions.

data List a = Nil | Cons a (List a)
data Alt a b = End | Cont a (Alt b a)

The second data type represents a list with alternating types. How does the fixpoint iteration proceed? At first, $I_{\mathsf{List}} = \emptyset$ and $I_{\mathsf{Alt}} = \emptyset$. In the first iteration, the index 1 for the type variable a is added to I_{List} and I_{Alt} since a is a constructor argument type. The second constructor arguments of Cons and Cont do not produce new constraints since List and Nat have no Data requirements in the beginning, yet. After the first iteration, we thus have $I_{\mathsf{List}} = I_{\mathsf{Alt}} = \{1\}$. In the second iteration, the List a argument of Cons requires a to be a Data type because we now have $1 \in I_{\mathsf{List}}$. So 1 (the index of a) is again added to I_{List}, leaving the set unchanged. Furthermore, the Alt b a argument of Cont requires b to be a Data type, as well, because $1 \in I_{\mathsf{Alt}}$ and b is used as the first type argument of Alt. As b is the second type parameter in the definition of Alt, the index 2 is added to I_{Alt}. The result $I_{\mathsf{List}} = \{1\}, I_{\mathsf{Alt}} = \{1, 2\}$ does not change in the next iteration, which means that it is the desired fixed point.

Chapter 3

Operational Semantics for CuMin

Having defined what well-formed CuMin programs look like, we want to define their meaning by giving them a *semantics*. There are two kinds of semantics, *denotational* and *operational* ones. A denotational semantics describes the meaning of programs as mathematical objects. Denotational semantics for CuMin and SaLT were given in [11] and were implemented by Fabian Thorand in his bachelor thesis.

The other approach are operational semantics. These describe the program's execution directly, instead of translating it to mathematical objects. An operational semantics for (a subset of) Curry can be found in [4], which is a modification of [2]. Based on this, Stefan Mehner describes such a semantics [10] for the variant of CuMin without general algebraic data types from [11]. A few small changes and generalizations lead to the semantics I describe below. Before that, I want to point out some properties of CuMin that may seem surprising at first.

3.1 Peculiarities of CuMin

To gain a better understanding of nondeterminism in CuMin, let us look at some examples of CuMin functions.

$$\begin{aligned}
&\mathit{coin} :: \mathsf{Nat}\\
&\mathit{coin} = \mathit{choose}_{\mathsf{Nat}}\ 0\ 1\\
&\mathit{double} :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{double}\ x = x + x\\
&\mathit{maybeDouble1} :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{maybeDouble1} = \mathit{choose}_{\mathsf{Nat}\to\mathsf{Nat}}\ \mathit{id}_{\mathsf{Nat}}\ \mathit{double}\\
&\mathit{maybeDouble2} :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{maybeDouble2}\ n = \mathit{maybeDouble1}\ n
\end{aligned}$$

Contrary to what one might expect, $\mathit{coin} + \mathit{coin}$ and *double coin* behave differently. The first one evaluates to 0, 1 or 2 as each of the summands can yield 0 or 1. The second expression can only evaluate to 0 or 2. This is because of an intricacy of CuMin (and Curry), called *call-time choice*. It means that occurrences of x in the body of *double* always represent the same shared value. The choice for the desired value of the nondeterministic operation *coin* is made at call-time. However, the value itself is still computed lazily (call-by-need). Call-time choice also affects let-bindings: **let** $x = \mathit{coin}$ **in** $x + x$ behaves exactly the same as *double coin*. In particular, one cannot substitute *coin* for x in $x + x$ without changing the meaning. In contrast, this is fine in purely functional languages like Haskell.

Coming from Haskell, it will also be surprising that *maybeDouble1* and *maybeDouble2* are *not* equivalent. This shows that η-equivalence does not generally hold for CuMin (and Curry). The difference between these two functions can only be observed when they are used as a higher-order function argument: The expression $map_{\mathsf{Nat},\mathsf{Nat}}$ *maybeDouble1* $[1,3]_{\mathsf{Nat}}$ evaluates to $[1,3]_{\mathsf{Nat}}$ or $[2,6]_{\mathsf{Nat}}$. This is because when *map* is called, *maybeDouble1* is chosen to be id_{Nat} or *double* due to call-time choice. This means that it will act the same way on each list element. On the other hand, the expression $map_{\mathsf{Nat},\mathsf{Nat}}$ *maybeDouble2* $[1,3]_{\mathsf{Nat}}$ evaluates to $[1,3]_{\mathsf{Nat}}$, $[1,6]_{\mathsf{Nat}}$, $[2,3]_{\mathsf{Nat}}$ or $[2,6]_{\mathsf{Nat}}$. The reason is that *maybeDouble2* is not "directly" nondeterministic, only when applied to an argument.

3.2 Formal Description of the Semantics

When evaluating CuMin expressions, we will have to keep track of variable assignments.

Definition 1 (Heap)**.** *A* heap *is a mapping $[x_1 \mapsto e_1, \dots, x_n \mapsto e_n]$ where the x_i are variables and each e_i is either an expression or a special marker* **free** $:: \tau$*, in which case x_i is called a* logic variable *of type τ. Every variable that occurs in an expression e_i has to be in the heap as well.*

Given two heaps Δ, Δ' with disjoint variables, one can form their union*. It is written by juxtaposition: $\Delta\ \Delta'$.*

Every expression will be associated with a corresponding heap that binds (at least) all the variables contained in the expression.

Definition 2 (Heap Expression Pair)**.** *A* heap expression pair $\Delta : e$ *is a heap Δ together with an expression e such that every unbound variable occurring in e is in the heap Δ.*

The operational semantics will describe how such heap expression pairs are evaluated. In the most common case, we want to know the value of an expression e in the context of a certain CuMin program, without a given heap. In this case, we simply evaluate the heap expression pair $[\,] : e$. When talking about evaluation, one has to specify when an expression is called *evaluated*. The following three notions will be useful:

Definition 3 (Normal Forms)**.** *An expression e (associated with a heap) is said to be*

1. *in* flat normal form (FNF) *if e is a literal, a partial or full application of a data constructor to heap variables, or a partial application of a top-level function f to heap variables. "Partial" here means that the number of arguments given to f is strictly smaller than the number of arguments in the definition of the function f.*

2. *a* value *if e is in flat normal form or a logic variable.*

3. *in* reduced normal form (RNF) *if e is a literal, a partial or full application of a data constructor to expressions in reduced normal form, or a partial application of a top-level function to expressions in reduced normal form.*

As an illustration, let us look at some expressions in the context of the following heap.

$$\Delta := [\,choice \mapsto \textbf{free} :: \mathsf{Bool}, x \mapsto \textbf{free} :: \mathsf{Nat}, y \mapsto x + x, n \mapsto \mathsf{Nil}_{\mathsf{Bool}}\,]$$

In this environment, *choice* is a value but not in FNF or RNF. Furthermore, $\mathsf{Cons}_{\mathsf{Bool}}$ *choice n* is in FNF and thus also a value but still not in RNF. On the other hand, $\mathsf{Cons}_{\mathsf{Bool}}\ \mathsf{True}\ \mathsf{Nil}_{\mathsf{Bool}}$ is in RNF but neither in FNF nor a value since True and $\mathsf{Nil}_{\mathsf{Bool}}$ are not variables. Additionally, there are terms like 2, $\mathsf{Nil}_{\mathsf{Nat}}$, or $map_{\mathsf{Bool},\mathsf{Nat}}$ that are values, in FNF and in RNF.

For each normal form, there is a corresponding evaluation strategy:

1. *Logical evaluation*, denoted by $\Downarrow^*$, which evaluates to values,
2. *Functional evaluation*, denoted by $\Downarrow$, which evaluates to flat normal form.
3. *Evaluation to reduced normal form*, denoted by $\Downarrow^!$ and sometimes also called "forcing".

These normal forms can be obtained by using the rules shown in figs. 3.1 to 3.3. Like the typing rules, the evaluation rules are implicitly indexed by a given CuMin program. This is again omitted from the notation for the sake of readability. Another technicality to discuss is related to substitution: A variable is called *fresh* if its name does not occur in the relevant expression and the CuMin program. Since we only ever substitute with fresh variables in the evaluation rules, variable capture (cf. section 2.5.1) cannot happen.

3.3 Explanation of the Semantics

First note that the evaluation of an expression is not unique. That is what nondeterminism is about, after all. Some expressions can even evaluate to infinitely many values. A simple example for that would be

$$\text{Guess}_n\ \frac{[x \mapsto \textbf{free} :: \mathsf{Nat}] : x \Downarrow^* [x \mapsto \textbf{free} :: \mathsf{Nat}] : x}{[x \mapsto \textbf{free} :: \mathsf{Nat}] : x \Downarrow [x \mapsto n] : n}$$

which works for any natural number n.

On the other hand, not every expression has a normal form. A trivial example is $\textbf{failed}_\tau$, which simply has no applicable reduction rule. This makes sense because this expression denotes failure. Having clarified that, let us take a look at the individual rules of logical evaluation.

- **Val.** If an expression is already a value, it does not need to be evaluated further.
- Rules related to variables:
  - **Lookup.** Given a heap variable that is not a logic variable, evaluate the expression bound by the variable. Additionally, the heap must be updated to ensure sharing. This avoids recomputing the same expression repeatedly. Also, if the heap were not updated, a variable could nondeterministically evaluate to different values depending on where it is used, but instead, call-time choice is desired.
  - **Let.** To evaluate a **let** binding, the bound variable and expression are added to the heap and the body is evaluated. The variable has to be replaced by a fresh name so that it will not shadow other heap variables. Note that the evaluation of the bound expression is deferred until it is needed (Lookup). This is *lazy evaluation.*
  - **Free.** Evaluating **let** . . . **free** bindings works analogously.
- **Function application.** There are three rules governing function application. Together, they ensure the intended behavior: lazy evaluation and call-time choice.
  - **Fun.** This rule can be used whenever a top-level function is fully applied and its arguments are only heap variables. The function call is then replaced by the right hand side of the function definition, with variables and types properly substituted. The reason that the arguments must be variables is to ensure call-time choice and sharing.
  - **Flatten.** To be able to apply the previous rule, all function arguments must be variables. This rule allows to achieve this by introducing **let** bindings. The argument of a function application is replaced by a fresh variable.

$$[\text{Val}] \quad \Delta : v \Downarrow^* \Delta : v \qquad \text{if } v \text{ is a value w.r.t. } \Delta$$

$$[\text{Lookup}] \quad \frac{\Delta : e \Downarrow^* \Delta' : v}{\Delta\,[x \mapsto e] : x \Downarrow^* \Delta'\,[x \mapsto v] : v} \qquad \text{unless } x \text{ is a logic variable}$$

$$[\text{Let}] \quad \frac{\Delta\,[y \mapsto e_1] : e_2\,[y/x] \Downarrow^* \Delta' : v}{\Delta : \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow^* \Delta' : v} \qquad \text{where } y \text{ is fresh}$$

$$[\text{Free}] \quad \frac{\Delta\,[y \mapsto \textbf{free} :: \tau] : e\,[y/x] \Downarrow^* \Delta' : v}{\Delta : \textbf{let } x :: \tau \textbf{ free in } e \Downarrow^* \Delta' : v} \qquad \text{where } y \text{ is fresh}$$

$$[\text{Fun}] \quad \frac{\Delta : e\,[\overline{\tau_m/\alpha_m}, \overline{y_n/x_n}] \Downarrow^* \Delta' : v}{\Delta : f_{\overline{\tau_m}}\ \overline{y_n} \Downarrow^* \Delta' : v}$$

for a top-level function $f :: \forall\overline{\alpha_m}.(\mathsf{Data}\ldots) \Rightarrow \tau; f\ \overline{x_n} = e$

$$[\text{Flatten}] \quad \frac{\Delta : \textbf{let } y = e \textbf{ in } \varphi\ y \Downarrow^* \Delta' : v}{\Delta : \varphi\ e \Downarrow^* \Delta' : v} \qquad \text{where } \varphi \text{ is in flat normal form and } y \text{ fresh}$$

$$[\text{Apply}] \quad \frac{\Delta : e_1 \Downarrow^* \Delta' : \varphi \qquad \Delta' : \varphi\ e_2 \Downarrow^* \Delta'' : v}{\Delta : e_1\ e_2 \Downarrow^* \Delta'' : v}$$

$$[\text{Plus}] \quad \frac{\Delta : e_1 \Downarrow \Delta' : n_1 \qquad \Delta' : e_2 \Downarrow \Delta'' : n_2}{\Delta : e_1 + e_2 \Downarrow^* \Delta'' : n} \qquad \text{for literals } n_1,\ n_2 \text{ and } n = n_1 + n_2$$

$$[\text{EqNat}] \quad \frac{\Delta : e_1 \Downarrow \Delta' : n_1 \qquad \Delta' : e_2 \Downarrow \Delta'' : n_2}{\Delta : e_1 \equiv e_2 \Downarrow^* \Delta'' : b}$$

where n_1, n_2 are literals and b is True if $n_1 = n_2$, False otherwise.

$$[\text{Eq}] \quad \frac{\Delta : e_1 \Downarrow \Delta' : \mathsf{C}_{\overline{\tau_m}}\ \overline{x_n} \qquad \Delta' : e_2 \Downarrow \Delta_0 : \mathsf{C}_{\overline{\tau_m}}\ \overline{y_n} \qquad \overline{\Delta_{n-1} : x_n \equiv y_n \Downarrow \Delta_n : \mathsf{True}}}{\Delta : e_1 \equiv e_2 \Downarrow \Delta_n : \mathsf{True}}$$

$$[\text{NEq}] \quad \frac{\Delta : e_1 \Downarrow \Delta' : \mathsf{C}_{\overline{\tau_m}}\ \overline{x_n} \qquad \Delta' : e_2 \Downarrow \Delta_0 : \mathsf{C}_{\overline{\tau_m}}\ \overline{y_n} \qquad \overline{\Delta_{i-1}\ : x_i \equiv y_i \Downarrow \Delta_i : b_i}}{\Delta : e_1 \equiv e_2 \Downarrow \Delta_i : \mathsf{False}}$$

where $i \in \{1, \ldots, n\}$ and b_j is True for all $j \in \{1, \ldots, i-1\}$ but b_i is False.

$$[\text{NEqCon}] \quad \frac{\Delta : e_1 \Downarrow \Delta' : \mathsf{C}_{\overline{\tau_m}}\ \overline{x_n} \qquad \Delta' : e_2 \Downarrow \Delta'' : \mathsf{D}_{\overline{\tau_m}}\ \overline{y_n}}{\Delta : e_1 \equiv e_2 \Downarrow \Delta'' : \mathsf{False}}$$

where C and D are different constructors.

$$[\text{CaseCon}] \quad \frac{\Delta : e \Downarrow \Delta' : \mathsf{C}_{\overline{\tau_m}}\ \overline{y_n} \qquad \Delta' : e'\,[\overline{y_n}/\overline{x_n}] \Downarrow^* \Delta'' : v}{\Delta : \textbf{case } e \textbf{ of } \{\ldots; \mathsf{C}\ \overline{x_n} \to e'; \ldots\} \Downarrow^* \Delta'' : v}$$

$$[\text{CaseVar}] \quad \frac{\Delta : e \Downarrow \Delta' : \mathsf{C}_{\overline{\tau_m}}\ \overline{x_n} \qquad \Delta'\,[y \mapsto \mathsf{C}_{\overline{\tau_m}}\ \overline{x_n}] : e'\,[y/x] \Downarrow^* \Delta'' : v}{\Delta : \textbf{case } e \textbf{ of } \{\ldots; x \to e'\} \Downarrow^* \Delta'' : v} \qquad \text{where } y \text{ is fresh}$$

Only applicable if none of the constructor patterns matched the constructor C.

Figure 3.1: Rules for logical evaluation

$$[\text{FNF}]\quad \frac{\Delta : e \Downarrow^{*} \Delta' : v}{\Delta : e \Downarrow \Delta' : v} \qquad \text{where } v \text{ is in flat normal form}$$

$$[\text{Guess}_n]\quad \frac{\Delta : e \Downarrow^{*} \Delta'\ [x :: \mathsf{Nat} \mapsto \mathbf{free}] : x}{\Delta : e \Downarrow \Delta'\ [x :: \mathsf{Nat} \mapsto n] : n} \qquad \text{for any natural number } n$$

$$[\text{Guess}_{\mathsf{C}}]\quad \frac{\Delta : e \Downarrow^{*} \Delta'\ [x \mapsto \mathbf{free} :: \mathsf{A}\ \overline{\rho_m}] : x}{\Delta : e \Downarrow \Delta'\ [\overline{y_n \mapsto \mathbf{free} :: \tau_n\ [\overline{\rho_m/\alpha_m}]}, x \mapsto \mathsf{C}_{\overline{\rho_m}}\ \overline{y_n}] : \mathsf{C}_{\overline{\rho_m}}\ \overline{y_n}} \qquad \text{for any constructor } \mathsf{C}$$

of the ADT $\mathsf{A}\ \overline{\alpha_m}$ with argument types $\overline{\tau_n}$ and where $\overline{y_n}$ are fresh variables

Figure 3.2: Rules for functional evaluation

$$[\text{RNF}]\quad \frac{\Delta : e \Downarrow \Delta' : v}{\Delta : e \Downarrow^{!} \Delta' : v} \qquad \text{where } v \text{ is in reduced normal form}$$

$$[\text{Force}]\quad \frac{\Delta : e \Downarrow \Delta_0 : f_{\overline{\rho_m}}\ \overline{y_n} \qquad \Delta_0 : y_1 \Downarrow^{!} \Delta_1 : e_1 \qquad \dots \qquad \Delta_{n-1} : y_n \Downarrow^{!} \Delta_n : e_n}{\Delta : e \Downarrow^{!} \Delta_n : f_{\overline{\rho_m}}\ \overline{e_n}}$$

where f is a constructor or top-level function

Figure 3.3: Rules for evaluation to reduced normal form

Note that this rule can be applied even if all arguments are already variables. This unnecessary indirection is computationally undesirable but not forbidden according to this semantics. However, it is avoided in the implementation.

- **Apply.** The last rule for function application can always be applied. (However, it is only useful if the other rules don't make progress.) If the function is not already a top level function or constructor, but a more complex expression, this rule allows it to be evaluated to flat normal form. This way, after possibly having applied Flatten several times, the expression will be a value or have the shape required by Fun.

• **Plus.** Addition is a primitive operation, it has to deal with concrete numbers, so its arguments must be in flat normal form. Since they have type Nat, they already have to be literals. So to evaluate the expression, the two arguments are evaluated *functionally* and the sum of the two literals is the result.

• **Equality tests.** In order to find out whether two expressions are equal, they first have to be evaluated to at least flat normal form. How evaluation continues, depends on the result.

- **EqNat.** If the resulting values are natural numbers, the comparison yields True if the numbers are equal, and False otherwise.
- **Eq.** If the resulting values are of an algebraic data type, they are equal if and only if their constructors match and each of their arguments are equal, which is checked recursively.
- **NEq.** If the constructors match, but some of their arguments are not equal, then evaluation stops at the first pair of arguments to differ. The remaining arguments are *not* evaluated. The comparison yields False.
- **NEqCon.** If the constructors do not match, the comparison yields False immediately. No arguments are evaluated in this case.

- **Case expressions.** In a case expression, the first part to be evaluated is the scrutinee. It has to be evaluated *functionally* since logic variables do not admit case analysis. Since the type of the scrutinee is a data type, it must be a fully applied constructor C. One of the following two rules can be applicable.
  - **CaseCon.** If one of the patterns in the case alternatives matches the constructor C, the corresponding expression is evaluated. Of course, the variables introduced in the constructor pattern must be replaced by the arguments of C.
  - **CaseVar.** If none of the constructor patterns matched, the case alternative of a “catch-all” variable pattern (if there is one) is chosen. This acts just like a **let** binding, the only difference being that the bound expression is always in flat normal form.

Functional evaluation essentially uses logical evaluation and then takes care of logic variables if there are any. In detail:

- **FNF.** If the logical evaluation of an expression is already in flat normal form, this is also the result of functional evaluation.
- **Guess$_n$.** If an expression evaluates to a logic variable of type Nat, its flat normal form is any natural number. The value of the variable is updated on the heap, too.
- **Guess$_C$.** If an expression evaluates to a logic variable with an algebraic data type, its flat normal form is given by any constructor C of the ADT, fully applied to new logic variables that are added to the heap. The value of the original variable is updated on the heap, as before.

These last two rules are the source of nondeterminism. Which constructor or which natural number is chosen for a logic variable is not determined, in contrast to the other rules where choices do not affect the result.

Finally, the reduced normal form is obtained by first evaluating functionally and then reducing subexpressions as much as possible. In detail:

- **RNF.** If the functional evaluation of an expression is already in reduced normal form, there is nothing to be done.
- **Force.** If the flat normal form of an expression is a constructor or function application, all its arguments are forced to reduced normal form. Then the result is in reduced normal form as well.

The operational semantics given above differs from [10] in that it supports general algebraic data types (not only lists and booleans), case expressions with catch-all variable patterns, and equality tests for arbitrary Data types, not only natural numbers. Although it is based on [2] and [4], these sources do not discriminate between functional and logic evaluation. Moreover, they require the input programs to be in a certain normalized form before they can be evaluated: Nested constructor and function invocations have to be eliminated. In the semantics given here, this normalization happens during evaluation, by means of the Flatten rule. The evaluation to reduced normal form is not explicitly described in these sources, but it is relatively straightforward and makes the implementation of the semantics much more convenient to use.

3.4 Examples

In order to understand the evaluation rules better, it is instructive to look at some examples. Starting simple, let us look at the logical evaluation of *double* 1.

$$\text{Flatten}\,\dfrac{\text{Let}\,\dfrac{\text{Fun}\,\dfrac{\text{Plus}\,\dfrac{\text{FNF}\,\dfrac{\text{Lookup}\,\dfrac{[y \mapsto 1] : y \Downarrow^* [y \mapsto 1] : 1}{[y \mapsto 1] : y \Downarrow^* [y \mapsto 1] : 1}}{[y \mapsto 1] : y \Downarrow [y \mapsto 1] : 1} \qquad \text{FNF}\,\dfrac{\text{Lookup}\,\dfrac{[y \mapsto 1] : y \Downarrow^* [y \mapsto 1] : 1}{[y \mapsto 1] : y \Downarrow^* [y \mapsto 1] : 1}}{[y \mapsto 1] : y \Downarrow [y \mapsto 1] : 1}}{[y \mapsto 1] : y + y \Downarrow^* [y \mapsto 1] : 2}}{[y \mapsto 1] : \mathit{double}\ y \Downarrow^* [y \mapsto 1] : 2}}{[\,] : \mathbf{let}\ x = 1\ \mathbf{in}\ \mathit{double}\ x \Downarrow^* [y \mapsto 1] : 2}}{[\,] : \mathit{double}\ 1 \Downarrow^* [y \mapsto 1] : 2}$$

Nondeterminism can lead to more than one result. One of the simplest possible examples is **let** $x :: \mathsf{Bool}$ **free in** x. Its flat normal forms are False and True:

$$\text{Guess}_{\mathsf{False}}\,\dfrac{\text{Free}\,\dfrac{[y \mapsto \mathbf{free} :: \mathsf{Bool}] : y \Downarrow^* [y \mapsto \mathbf{free} :: \mathsf{Bool}] : y}{[\,] : \mathbf{let}\ x :: \mathsf{Bool}\ \mathbf{free\ in}\ x \Downarrow^* [y \mapsto \mathbf{free} :: \mathsf{Bool}] : y}}{[\,] : \mathbf{let}\ x :: \mathsf{Bool}\ \mathbf{free\ in}\ x \Downarrow [y \mapsto \mathsf{False}] : \mathsf{False}}$$

$$\text{Guess}_{\mathsf{True}}\,\dfrac{\text{Free}\,\dfrac{[y \mapsto \mathbf{free} :: \mathsf{Bool}] : y \Downarrow^* [y \mapsto \mathbf{free} :: \mathsf{Bool}] : y}{[\,] : \mathbf{let}\ x :: \mathsf{Bool}\ \mathbf{free\ in}\ x \Downarrow^* [y \mapsto \mathbf{free} :: \mathsf{Bool}] : y}}{[\,] : \mathbf{let}\ x :: \mathsf{Bool}\ \mathbf{free\ in}\ x \Downarrow [y \mapsto \mathsf{True}] : \mathsf{True}}$$

3.4.1 The Coin Example

As a more complex instance, consider *coin*. As an abbreviation, let $\Delta := [x' \mapsto 0, y' \mapsto 1]$ and $\Delta' := \Delta\ [c' \mapsto \mathsf{False}]$.

$$\text{Fun}\,\dfrac{\text{Apply}\,\dfrac{\dfrac{\dots}{[\,] : \mathit{choose}_{\mathsf{Bool}}\ 0 \Downarrow^* [x' \mapsto 0] : \mathit{choose}_{\mathsf{Bool}}\ x'} \qquad \dfrac{\dots}{[x' \mapsto 0] : \mathit{choose}_{\mathsf{Bool}}\ x'\ 1 \Downarrow^* \Delta' : 0}}{[\,] : \mathit{choose}_{\mathsf{Bool}}\ 0\ 1 \Downarrow^* \Delta' : 0}}{[\,] : \mathit{coin} \Downarrow^* [c' \mapsto \mathsf{False}] : 0}$$

The left subderivation continues as follows.

$$\text{Flatten}\,\dfrac{\text{Let}\,\dfrac{[x' \mapsto 0] : \mathit{choose}_{\mathsf{Bool}}\ x' \Downarrow^* [x' \mapsto 0] : \mathit{choose}_{\mathsf{Bool}}\ x'}{[\,] : \mathbf{let}\ x = 0\ \mathbf{in}\ \mathit{choose}_{\mathsf{Bool}}\ x \Downarrow^* [x' \mapsto 0] : \mathit{choose}_{\mathsf{Bool}}\ x'}}{[\,] : \mathit{choose}_{\mathsf{Bool}}\ 0 \Downarrow^* [x' \mapsto 0] : \mathit{choose}_{\mathsf{Bool}}\ x'}$$

The right subderivation goes on like this.

$$\text{Flatten}\,\dfrac{\text{Let}\,\dfrac{\text{Fun}\,\dfrac{\text{Free}\,\dfrac{\text{CaseCon}\,\dfrac{\text{Guess}_{\mathsf{False}}\,\dfrac{\Delta\ [c' \mapsto \mathbf{free} :: \mathsf{Bool}] : c' \Downarrow^* \Delta\ [c' \mapsto \mathbf{free} :: \mathsf{Bool}] : c'}{\Delta\ [c' \mapsto \mathbf{free} :: \mathsf{Bool}] : c' \Downarrow \Delta' : \mathsf{False}} \qquad \Delta' : 0 \Downarrow^* \Delta' : 0}{\Delta\ [c' \mapsto \mathbf{free} :: \mathsf{Bool}] : \mathbf{case}\ c'\ \mathbf{of}\ \{\mathsf{False} \to 0; \dots\} \Downarrow^* \Delta' : 0}}{\Delta : \mathbf{let}\ c :: \mathsf{Bool}\ \mathbf{free\ in\ case}\ c\ \mathbf{of}\ \{\mathsf{False} \to 0; \mathsf{True} \to 1\} \Downarrow^* \Delta' : 0}}{\Delta : \mathit{choose}_{\mathsf{Bool}}\ x'\ y' \Downarrow^* \Delta' : 0}}{[x' \mapsto 0] : \mathbf{let}\ y = 1\ \mathbf{in}\ \mathit{choose}_{\mathsf{Bool}}\ x'\ y \Downarrow^* \Delta' : 0}}{[x' \mapsto 0] : \mathit{choose}_{\mathsf{Bool}}\ x'\ 1 \Downarrow^* \Delta' : 0}$$

A completely analogous derivation that uses $\text{Guess}_{\mathsf{True}}$ instead of $\text{Guess}_{\mathsf{False}}$ yields the other evaluation $[\,] : \mathit{coin} \Downarrow^* \Delta\ [c' \mapsto \mathsf{True}] : 1$.

3.4.2 Call-time Choice

In the introduction of this chapter, we discussed the examples $coin + coin$ and $\textbf{let}\ c = coin\ \textbf{in}\ c + c$. Let us find out how the difference between them manifests itself.

$$\text{Plus}\ \dfrac{\text{FNF}\ \dfrac{\dfrac{\dots}{[\,] : coin \Downarrow^* \Delta : i}}{[\,] : coin \Downarrow \Delta : i} \qquad \text{FNF}\ \dfrac{\dfrac{\dots}{\Delta : coin \Downarrow^* \Delta' : j}}{\Delta : coin \Downarrow \Delta' : j}}{[\,] : coin + coin \Downarrow^* \Delta' : i + j}$$

This derivation works for all $i = 0, 1$ and $j = 0, 1$. Thus, the possible results are 0, 1 and 2. In contrast, consider the derivation for $\textbf{let}\ c = coin\ \textbf{in}\ c + c$.

$$\text{Let}\ \dfrac{\text{Plus}\ \dfrac{\text{FNF}\ \dfrac{\text{Lookup}\ \dfrac{\dfrac{\dots}{[c' \mapsto coin] : coin \Downarrow \Delta\,[c' \mapsto i] : i}}{[c' \mapsto coin] : c' \Downarrow^* \Delta\,[c' \mapsto i] : i}}{[c' \mapsto coin] : c' \Downarrow \Delta\,[c' \mapsto i] : i} \qquad \text{FNF}\ \dfrac{\text{Lookup}\ \dfrac{\Delta\,[c' \mapsto i] : c' \Downarrow^* \Delta\,[c' \mapsto i] : i}{\Delta\,[c' \mapsto i] : c' \Downarrow^* \Delta\,[c' \mapsto i] : i}}{\Delta\,[c' \mapsto i] : c' \Downarrow \Delta\,[c' \mapsto i] : i}}{[c' \mapsto coin] : c' + c' \Downarrow^* \Delta\,[c' \mapsto i] : i + i}}{[\,] : \textbf{let}\ c = coin\ \textbf{in}\ c + c \Downarrow^* \Delta\,[c' \mapsto i] : i + i}$$

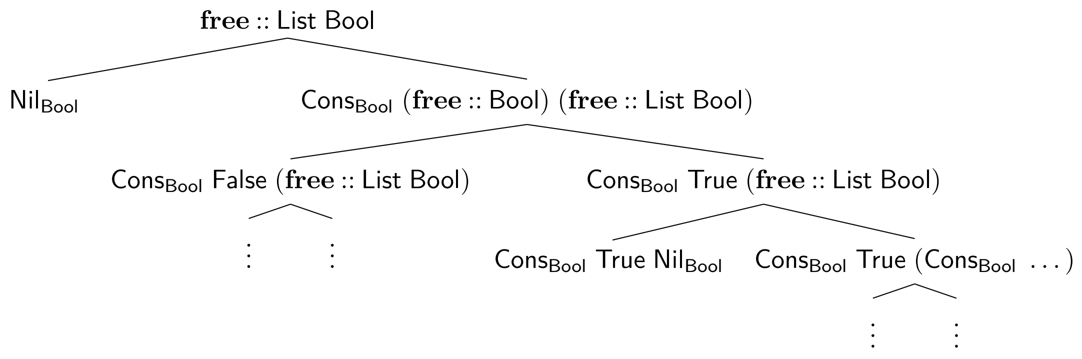
Again, this works for all $i = 0, 1$. But the two summands are not independent in this case. Hence, the only possible results are 0 and 2.

3.4.3 Reduced Normal Form

So far, we have not dealt with $\Downarrow^!$. For sake of completeness, we will analyze how $\textbf{let}\ y :: \textsf{List Bool}\ \textbf{free in}\ y$ can evaluate to $\textsf{Cons}_{\textsf{Bool}}\ \textsf{True}\ \textsf{Nil}_{\textsf{Bool}}$. For lack of space, the heaps are omitted. Nevertheless, the idea should become clear.

$$\text{Force}\ \dfrac{\text{Guess}_{\textsf{Cons}}\ \dfrac{\text{Free}\ \dfrac{(\text{as above} \dots)}{\textbf{let} \dots \Downarrow^* y'}}{\textbf{let} \dots \Downarrow \textsf{Cons}_{\textsf{Bool}}\ x\ xs} \qquad \text{RNF}\ \dfrac{\text{Guess}_{\textsf{True}}\ \dfrac{\text{FNF}\ \dfrac{\dots}{x \Downarrow^* x}}{x \Downarrow \textsf{True}}}{x \Downarrow^! \textsf{True}} \qquad \text{RNF}\ \dfrac{\text{Guess}_{\textsf{Nil}}\ \dfrac{\text{FNF}\ \dfrac{\dots}{xs \Downarrow^* xs}}{xs \Downarrow \textsf{Nil}_{\textsf{Bool}}}}{xs \Downarrow^! \textsf{Nil}_{\textsf{Bool}}}}{\textbf{let}\ x :: \textsf{List Bool}\ \textbf{free in}\ x \Downarrow^! \textsf{Cons}_{\textsf{Bool}}\ \textsf{True}\ \textsf{Nil}_{\textsf{Bool}}}$$

The different choices of the applied Guess rules can be visualized in a tree like this.


free :: List Bool
Nil_Bool
Cons_Bool (free :: Bool) (free :: List Bool)
Cons_Bool False (free :: List Bool)
Cons_Bool True (free :: List Bool)
Cons_Bool True Nil_Bool
Cons_Bool True (Cons_Bool ...)

3.5 Implementation

3.5.1 Nondeterminism and Search

While the operational semantics formalizes which evaluations are valid, it does not directly specify how all the nondeterministic results of a computation can be found, i.e. what constructors should be chosen in the Guess rules and in what order. As we have seen in the previous section, trees are a natural representation for this.

The implementation of the semantics is thus split into two components: the actual evaluation of an expression, which generates such an evaluation tree; and the traversal of this tree, for example using depth-first or breadth-first search. This may sound inefficient at first, as only parts of the generated tree may actually be needed. However, since Haskell is a lazy language, the tree is generated on demand, while being traversed. In this way, laziness allows decoupling evaluation and search.

3.5.2 Evaluation

Before explaining the actual implementation, I have to describe how the various objects and normal forms where modeled as data types.

A heap is represented as a standard map data structure (Data.Map) with variable names (strings) as keys and expressions as values. The normal forms are algebraic data types, as expected.

```
data FNF
  = PartialApp BindingName [Type] [VarName]
  | ConValue DataConName [Type] [VarName]
  | Literal Lit
data Value
  = Fnf FNF
  | Logic VarName Type
```

Furthermore, there are (among others) the following evaluation functions.

```
evaluateFunctionally :: Exp → EvalT TreeM FNF
evaluateLogically    :: Exp → EvalT TreeM Value
```

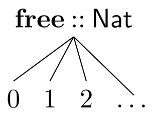
These functions accept a CuMin expression and produce a tree with all possible results at the leaves, flat normal forms and values, respectively. EvalT is a monad transformer [9] that manages the state of the computation, namely the heap and a counter for generating fresh names, as well as an environment with the data types and functions of the CuMin program. TreeM is a monad that supports nondeterminism by building an evaluation tree. EvalT TreeM is a monad that combines these stateful and nondeterministic effects. This way, one does not have to manage the state and trees explicitly throughout the computation; it is instead handled by the monadic bind operation.

As for the implementation of the evaluation rules, there are some details to be discussed. First, the rules often require fresh variables to be added to the heap. To guarantee that all variables are fresh, every new variable name includes the current value of a counter which is increased afterwards, hence ensuring that all generated variables are unique. Avoiding variable capture on substitution (cf. section 2.5.1) has to be taken care of as well. In this case, however, it is not an issue since one only ever substitutes fresh variables for existing ones.

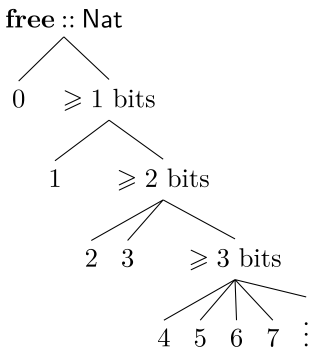
In most cases, the shape of an expression determines the next evaluation rule to apply. For instance, if it is an addition, only the Plus rule can be applied. In other cases, like in case expressions or equality tests, parts of the expression have to be evaluated, and then there is only one rule that matches. Logic variables are inherently nondeterministic, so in this case more than one rule is applicable, and the evaluation tree branches. The only remaining case is function application. The rules Fun, Apply, Flatten can be employed and there may be more than one choice. The implementation uses the following strategy: It applies the Fun rule first, whenever possible. If not, it tries the Apply rule. In case this also had no effect, it uses the Flatten rule. This strategy always makes progress (cf. section 3.3).

All in all, logical evaluation proceeds like this: It checks whether the expression is already a value, and if so, does nothing (Val rule). Otherwise, depending on the shape of the expression, a suitable rule is applied according to the details in the previous paragraph. Functional evaluation invokes logical evaluation first. If the result is already in flat normal form, nothing has to be done (FNF rule). Otherwise, the result is a logic variable which results in a branching of the evaluation tree, one new branch for each applicable Guess rule. Evaluation to reduced normal form uses functional evaluation. If the result is in RNF, nothing is to be done (RNF rule). Otherwise, the subexpressions are recursively forced (Force rule).

One more thing to discuss is guessing natural numbers. One could simply generate them in a tree like this:



The disadvantage is that breadth-first search will not find all solutions on such a tree since it contains a node with infinitely many direct children. As completeness of BFS is desirable, the program will instead generate a tree with only finitely many nodes on each level, namely the numbers with i bits on the i-th level. This tree also yields the same result, independently of whether BFS or DFS is used:



3.5.3 Trees and Traversals

An evaluation tree is represented by the data type

$$\textbf{data}\ \mathsf{Tree}\ a = \mathsf{Leaf}\ a \mid \mathsf{Branches}\ [\mathsf{Tree}\ a]$$

It can be made a monad[1], i.e. there are two functions $\mathit{return} :: a \to \mathsf{Tree}\ a$ and $(\gg=) :: \mathsf{Tree}\ a \to (a \to \mathsf{Tree}\ b) \to \mathsf{Tree}\ b$. The former simply creates a leaf and the bind operation $\gg=$ performs substitution on the leaves. The operation that is important for nondeterminism is given by the function

$$\begin{aligned}&\mathit{branch} :: [a] \to \mathsf{Tree}\ a\\ &\mathit{branch} = \mathsf{Branches} \circ \mathit{map}\ \mathsf{Leaf}\end{aligned}$$

which creates a tree with the given leaves. Failure is represented by a tree without leaves:

$$\begin{aligned}&\mathit{failure} :: \mathsf{Tree}\ a\\ &\mathit{failure} = \mathit{branch}\ [\,]\end{aligned}$$

[1] In fact, this type constructor represents the free monad over the list functor.

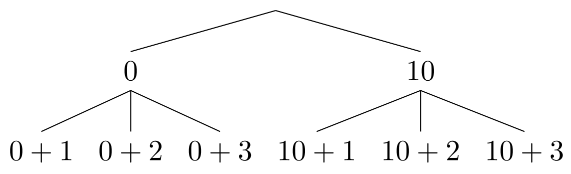
In order to better understand how these functions work, consider the following code sample.

```
branch [0, 10] ≫= λx → branch [1, 2, 3] ≫= λy → return (x + y)
```

There is special notation for ≫= in Haskell, which allows it to be rewritten like this.

```
do x ← branch [0, 10]
   y ← branch [1, 2, 3]
   return (x + y)
```

The code produces the tree visualized below. As the example demonstrates, *branch* produces the nondeterminism, and the monadic structure of Tree hides the composition of this effect, namely the substitution of trees.



However, this tree structure performed rather badly. Profiling the application revealed that most of the time was spent performing substitution on the tree. This takes a lot of time for deep trees since it has to be traversed completely to get to the leaves. This problem turns out to be well-known [14]. The solution is called the *codensity transformation*. It works by "focusing" on the leaves of the tree to speed up substitution. The transformed type looks like this:

```
newtype CTree a = CTree (∀r.(a → Tree r) → Tree r)
```

It represents the tree by a function taking a function of type $a \to$ Tree r which, when applied to the leaves of the CTree a, yields a Tree r. It can be given a monad instance that performs much better. For certain evaluation trees with lots of branches, it was more than ten times faster. Such a CTree is converted to a Tree after construction, so that it can be traversed.

I implemented two kinds of traversals for evaluation trees: breadth-first search and depth-first search, each with and without a depth limit. Each of them has advantages and drawbacks. Depth-first search is incomplete: In many infinite trees, not every solution will be found in finite time. Breadth-first search is complete but uses more memory. A more detailed comparison of the two can be found in section 3.6.

It was mentioned before that the monad EvalT TreeM combines the nondeterministic effects of trees with the stateful effects of the EvalT monad transformer. The implementation of the Guess rule for natural numbers demonstrates such a usage. The part of the function that generates the evaluation tree for a logic variable v is given below in a slightly simplified form.

```
do n ← lift (branchNatLongerThan 0)
   updateVarOnHeap v (ELit (LNat n))
   return (Literal (LNat n))
```

In the following explanation, I will omit some technical details, such as the *lift* function, but it should give a rough idea of how it works: The expression *branchNatLongerThan* 0 uses *branch* internally to create the evaluation tree for natural numbers that was discussed in the previous section. For every natural number n in this tree, *updateVarOnHeap* sets the value of the variable v on the heap to the literal n, in each branch of the computation. Finally, the result of the computation is given by the literal n, which is in flat normal form. This example combines both nondeterministic and stateful effects, triggered by functions like *branch* or *updateVarOnHeap*. One does not have to manually propagate them through the program, which makes the actual computation much clearer.

3.5.4 REPL

As an interface to the evaluation functions, I created a REPL (read-evaluate-print-loop) where the user can enter expressions and have them evaluated.

The REPL can evaluate expressions functionally, when given the commands `:e` or `:eval`, printing the results together with the corresponding heap. When no command is given, it evaluates expressions to reduced normal form. This can also be explicitly specified by the commands `:f` or `:force`. In this case, the heap is unnecessary, since there are no variables in the reduced normal form. The expressions are type checked before evaluation and the computation time is displayed afterwards. Evaluation can also be interrupted with the key combination `Ctrl + C`, which is useful for non-terminating expressions. To quit the REPL, the user should type `:q` or `:quit`.

There are two parameters that the user can change, namely the search depth limit, which is infinity by default, and the search strategy, which is BFS by default. The values of the parameters can be viewed with `:get` and they can be changed using `:set`.

As an illustration, let us have a look at an example session. The REPL was loaded with the file `List.cumin`, which contains the function *last* from the introduction, as a command-line argument.

```
Type ":h" (without quotes) for help.
Loaded module ‘List‘.
> :h
List of commands:
 * :h, :help
        Show this help text.
 * :q, :quit
        Quit the program.
 * :r, :reload
        Reload the current module.
 * :f <expr>, :force <expr>, <expr>
        Force the expression <expr> to reduced normal form.
 * :e <expr>, :eval <expr>
        Evaluate the expression <expr> to flat normal form.
 * :g, :get
        List all configurable properties and their current values.
 * :s <prop>=<val>, :set <prop>=<val>
        Set property <prop> to value <val>. For details use ':get'.
> last<:Bool:> [True, False]<:Bool:>
 :: Bool
 = False<::>

CPU time elapsed: 0.000 s
> let x :: Bool free in x
 :: Bool
 = False<::>
 = True<::>

CPU time elapsed: 0.000 s
> choose<:Bool:> True False
 :: Bool
 = True<::>
 = False<::>

CPU time elapsed: 0.000 s
```

```
> :get
Current settings:
 * depth=inf:
   The search depth limit. Values: inf, 0, 1, 2, ...
 * strategy=bfs:
   The search strategy: bfs, dfs
> :set depth=3
> let x :: List Bool free in x
 :: List Bool
 = Nil<:Bool:>
 = Cons<:Bool:> False<::>
     Nil<:Bool:>
 = Cons<:Bool:> True<::>
     Nil<:Bool:>

CPU time elapsed: 0.000 s
> :e let x :: List Bool free in x
 :: List Bool
 ~> [_1_x -> Nil<:Bool:>]
      : Nil<:Bool:>
 ~> [_1_x -> Cons<:Bool:>
    _2_conArg _3_conArg
    ,_2_conArg -> free :: Bool
    ,_3_conArg -> free :: List Bool]
      : Cons<:Bool:> _2_conArg
      _3_conArg

CPU time elapsed: 0.000 s
> :q
Bye.
```

The sample session demonstrates the evaluation of a deterministic example, namely the *last* function, and nondeterministic examples with logic variables. In the latter case, evaluation has more than one result, and all of them are displayed. The use of `:get` shows the default strategy and search depth limit. Afterwards, the latter is set to 3 with the `:set` command. Only because of that, the next evaluation terminates, yielding three boolean lists. Without the depth limit, there would be an infinite number of results. While all expressions so far were evaluated to reduced normal form, the last example uses the command `:e` to evaluate only to flat normal form. As a consequence, the results are not fully evaluated, e.g. the `Cons` constructor is applied to logic variables on the heap.

The REPL is implemented using the *Haskeline* library[2], which is also used by GHCi, the REPL of the Glasgow Haskell compiler. It provides a history of the previous inputs that can be selected using the up and down keys.

3.5.5 Testing

In order to verify the correctness of the interpreter, I created a test-suite called `cumin-test` in the implementation. It performs two kinds of checks: First, it evaluates certain test expressions to reduced normal form, and compares them with the corresponding expected results. As a second check, I used a denotational semantics for CuMin, implemented by Fabian Thorand in his bachelor thesis. The test expressions are also evaluated using this implementation, and it is checked whether the results agree with the operational semantics.

[2] `https://hackage.haskell.org/package/haskeline`

Program	All results computed?	DFS	BFS
Peano subtraction	all results	78 ms	72 ms
Peano division	all results	110 ms	110 ms
Last	all results	2.3 ms	2.2 ms
Permutation sort	all results	3.9 ms	4.0 ms
Nat subtraction	only the first one	38 ms	39 ms
Nat multiplication	only the first one	0.28 ms	58 ms

Figure 3.4: Average running times with different search strategies

3.6 Assessment of the Search Strategies

To examine the effect of different search strategies, I created a benchmark[3] with some test programs and measured the time to evaluate certain nondeterministic expressions to reduced normal form using breadth-first search or depth-first search, respectively. The test expressions were the following:

- subtracting two natural numbers in Peano representation[4] using a free variable and addition,
- dividing two Peano numbers using a free variable and multiplication,
- finding the last element of a 20-element list of boolean values, as shown in the introduction,
- sorting a four-element list of Peano numbers by trying all permutations,
- subtracting two primitive natural numbers using a logic variable and addition and
- multiplying two primitive natural numbers m, n in terms of subtraction: $mn = n + (m-1)n$.

For the last two test cases, evaluation was stopped as soon as the first solution was found. In the other cases, the evaluation tree had to be searched exhaustively.

To determine the execution times (wall-clock time), I used the benchmarking library `criterion`[5]. By running the benchmarks many times, it can accurately measure even very short running times. The program was compiled by GHC 7.8.3 with optimizations enabled (`-O2`). The results are shown in fig. 3.4.

As one can see, in most test cases, DFS and BFS take approximately the same time. However, in the last case, DFS is significantly faster. The reason for that is that the only successful "leaf" in the computation tree is at a certain place which is explored much earlier by DFS than by BFS. However, in other scenarios, the situation may be reversed since DFS can "get stuck" in a branch that does not yield a solution.

BFS has the theoretical disadvantage of worse space complexity than DFS. But in the examples I tested, this did not seem to be a problem. On the other hand, it has the advantage of finding all solutions even in infinitely deep trees. Therefore, it seems to be preferable as a default search strategy.

[3] a separate executable in the implementation, called `cumin-bench`

[4] A Peano number is either zero or represented as the successor of another Peano number: **data** Peano = Zero | Succ Peano.

[5] `http://http://hackage.haskell.org/package/criterion`

Chapter 4

Translating CuMin to SaLT

CuMin is convenient because nondeterminism is implicit. On the other hand, this makes it harder to analyze whether a function is actually deterministic. For that reason, SaLT was introduced in [11]. In SaLT, every expression that may assume multiple values must be set typed.

4.1 The Translation Rules

The translation method below is based on [11]. It had to be adapted because our versions of the languages CuMin and SaLT are more general: Constructors do not have to be fully applied, there are more ADTs than List, Bool and Pair, and the syntax for indexed unions is more general. As a consequence, one has to keep track of type information, in contrast to [11] where the transformation is purely syntactical.

The translation method is pessimistic insofar as it transforms every CuMin expression into a set-typed expression even if it is deterministic. This shortcoming will be partly addressed in section 4.2.

4.1.1 Translating Types

Every CuMin expression of type τ is translated to a SaLT expression of type $\{\lfloor\tau\rfloor\}$ where $\lfloor\cdot\rfloor$ inserts $\{\cdot\}$ to the right of every function arrow (fig. 4.1). For example, a CuMin expression f of type $(\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Nat}$ will be translated to a SaLT expression of type $\{(\mathsf{Nat} \to \{\mathsf{Bool}\}) \to \{\mathsf{Nat}\}\}$. The reason for the outer $\{\cdot\}$ is that f itself may be nondeterministic, i.e. it might represent multiple functions. The $\{\cdot\}$ braces in the argument are because f may be given a nondeterministic function as an argument. The remaining $\{\cdot\}$ is explained by the fact that f may compute more than one natural number.

4.1.2 Translating Data Declarations

In the same way as types, we have to translate data type declarations. Recall that an ADT declaration in CuMin looks like this.

$$\textbf{data}\ \mathsf{A}\ \alpha_1 \ldots \alpha_m = \mathsf{C}_1\ \tau_{11}\ \tau_{12} \cdots \mid \mathsf{C}_2\ \tau_{21}\ \tau_{22} \cdots \mid \ldots$$

Here, $\mathsf{C}_1, \mathsf{C}_2 \ldots$ are the constructors and $\tau_{11}, \tau_{12} \ldots$ are their argument types. It will be translated to the following SaLT ADT declaration.

$$\textbf{data}\ \mathsf{A}\ \alpha_1 \ldots \alpha_m = \mathsf{C}_1\ \lfloor\tau_{11}\rfloor\ \lfloor\tau_{12}\rfloor \cdots \mid \mathsf{C}_2\ \lfloor\tau_{21}\rfloor\ \lfloor\tau_{22}\rfloor \cdots \mid \ldots$$

$$\begin{aligned}
\lfloor \mathsf{Nat} \rfloor &= \mathsf{Nat} \\
\lfloor \mathsf{A}\ \tau_1 \dots \tau_n \rfloor &= \mathsf{A}\ \lfloor \tau_1 \rfloor \dots \lfloor \tau_n \rfloor \qquad \text{where } \mathsf{A} \text{ is the name of an ADT.} \\
\lfloor \alpha \rfloor &= \alpha \qquad \text{where } \alpha \text{ is a type variable.} \\
\lfloor \tau' \to \tau \rfloor &= \lfloor \tau' \rfloor \to \{\lfloor \tau \rfloor\}
\end{aligned}$$

Figure 4.1: Translation of types

As an example, consider difference lists.

$$\textbf{data}\ \mathsf{DList}\ \alpha = \mathsf{DList}\ (\mathsf{List}\ \alpha \to \mathsf{List}\ \alpha)$$

This is translated to the following SaLT declaration.

$$\textbf{data}\ \mathsf{DList}\ \alpha = \mathsf{DList}\ (\mathsf{List}\ \alpha \to \{\mathsf{List}\ \alpha\})$$

Such data structures are rather rare, however. Most of the time, the data declarations will contain no function types and the translation to SaLT will look the same.

4.1.3 Translating Expressions

How CuMin expressions are translated can be seen in fig. 4.2. The conversion function is denoted by $\lceil \cdot \rceil$. As mentioned before, an expression of type τ is translated to one of type $\{\lfloor \tau \rfloor\}$. This is achieved by adding sufficiently many $\{\cdot\}$ in the right places (cf. the first four rules). **unknown** already has $\{\cdot\}$-type in SaLT, so it does not need extra $\{\cdot\}$-braces.

The other translation rules handle expressions composed of subexpressions. They generally work by translating these subexpressions, "extracting" the elements using $\ggg=$ and acting on them. For example $1 + 1$ will be translated to

$$\{1\} \ggg= \lambda x :: \mathsf{Nat} \to \{1\} \ggg= \lambda y :: \mathsf{Nat} \to \{x + y\}$$

Needless to say, this translation is rather naive and not very efficient as it could simply be translated to $\{1 + 1\}$. We will address this problem later.

The rules in fig. 4.2 are taken from [11] with mostly small modifications because of the differences in syntax and generality of ADTs. However, the translation of constructors had to be generalized because in our version of CuMin, they are allowed to be partially applied. Therefore, they are translated similarly to regular CuMin functions, which are discussed in the next section, namely by wrapping each "level" in singleton sets.

4.1.4 Translating Function Declarations

The final step in translating CuMin programs to SaLT programs are function declarations. Remember that a function declaration in CuMin is given by

$$\begin{aligned}
&f :: \forall \alpha_1 \dots \alpha_m . \tau_1 \to \cdots \to \tau_n \to \tau \\
&f\ x_1 \dots x_n = e
\end{aligned}$$

where e denotes the expression on the right hand side of the function definition.

Such a function is translated to the following SaLT function.

$$
\begin{aligned}
\lceil x \rceil &= \{x\} \qquad \text{where } x \text{ is a variable}\\
\lceil n \rceil &= \{n\} \qquad \text{where } n \text{ is a literal}\\
\lceil \mathsf{C}_{\overline{\rho_m}} \rceil &= \{\lambda x_1 :: \lfloor \tau_1' \rfloor \to \ldots \{\lambda x_n :: \lfloor \tau_n' \rfloor \to \mathsf{C}_{\overline{\lfloor \rho_m \rfloor}}\; x_1 \ldots x_n\} \ldots\}\\
&\qquad \text{for every } \mathbf{data}\ \mathsf{A}\ \overline{\alpha_m} = \cdots \mid \mathsf{C}\ \tau_1 \ldots \tau_n \mid \ldots \text{ in CuMin}\\
&\qquad \text{and where } \tau_i' = \tau_i\ [\overline{\rho_m/\alpha_m}].\\
\lceil \mathbf{failed}_\tau \rceil &= \{\mathbf{failed}_{\lfloor \tau \rfloor}\}\\
\lceil \mathbf{let}\ x :: \tau\ \mathbf{free\ in}\ e \rceil &= \mathbf{unknown}_{\lfloor \tau \rfloor} \ggg\!= \lambda x :: \lfloor \tau \rfloor \to \lceil e \rceil\\
\lceil f_{\tau_1,\ldots,\tau_n} \rceil &= f_{\lfloor \tau_1 \rfloor,\ldots,\lfloor \tau_n \rfloor}\\
\lceil \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rceil &= \lceil e_1 \rceil \ggg\!= \lambda x :: \tau \to \lceil e_2 \rceil\\
&\qquad \text{where } \lceil e_1 \rceil :: \{\tau\}.\\
\lceil f\ e \rceil &= \lceil f \rceil \ggg\!= \lambda f' :: \tau_1 \to \lceil e \rceil \ggg\!= \lambda e' :: \tau_2 \to f'\ e'\\
&\qquad \text{where } \lceil f \rceil :: \{\tau_1\},\ \lceil e \rceil :: \{\tau_2\} \text{ and } f', e' \text{ are fresh variables.}\\
\lceil e_1 + e_2 \rceil &= \lceil e_1 \rceil \ggg\!= \lambda x_1 :: \mathsf{Nat} \to \lceil e_2 \rceil \ggg\!= \lambda x_2 :: \mathsf{Nat} \to \{x_1 + x_2\}\\
&\qquad \text{where } x_1 \text{ and } x_2 \text{ are fresh variables.}\\
\lceil e_1 \equiv e_2 \rceil &= \lceil e_1 \rceil \ggg\!= \lambda x_1 :: \tau \to \lceil e_2 \rceil \ggg\!= \lambda x_2 :: \tau \to \{x_1 \equiv x_2\}\\
&\qquad \text{where } \lceil e_i \rceil :: \{\tau\} \text{ and } x_i \text{ are fresh variables.}\\
\lceil \mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to e_1; \ldots\} \rceil &= \lceil e \rceil \ggg\!= \lambda x :: \tau \to \mathbf{case}\ x\ \mathbf{of}\ \{p_1 \to \lceil e_1 \rceil; \ldots\}\\
&\qquad \text{where } \lceil e \rceil :: \{\tau\} \text{ and } x \text{ is a fresh variable.}
\end{aligned}
$$

Figure 4.2: Translation rules for expressions

$$
\begin{aligned}
&f :: \forall \alpha_1 \ldots \alpha_m.\{\lfloor \tau_1 \to \cdots \to \tau_n \to \tau \rfloor\}\\
&f = \{\lambda x_1 :: \lfloor \tau_1 \rfloor \to \ldots \{\lambda x_n :: \lfloor \tau_n \rfloor \to \lceil e \rceil\} \ldots\}
\end{aligned}
$$

Note that we now have to use explicit lambda abstractions (which do not even exist in CuMin) because each (sub-)function needs to be wrapped in $\{\cdot\}$-braces.

4.1.5 Examples

Some example translations can be seen below. The original CuMin functions are on the left and their translated SaLT counterparts on the right-hand side.

$id :: \forall a.a \to a$ $id\ x = x$	$id :: \forall a.\{a \to \{a\}\}$ $id = \{\lambda x :: a \to \{x\}\}$
$choose :: \forall a.a \to a \to a$ $choose\ x\ y = \mathbf{let}\ c :: \mathsf{Bool}\ \mathbf{free\ in}$ $\quad \mathbf{case}\ c\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to y\}$	$choose :: \forall a.\{a \to \{a \to \{a\}\}\}$ $choose = \{\lambda x :: a \to \{\lambda y :: a \to$ $\quad \mathbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda c :: \mathsf{Bool} \to$ $\quad \mathbf{case}\ c\ \mathbf{of}\ \{\mathsf{True} \to \{x\}; \mathsf{False} \to \{y\}\}\}\}$

```
length :: ∀a.List a → Nat
length xs = case xs of
  Nil → 0
  Cons y ys → 1 + length_a ys
```

```
length :: ∀a.{List a → {Nat}}
length = {λxs :: List a → {xs} >>=
  λscrutinee :: List a → case scrutinee of
    Nil → {0}
    Cons y ys → {1} >>= λone :: Nat →
      (length_a >>= λlen :: (List a → {Nat}) →
      {ys} >>= λys' :: List a → len ys') >>=
      λl :: Nat → {one + l}
  }
```

4.2 Improving the Generated SaLT Code

As one can see in the example programs, the translated expressions are often unnecessarily set-typed, requiring a lot of “plumbing” with $\{\cdot\}$ and $\gg\!=$. However, there are some simple transformations that can be used to make the SaLT code much more efficient.

One transformation that is not directly useful but helps a lot with other simplifications is β-reduction: Given an expression of the form $(\lambda x \to e_1)\ e_2$, one can turn it into $e_1\ [e_2/x]$. However, β-reducing is not always beneficial: Substituting the expression e_2 into e_1 can lead to wasteful re-computation if x occurs in e_1 more than once. Hence, this simplification should only be used when the bound variable occurs at most once.[1] Note that variable capture on substitution is an issue, as well. It is addressed in section 4.3.

Similarly to β-reduction, there is η-reduction: An expression of the form $\lambda x \to f\ x$ is equivalent to f if x does not occur freely in f. In contrast to β-reduction, this transformation is always safe and beneficial. Note that while η-reduction is not valid for CuMin (cf. section 3.1), it is allowed in SaLT because there is no implicit nondeterminism or call-time choice.

It was mentioned before that the set type constructor $\{\cdot\}$ forms a monad, in particular, it obeys the *monad laws* listed below. To give some intuition, I also state the laws using set notation.

1. $(\{e\} \gg\!= f)\ \cong\ (f\ e)$ or, in set notation $\bigcup_{x\in\{e\}}(f\ x) \cong (f\ e)$
2. $(e \gg\!= \lambda x \to \{x\})\ \cong\ e$ or, in set notation $\bigcup_{x\in e}\{x\} \cong e$
3. $(e \gg\!= f) \gg\!= g\ \cong\ e \gg\!= (\lambda x \to f\ x \gg\!= g)$ or $\bigcup_{y\in(\bigcup_{x\in e}(f\ x))}(g\ y) \cong \bigcup_{x\in e}\left(\bigcup_{y\in(f\ x)}(g\ y)\right)$

The symbol $\cong$ denotes *semantic equivalence*, which means that two expressions evaluate to the same result, and is formally defined in [11]. For lack of space, I cannot develop this theoretical background, which is necessary to prove these laws.

The first monad law, viewed as a transformation from left to right, is extremely useful for simplifying the translated SaLT programs. CuMin literals and variables are translated to singleton sets, which makes this law applicable in many cases. Afterwards, β-reduction can be applied as a next step. As an example, consider the term $1 + 1$. Its translated version can be simplified using the first monad law twice and then β-reducing twice:

$$\begin{aligned}
&\{1\} \gg\!= \lambda x :: \mathsf{Nat} \to \{1\} \gg\!= \lambda y :: \mathsf{Nat} \to \{x + y\}\\
\cong\ &(\lambda x :: \mathsf{Nat} \to \{1\} \gg\!= \lambda y :: \mathsf{Nat} \to \{x + y\})\ 1\\
\cong\ &(\lambda x :: \mathsf{Nat} \to (\lambda y :: \mathsf{Nat} \to \{x + y\})\ 1)\ 1\\
\cong\ &(\lambda y :: \mathsf{Nat} \to \{1 + y\})\ 1\\
\cong\ &\{1 + 1\}
\end{aligned}$$

[1] There are actually more cases where this is safe, for example, if the variable occurs only once in each branch of a case expression. However, there is still some code being duplicated, which may increase program size considerably. To keep things simple, I did not explore that further.

The second monad law is not very useful, it can only be applied in case of unnecessary let-bindings like **let** $x = e$ **in** x.

The utility of the third monad law is not immediately obvious. However, it can be used to "re-associate" $\ggg=$-bindings, thus enabling the application of the first rule in some cases. For instance, consider the expression $(x \ggg= \lambda y \to \{f\}) \ggg= \lambda g \to g\ y$. At first sight, neither the first nor the second law can be applied. However, the third monad law allows us to transform this into $x \ggg= \lambda y \to (\{f\} \ggg= \lambda g \to g\ y)$. Now, the first monad law is applicable and yields $x \ggg= \lambda y \to f\ y$ after β-reduction. Using η-equivalence, we arrive at $x \ggg= f$, as desired.

This is not a hypothetical scenario but happens in real translated SaLT programs. For instance, consider the CuMin expression $\mathsf{Cons}_{\mathsf{Nat}}\ coin\ \mathsf{Nil}_{\mathsf{Nat}}$. Translating this to SaLT and applying the first monad law yields:

$$\begin{aligned}
&\lceil \mathsf{Cons}_{\mathsf{Nat}}\ coin\ \mathsf{Nil}_{\mathsf{Nat}} \rceil \\
&\quad\cong\ (coin \ggg= \lambda c :: \mathsf{Nat} \to \{\lambda xs :: \mathsf{List\ Nat} \to \{\mathsf{Cons}_{\mathsf{Nat}}\ c\ xs\}\}) \\
&\qquad \ggg= \lambda f :: (\mathsf{List\ Nat} \to \{\mathsf{List\ Nat}\}) \to f\ \mathsf{Nil}_{\mathsf{Nat}}
\end{aligned}$$

Applying the third monad law enables the first monad law and β-reduction:

$$\begin{aligned}
&coin \ggg= \lambda c :: \mathsf{Nat} \to \\
&\quad (\{\lambda xs :: \mathsf{List\ Nat} \to \{\mathsf{Cons}_{\mathsf{Nat}}\ c\ xs\}\} \\
&\qquad \ggg= \lambda f :: (\mathsf{List\ Nat} \to \{\mathsf{List\ Nat}\}) \to f\ \mathsf{Nil}_{\mathsf{Nat}}) \\
&\cong\ coin \ggg= \lambda c :: \mathsf{Nat} \to \\
&\quad (\lambda xs :: \mathsf{List\ Nat} \to \{\mathsf{Cons}_{\mathsf{Nat}}\ c\ xs\})\ \mathsf{Nil}_{\mathsf{Nat}} \\
&\cong\ coin \ggg= \lambda c :: \mathsf{Nat} \to \{\mathsf{Cons}_{\mathsf{Nat}}\ c\ \mathsf{Nil}_{\mathsf{Nat}}\}
\end{aligned}$$

In general, we can limit ourselves to only ever applying the third monad law from left to right. This is because the first monad law can be used much more often than the second one, and it benefits only from this direction. In the example programs I looked at, applying the third monad law from right to left was never useful.

As a larger example, let us look at how the simplifications transform the prelude function *length*. The original version is on the left, the simplified one on the right.

```
length :: ∀a.{List a → {Nat}}
length = {λxs :: List a → {xs} >>=
  λscrutinee :: List a → case scrutinee of
    Nil → {0}
    Cons y ys → {1} >>= λone :: Nat →
      (length_a >>= λlen :: (List a → {Nat}) →
      {ys} >>= λys' :: List a → len ys') >>=
      λl :: Nat → {one + l}
  }
```

```
length :: ∀a.{List a → {Nat}}
length = {λxs :: List a → case xs of
  Nil → 0
  Cons y ys → length_a >>=
    λlength' :: (List a → {Nat}) →
    length' ys >>= λl :: Nat → {1 + l}}
```

This is a considerable improvement. There is only one thing that could be done better in a manual translation, by exploiting the fact that the function is actually deterministic, which is discussed in section 5.5 by means of the same example.

4.3 Implementation

The implementation is relatively close to the translation and simplification method described above. The program recursively traverses the syntax tree and applies these rules. In the following, I will explain some implementation details. An overview of the implementation is given afterwards.

While the translation given in [11] is purely syntactical, the adapted version presented here requires type information. For example, let bindings do not have a type annotation but they are translated to lambda abstractions, which need one. As a consequence, one has to keep track of the types of bound variables while traversing the syntax tree. Another problem is fresh variables and capture avoidance which was briefly discussed in section 2.5.1. Fresh variables could be generated as before, by appending a unique number. Variable capture is a real problem, however: β-reducing $(\lambda z \to (\lambda y \to z))\ (\lambda x \to y)$ by blind substitution yields $\lambda y \to (\lambda x \to y)$ which is incorrect, since the variable y is not free anymore. This could be solved by examining the variables of the substituted expression and renaming them, if necessary. This is rather complicated and relatively hard to get right.

I chose the following different solution: I used a nameless representation of terms, where variables are not identified by names but by "how many levels up the syntax tree" they were bound. This is made precise below.

4.3.1 Nameless Representation

To handle bound variables, I used the `bound` library[2] by Edward Kmett. It has the following variable type:

$$\textbf{data}\ \mathsf{Var}\ b\ a = \mathsf{B}\ b \mid \mathsf{F}\ a$$

$\mathsf{B}\ b$ represents a variable bound directly by the next binding up the syntax tree. $\mathsf{F}\ a$ represents a free variable, it is not directly bound by the next parent binding. But it may be bound at higher levels in the syntax tree.

For illustration purposes, consider the following data type for lambda abstractions:

$$\textbf{data}\ \mathsf{Exp}\ v = \mathsf{Var}\ v \mid \mathsf{Lam}\ (\mathsf{Scope}\ ()\ \mathsf{Exp}\ v)$$

Here, v is the variable type of the expression. Scope is provided by the `bound` library and represents a binder. The general form is

$$\textbf{data}\ \mathsf{Scope}\ b\ f\ a = \mathsf{Scope}\ (f\ (\mathsf{Var}\ b\ (f\ a)))$$

where b represents additional information for bound variables, in this case none, represented by the unit type (); f represents the expression type; and a stands for the type of free variables, not bound in this scope. For example, the expression $\lambda x \to \lambda y \to x$ would be represented as

$$\mathsf{Lam}\ (\mathsf{Scope}\ (\mathsf{Lam}\ (\mathsf{Scope}\ (\mathsf{Var}\ (\mathsf{F}\ (\mathsf{Var}\ (\mathsf{B}\ ()))))))))$$

where F "lifts" the bound variable B () one level up, so it is bound by the outer lambda abstraction instead of the inner one. For lack of space, I cannot give a longer introduction to the library. But I want to highlight some of its advantages.

First of all, variable capture is not a problem anymore, and substitution is for the most part handled in the library. Additionally, there is a lot more type safety than in a representation with names: For example, a term without free variables can be given the type Exp Void where Void is an empty type. In fact, this is what I do for SaLT functions in the program, since they have to be closed terms. (Calling top-level functions is not represented as a free variable.) Furthermore, lots of mistakes when handling variables can be caught at compile time. The reason is that many bugs, such as forgetting a binder, lead to type errors in the Haskell code because the variable types do not match up.

[2] `http://hackage.haskell.org/package/bound`

4.3.2 General Approach

I implemented the translation method as a program called `cumin2salt`. On execution, it is passed a CuMin program to translate, and a flag indicating whether the result should be simplified. My implementation proceeds as follows.

The CuMin program is parsed and type checked. If there were no errors, it is translated to the nameless representation. This one, in turn, is transformed to a nameless SaLT representation, following the translation rules. If desired by the user, the generated SaLT code is simplified using β-reduction, η-reduction and the monad laws. This nameless SaLT AST is then translated to a regular SaLT AST, by giving names to the bound variables. In order to guarantee uniqueness, a different number is appended to each one. Note that only one renaming pass over the AST is necessary, everything before is handled by the nameless representation. Also, the original variable names are still retained if possible to make the output more readable. Finally, this SaLT AST is pretty-printed and written to a file.

4.3.3 Command Line Interface

The command line arguments to the program look like this:

```
cumin2salt INPUT [-o OUTPUT] [-s|--simplify] [--with-prelude]
```

The program is given a CuMin input file name and, possibly, a SaLT output file name (default: `Out.salt`). If the SaLT output should be simplified, one should pass the option `-s`. The switch `--with-prelude` controls whether the prelude functions should be part of the output. Normally, the translated prelude functions need not be included, as they are provided by the alternative SaLT prelude (cf. section 2.4). Of course, `--help` can be used to show a help text.

4.3.4 Example

As an exemplary output, consider the following automatic translation of the *length* function, with simplifications applied.

$$\begin{array}{l}
\mathit{length} :: \forall a. \{\mathsf{List}\ a \to \{\mathsf{Nat}\}\} \\
\mathit{length} = \{\lambda \mathit{xxs\_56} :: \mathsf{List}\ a \to \textbf{case}\ \mathit{xxs\_56}\ \textbf{of} \\
\quad \mathsf{Nil} \to \{0\} \\
\quad \mathsf{Cons}\ \mathit{x\_57}\ \mathit{xs\_58} \to \mathit{length}_a \gg\!= (\lambda \mathit{arg\_59} :: (\mathsf{List}\ a \\
\qquad \to \{\mathsf{Nat}\}) \to \mathit{arg\_59}\ \mathit{xs\_58} \gg\!= \\
\qquad (\lambda \mathit{primOpArg\_60} :: \mathsf{Nat} \to \{1 + \mathit{primOpArg\_60}\}))\}
\end{array}$$

As one can see, numbers are appended to variable names to make them unique. Apart from that, this output matches the manual translation seen above.

4.3.5 Correctness

To ensure correctness of the translation program, I took the following measures: After every simplification, it is checked that the type did not change. Additionally, after the whole translation, it is checked that the SaLT functions have the right types, namely that a CuMin function of type τ is translated to a SaLT function of type $\{\lfloor\tau\rfloor\}$. This is a useful consistency check and catches a large class of bugs.

Testing is still necessary, of course. The implementation of denotational semantics for CuMin and SaLT by Fabian Thorand provides a good way to do this. A CuMin expression in the context of some program must have the same semantics as the translated expression in context

Program	unoptimized	optimized
Peano addition	23 ms	10 ms
Peano subtraction	220 ms	180 ms
Peano division	570 ms	270 ms
Last	96 ms	47 ms
Permutation sort	44 ms	15 ms

Figure 4.3: Average running times for SaLT code with and without simplifications

of the translated program. To verify that, I created a test suite in the implementation, called `trans-test`, which checks this equivalence for a number of test expressions, and both for the original and for the simplified SaLT code.

Together, type checking and testing using the denotational semantics strengthen the claim that the translation preserves the semantics and thus works as intended.

4.3.6 Simplifications and Performance

While the main reason for studying the translation to SaLT and the simplifications was to analyze the non-determinism in a CuMin program (next chapter), it is still interesting to measure the effects of the simplifications on the performance of SaLT programs.

I created a benchmark[3] using the same infrastructure as in section 3.6. The setup was the following: I wrote a CuMin program with some expressions to benchmark, which was then translated to SaLT using my implementation, both with and without simplifications. In each version, the test expressions were evaluated to reduced normal form, using the implementation of the semantics for SaLT by Fabian Thorand. Only the first fully evaluated result was computed with BFS. The benchmarks were the following:

- adding two natural numbers in Peano representation, which is completely deterministic,
- subtracting two Peano numbers, using a logic variable,
- dividing two Peano numbers, using a logic variable,
- computing the last element of a seven-element list of booleans, using logic variables, and
- sorting a four-element list of Peano numbers by trying all permutations.

The results are shown in fig. 4.3. As expected, the simplifications speed up the execution of SaLT programs significantly, on average by a factor of two. In the case of permutation sort, the optimized version is even three times as fast as the unoptimized one.

[3] a separate executable in the implementation, called `opt-bench`

Chapter 5

Nondeterminism Analysis

The original motivation for introducing the language SaLT and its translation in [11] was to derive free theorems for CuMin. In order to do this, one has to restrict the nondeterministic behavior of the functions involved. SaLT serves the purpose of making nondeterminism explicit so that one can analyze it properly. This chapter is about how exactly this can be done.

5.1 Introduction

It was mentioned before that a SaLT function whose signature contains no set types will be completely deterministic because all nondeterminism is marked with $\{\cdot\}$. On the other hand, the translation routine from the previous chapter always produces set-typed expressions even if they are in fact deterministic. Set-typed functions are deterministic if they only return singleton sets. As a trivial example, consider the translated identity function.

$$
\begin{aligned}
&id :: \forall a.\{a \to \{a\}\}\\
&id = \{\lambda x \to \{x\}\}
\end{aligned}
$$

It is a singleton set containing a function returning a singleton set, so the original CuMin function *id* is completely deterministic.

In general, determining whether a function is deterministic is undecidable. This can be proven by reduction from the halting problem.[1] Consider the following SaLT expression.

$$\textbf{case}\ \mathit{condition}\ \textbf{of}\ \{\mathsf{True} \to \{\mathsf{True}\}; \mathsf{False} \to \textbf{unknown}_{\mathsf{Bool}}\}$$

This is deterministic if and only if always the first branch is evaluated. However, we cannot decide whether *condition* is always True since it is even undecidable whether its evaluation terminates at all. So any method of detecting determinism will not be complete. However, one can usually accomplish a lot with purely syntactic transformations, as I will describe below.

In what follows, I will not always strictly adhere to the SaLT syntax in order to keep the code readable. For example, I will omit type instantiations if they are clear from the context and I will allow additional infix operators to be defined. Moreover, I will sometimes write SaLT function definitions in an equational style instead of using explicit lambda abstractions.

The concept of *semantic equivalence*, denoted by $\cong$, will be used extensively in this chapter. It means that two expressions give the same results on evaluation. A formal definition is given in [11] and depends on the denotational semantics defined in the same paper. As there is no space

[1] As SaLT includes a simply-typed lambda calculus and allows unrestricted recursion, it is Turing complete.

to develop this semantics here, the reader will have to believe some claims I make about certain semantic equivalences. Most of the time, however, equational reasoning, e.g. with β-reduction or monad laws for sets, will suffice to show the equivalences we are interested in.

5.2 Useful Combinators

In the rest of this chapter, the following SaLT combinators will be useful.

$$
\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c)\\
&(\circ)\ f\ g\ x = f\ (g\ x)\\
&\{\cdot\} :: a \to \{a\}\\
&\{\cdot\}\ x = \{x\}\\
&\mathit{sMap} :: (a \to b) \to \{a\} \to \{b\}\\
&\mathit{sMap}\ f\ s = s \ggg\!= \{\cdot\} \circ f
\end{aligned}
$$

The operator $(\circ)$ is simply function composition. I will often use the section notation for $(\circ)$ as in Haskell: $(f\circ)$ stands for $\lambda g \to f \circ g$ and $(\circ f)$ stands for $\lambda g \to g \circ f$. The function *sMap* has been mentioned earlier already. It applies a function to each element of a set. The combinator $\{\cdot\}$ places an expression in a singleton set.

Similarly to the monad laws for sets, the following laws hold for *sMap*.

1. $\mathit{sMap}\ f\ \{e\}\ \cong\ \{f\ e\}$
2. $\mathit{sMap}\ f\ s \ggg\!= g\ \cong\ s \ggg\!= (g \circ f)$
3. $s \ggg\!= (\mathit{sMap}\ f \circ g)\ \cong\ \mathit{sMap}\ f\ (s \ggg\!= g)$

The proofs employ the monad laws.

$$
\begin{aligned}
&\mathit{sMap}\ f\ \{e\}\\
&\ \cong \quad \text{-- definition of } \mathit{sMap}\\
&\{e\} \ggg\!= \{\cdot\} \circ f\\
&\ \cong \quad \text{-- first monad law}\\
&(\{\cdot\} \circ f)\ e\\
&\ \cong \quad \text{-- definition of } (\circ) \text{ and } \{\cdot\}\\
&\{f\ e\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sMap}\ f\ s \ggg\!= g\\
&\ \cong \quad \text{-- definition of } \mathit{sMap}\\
&(s \ggg\!= \{\cdot\} \circ f) \ggg\!= g\\
&\ \cong \quad \text{-- third monad law}\\
&s \ggg\!= (\lambda x \to \{f\ x\} \ggg\!= g)\\
&\ \cong \quad \text{-- first monad law}\\
&s \ggg\!= \lambda x \to g\ (f\ x)\\
&\ \cong \quad \text{-- definition of } (\circ)\\
&s \ggg\!= (g \circ f)
\end{aligned}
$$

$$
\begin{aligned}
&s \ggg\!= (\mathit{sMap}\ f \circ g)\\
&\ \cong \quad \text{-- definition of } (\circ)\\
&s \ggg\!= (\lambda x \to \mathit{sMap}\ f\ (g\ x))
\end{aligned}
$$

$\cong$ -- definition of *sMap*
$s \gg\!= (\lambda x \to g\ x \gg\!= (\{\cdot\} \circ f))$
$\cong$ -- third monad law
$(s \gg\!= g) \gg\!= (\{\cdot\} \circ f)$
$\cong$ -- definition of *sMap*
$sMap\ f\ (s \gg\!= g)$

Another concept to discuss is *strictness*. A SaLT function $f :: \tau' \to \tau$ is called strict if f $\mathbf{failed}_{\tau'} \cong \mathbf{failed}_{\tau}$. Intuitively, this means that f evaluates its argument, for example, using a case expression. Some transformations in the following only work for strict functions. The functions $\{\cdot\}$, $(f\circ)$ and $sMap\ f$ are strict if f is strict, according to the semantics from [11].

5.3 Determinism

First of all, we have to define what determinism means. We will first do that for expressions that are not functions: If τ is not a function type, a CuMin expression $e :: \tau$ is called *deterministic* if there is a SaLT expression $e' :: \lfloor\tau\rfloor$ such that $\lceil e \rceil \cong \{e'\}$. Such an e' is called a *witness*. This means that e is equivalent to a singleton set in SaLT. Note that according to the semantics from [11], $\mathbf{failed}_{\{\tau\}} \cong \{\mathbf{failed}_{\tau}\}$ holds. Hence, a failed computation can also be regarded as a singleton set.

Transferring this to the operational semantics, it means that there is at most one successful evaluation for e. In particular, the evaluation of e is allowed to fail. A sufficient (not necessary) condition for at most one evaluation is that the Guess rule from the operational semantics is never used. This is essentially what we will try to show when proving an expression deterministic in the following. The condition is not necessary since evaluating $last_{\mathsf{Bool}}$ [True], with the function *last* from the introduction, uses the Guess rule due to logic variables, but the expression still has only one result: True.

How can one derive that an expression is deterministic without evaluating it? It can often be achieved by inlining top-level functions (replacing them by the right-hand side of their definition) and applying the monad laws for sets.

For instance, consider the CuMin expression *double* (*double* 1). To show that it is deterministic, one translates the expression to SaLT, simplifies the result and inlines functions to apply monad laws.

$\lceil double\ (double\ 1)\rceil$
$\cong$ -- translation and simplification
$double \gg\!= \lambda f \to (double \gg\!= \lambda g \to g\ 1) \gg\!= f$
$\cong$ -- inlining *double*
$\{\lambda x \to \{x + x\}\} \gg\!= \lambda f \to (\{\lambda x \to \{x + x\}\} \gg\!= \lambda g \to g\ 1) \gg\!= f$
$\cong$ -- first monad law
$\{\lambda x \to \{x + x\}\} \gg\!= \lambda f \to ((\lambda x \to \{x + x\})\ 1) \gg\!= f$
$\cong$ -- β-reduction
$\{\lambda x \to \{x + x\}\} \gg\!= \lambda f \to \{1 + 1\} \gg\!= f$
$\cong$ -- first monad law and β-reduction
$\{1 + 1\} \gg\!= \lambda x \to \{x + x\}$
$\cong$ -- first monad law and β-reduction
$\{(1 + 1) + (1 + 1)\}$

This is a singleton set, so the original expression is deterministic. Note that we did not have to evaluate the expression completely to recognize that.

As another example, consider the CuMin expression $\mathit{guard}_{\mathsf{Nat}}\ \mathit{cond}\ 1$ where cond is some deterministic expression and the guard function returns its second argument if and only if the first argument is True:

$$\begin{aligned}&\mathit{guard} :: \forall a.\mathsf{Bool} \to a \to a\\ &\mathit{guard}\ \mathit{cond}\ x = \mathbf{case}\ \mathit{cond}\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to \mathbf{failed}_a\ \}\end{aligned}$$

For this instance, another rule is useful: In a case expression where each branch is a singleton set, this singleton set can be extracted and moved out of the case expression:

$$\begin{aligned}&\mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to \{e_1\}; \ldots; p_n \to \{e_n\}\}\\ &\ \cong\ \{\mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\}\}\end{aligned}$$

Even more generally, the following transformation is valid for strict functions $h :: \tau' \to \{\tau\}$.

$$\begin{aligned}&\mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to h\ e_1; \ldots; p_n \to h\ e_n\}\\ &\ \cong\ h\ \mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\}\end{aligned}$$

Strictness of h is necessary because, if e fails, so will the case expression. For the equivalence to hold in this case, h also has to fail. Again, a formal proof requires the semantics from [11].

Using this additional rule, one can show that $\mathit{guard}_{\mathsf{Nat}}\ \mathit{cond}\ 1$ is equivalent to a singleton set.

$$\begin{aligned}&\lceil \mathit{guard}_{\mathsf{Nat}}\ \mathit{cond}\ 1\rceil\\
&\ \cong\quad \text{-- translation and simplification}\\
&\mathit{guard}_{\mathsf{Nat}} \ggg\!= \lambda g \to \lceil \mathit{cond}\rceil \ggg\!= \lambda c \to g\ c \ggg\!= \lambda g' \to g'\ 1\\
&\ \cong\quad \text{-- assumption: } \mathit{cond} \text{ is deterministic}\\
&\mathit{guard}_{\mathsf{Nat}} \ggg\!= \lambda g \to \{\mathit{cond}'\} \ggg\!= \lambda c \to g\ c \ggg\!= \lambda g' \to g'\ 1\\
&\ \cong\quad \text{-- first monad law, } \beta\text{-reduction}\\
&\mathit{guard}_{\mathsf{Nat}} \ggg\!= \lambda g \to g\ \mathit{cond}' \ggg\!= \lambda g' \to g'\ 1\\
&\ \cong\quad \text{-- inlining } \mathit{guard}\\
&\{\lambda \mathit{cond} \to \{\lambda x \to \mathbf{case}\ \mathit{cond}\ \mathbf{of}\ \{\mathsf{True} \to \{x\}; \mathsf{False} \to \{\mathbf{failed}_{\mathsf{Nat}}\ \}\}\}\}\\
&\quad \ggg\!= \lambda g \to g\ \mathit{cond}' \ggg\!= \lambda g' \to g'\ 1\\
&\ \cong\quad \text{-- case rule}\\
&\{\lambda \mathit{cond} \to \{\lambda x \to \{\mathbf{case}\ \mathit{cond}\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to \mathbf{failed}_{\mathsf{Nat}}\ \}\}\}\}\\
&\quad \ggg\!= \lambda g \to g\ \mathit{cond}' \ggg\!= \lambda g' \to g'\ 1\\
&\ \cong\quad \text{-- first monad law, } \beta\text{-reduction}\\
&\{\lambda x \to \{\mathbf{case}\ \mathit{cond}'\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to \mathbf{failed}_{\mathsf{Nat}}\ \}\}\} \ggg\!= \lambda g' \to g'\ 1\\
&\ \cong\quad \text{-- first monad law, } \beta\text{-reduction}\\
&\{\mathbf{case}\ \mathit{cond}'\ \mathbf{of}\ \{\mathsf{True} \to 1; \mathsf{False} \to \mathbf{failed}_{\mathsf{Nat}}\ \}\}\end{aligned}$$

This is again a singleton set, i.e. under the assumption that cond is deterministic, so is $\mathit{guard}_{\mathsf{Nat}}\ \mathit{cond}\ 1$.

As the next step, let us define determinism for functions. A CuMin expression $f :: \tau_1 \to \tau_2$ is called *deterministic* if there is a SaLT function $f' :: \lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor$ such that $\lceil f\rceil \ \cong\ \{\{\cdot\} \circ f'\}$. (Recall that $\lceil f\rceil :: \{\tau_1 \to \{\tau_2\}\}$.) Again, f' is called a *witness*. Not only does this mean that $\lceil f\rceil$ is a singleton set, but this also implies that applying this single function to any SaLT expression of type $\lfloor\tau_1\rfloor$ yields a singleton set.

Operationally speaking, this means that there is at most one successful evaluation when applying f to any deterministic expression of type τ_1.

Although the methods are the same as for deterministic expressions, let us look at how functions can be proven deterministic. Consider the CuMin function *double* as an example. One can see that in this case, a witness is given by $\lambda x :: \mathsf{Nat} \to x + x$.

$\lceil double \rceil$
$\quad\cong\quad$ -- translating to SaLT and simplifying
$double$
$\quad\cong\quad$ -- inlining *double*
$\{\,\lambda x :: \mathsf{Nat} \to \{\,x + x\,\}\,\}$
$\quad\cong\quad$ -- rewriting using $\{\cdot\}$ (β-expansion)
$\{\,\{\cdot\} \circ (\lambda x :: \mathsf{Nat} \to x + x)\,\}$

5.4 Multideterminism

We will now consider a weaker notion of determinism, given in [11]. A CuMin expression $f :: \tau_1 \to \tau_2$ is called *multideterministic* if there is a set of SaLT functions $f' :: \{\lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor\}$ such that $\lceil f\rceil \cong sMap\ (\lambda g \to \{\cdot\} \circ g)\ f'$. Again, such an f' is called a *witness.*

Intuitively, this means that the set on the inner level of $\lceil f\rceil :: \{\lfloor\tau_1\rfloor \to \{\lfloor\tau_2\rfloor\}\}$ are actually unnecessary, i.e. $\lceil f\rceil$ represents a set of functions returning singleton sets. The motivation for this definition in [11] is the derivation of free theorems, which requires the inner level of nondeterminism to be restricted.

Operationally, one can think of multideterminism as follows. When applying f to a deterministic CuMin expression e, f has to be in flat normal form at some point. The Apply rule may be used to achieve this. Since f is not required to be deterministic, there may be more than one flat normal form since f is represented by a set of functions in SaLT. However, once a flat normal form is obtained, there must be at most one evaluation result for the function application.

The definition and properties of *sMap* from above are useful in order to prove that functions are multideterministic. To give a concrete example, let us show that *maybeDouble1* from section 3.1 is multideterministic. Recall the CuMin definitions:

$maybeDouble1 :: \mathsf{Nat} \to \mathsf{Nat}$
$maybeDouble1 = choose_{\mathsf{Nat}\to\mathsf{Nat}}\ id_{\mathsf{Nat}}\ double$

$choose :: \forall a.a \to a \to a$
$choose\ x\ y = \mathbf{let}\ b :: \mathsf{Bool}\ \mathbf{free\ in\ case}\ b\ \mathbf{of}\ \{\mathsf{True} \to x; \mathsf{False} \to y\}$

Their SaLT translations look like this after simplification:

$maybeDouble1 :: \{\mathsf{Nat} \to \{\mathsf{Nat}\}\}$
$maybeDouble1 = choose_{\mathsf{Nat}\to\{\mathsf{Nat}\}} \gg\!= \lambda c \to$
$\quad id_{\mathsf{Nat}\to\{\mathsf{Nat}\}} \gg\!= \lambda i \to$
$\quad c\ i \gg\!= \lambda cc \to$
$\quad double \gg\!= \lambda d \to$
$\quad cc\ d$
$choose :: \forall a.\{a \to \{a \to \{a\}\}\}$
$choose = \{\,\lambda x \to \{\,\lambda y \to \mathbf{unknown}_{\mathsf{Bool}} \gg\!= \lambda c \to$
$\quad \mathbf{case}\ c\ \mathbf{of}\ \{\mathsf{True} \to \{\,x\,\}; \mathsf{False} \to \{\,y\,\}\}\}\}$

We now want to analyze the SaLT function *maybeDouble1* and try to find a witness for its multideterminism. First, we inline *choose* and simplify.

$maybeDouble1 = id_{\mathsf{Nat}} \gg\!= \lambda i \to$
$\quad double \gg\!= \lambda d \to$
$\quad \mathbf{unknown}_{\mathsf{Bool}} \gg\!= \lambda b \to$
$\quad \mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to \{\,i\,\}; \mathsf{False} \to \{\,d\,\}\}$

Inlining *id* and *double*, followed by simplifications with monad laws, yields:

$$
\begin{array}{l}
\textbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda c \rightarrow \\
\quad \textbf{case}\ c\ \textbf{of}\ \{\mathsf{True} \rightarrow \{\lambda x \rightarrow \{x\}\}; \mathsf{False} \rightarrow \{\lambda x \rightarrow \{x + x\}\}\} \\
\cong \quad \text{-- since } \mathit{sMap}\ f\ \{e\} = \{f\ e\} \\
\textbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda c \rightarrow \\
\quad \textbf{case}\ c\ \textbf{of}\ \{\mathsf{True} \rightarrow \mathit{sMap}\ (\{\cdot\}\circ)\ \{\lambda x \rightarrow x\}; \mathsf{False} \rightarrow \mathit{sMap}\ (\{\cdot\}\circ)\ \{\lambda x \rightarrow x + x\}\} \\
\cong \quad \text{-- case rule} \\
\textbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda c \rightarrow \\
\quad \mathit{sMap}\ (\{\cdot\}\circ)\ (\textbf{case}\ c\ \textbf{of}\ \{\mathsf{True} \rightarrow \{\lambda x \rightarrow x\}; \mathsf{False} \rightarrow \{\lambda x \rightarrow x + x\}\}) \\
\cong \quad \text{-- rewrite with } (\circ) \\
\textbf{unknown}_{\mathsf{Bool}} \ggg\!= (\mathit{sMap}\ (\{\cdot\}\circ) \circ (\lambda c \rightarrow \\
\quad \textbf{case}\ c\ \textbf{of}\ \{\mathsf{True} \rightarrow \{\lambda x \rightarrow x\}; \mathsf{False} \rightarrow \{\lambda x \rightarrow x + x\}\})) \\
\cong \quad \text{-- since } s \ggg\!= (\mathit{sMap}\ f \circ g) \ \cong\ \mathit{sMap}\ f\ (s \ggg\!= g) \\
\mathit{sMap}\ (\{\cdot\}\circ) \\
\quad (\textbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda c \rightarrow \textbf{case}\ c\ \textbf{of}\ \{\mathsf{True} \rightarrow \{\lambda x \rightarrow x\}; \mathsf{False} \rightarrow \{\lambda x \rightarrow x + x\}\})
\end{array}
$$

For the case rule, we used that *sMap* $(\{\cdot\}\circ)$ is strict, which follows from the semantics in [11]. The witness $\textbf{unknown}_{\mathsf{Bool}} \ggg\!= \lambda c \rightarrow \textbf{case}\ c\ \textbf{of}\ \{\ldots\}$ represents the set $\{\lambda x \rightarrow x, \lambda x \rightarrow x + x\}$ of two deterministic functions. This shows that *maybeDouble1* is multideterministic. On the other hand, such a proof fails for *maybeDouble2*, which is another illustration of their discrepancy.

5.5 Recursive Definitions

From what we discussed before, it is not clear how to handle recursion. As an illustration, consider the infinite list:

$$
\begin{array}{l}
\mathit{ones} :: \mathsf{List\ Nat} \\
\mathit{ones} = \mathsf{Cons}_{\mathsf{Nat}}\ 1\ \mathit{ones}
\end{array}
$$

Intuitively, it is clear that this is deterministic but our methods from before (especially inlining) will not work. Let us look at the SaLT code.

$$
\begin{array}{l}
\mathit{ones} :: \{\mathsf{List\ Nat}\} \\
\mathit{ones} = \mathit{ones} \ggg\!= \lambda x \rightarrow \{\mathsf{Cons}_{\mathsf{Nat}}\ 1\ x\}
\end{array}
$$

At first sight, it seems that there is no way to "extract" any singleton sets from this. However, I claim that it is equivalent to the following.

$$
\begin{array}{l}
\mathit{ones}' :: \mathsf{List\ Nat} \\
\mathit{ones}' = \mathsf{Cons}_{\mathsf{Nat}}\ 1\ \mathit{ones}' \\
\mathit{ones} :: \{\mathsf{List\ Nat}\} \\
\mathit{ones} = \{\mathit{ones}'\}
\end{array}
$$

To justify this transformation, consider the following more general scenario. We have a recursive, set-typed function definition f.

$$
\begin{array}{l}
f :: \{\tau\} \\
f = e
\end{array}
$$

The definition of f can be rewritten as $f = g\ f$ for some non-recursive function $g :: \{\tau\} \rightarrow \{\tau\}$, namely $g := \lambda x \rightarrow e'$ where x is a fresh variable and e' is the expression e with every occurrence of f replaced by x. Moreover, suppose we know that $g \circ h \;\cong\; h \circ g'$ for some $g' :: \tau' \to \tau'$ and $h :: \tau' \to \{\tau\}$ where h is *strict*, i.e. $h\ \mathbf{failed}_{\tau'} \;\cong\; \mathbf{failed}_{\{\tau\}}$. Under these circumstances, f can be defined like this.

$$\begin{array}{l} f :: \{\tau\} \\ f = h\ f' \\ f' :: \tau' \\ f' = g'\ f' \end{array}$$

This transformation is known as *fixed point fusion* [12]. A proof of this rule for SaLT would again rely on the semantics from [11]. To give some intuition, consider the following argument. The definition of $f = g\ f$ should be the "limit" of $g\ \mathbf{failed}_{\{\tau\}}$, $g\ (g\ \mathbf{failed}_{\{\tau\}})$, $g\ (g\ (g\ \mathbf{failed}_{\{\tau\}}))$ etc. If one knows that $g \circ h \;\cong\; h \circ g'$, one can reason as follows.

$$\begin{array}{l} g\ (\ldots (g\ \mathbf{failed}_{\{\tau\}}) \ldots) \\ \;\cong \quad \text{-- strictness of } h \\ g\ (\ldots (g\ (h\ \mathbf{failed}_{\tau'})) \ldots) \\ \;\cong \quad \text{-- since } g \circ h \;\cong\; h \circ g' \\ g\ (\ldots h\ (g'\ \mathbf{failed}_{\tau'}) \ldots) \\ \;\cong \quad \text{-- } \ldots \text{iterating} \ldots \\ \;\cong\; h\ (g'\ (\ldots (g'\ \mathbf{failed}_{\tau'}) \ldots)) \end{array}$$

So f is equivalent to the "limit" of $h\ (g'\ \mathbf{failed}_{\tau'})$, $h\ (g'\ (g'\ \mathbf{failed}_{\tau'}))$ etc., which means it is equivalent to f defined as $f = h\ f'$ where $f' = g'\ f'$.

In the concrete example above, we can choose $g := \lambda y \to y \ggg\!= \lambda x \to \{\mathsf{Cons}_{\mathsf{Nat}}\ 1\ x\}$ to satisfy $\mathit{ones} = g\ \mathit{ones}$. Furthermore one can show the following.

$$\begin{array}{l} g \circ \{\cdot\} \\ \;\cong\; \lambda y \to \{y\} \ggg\!= \lambda x \to \{\mathsf{Cons}_{\mathsf{Nat}}\ 1\ x\} \\ \;\cong\; \lambda y \to \{\mathsf{Cons}_{\mathsf{Nat}}\ 1\ y\} \\ \;\cong\; \{\cdot\} \circ \mathsf{Cons}_{\mathsf{Nat}}\ 1 \end{array}$$

So choosing $g' = \mathsf{Cons}_{\mathsf{Nat}}\ 1$ and $h = \{\cdot\}$ allows exactly the transformation we did above. The required strictness of $\{\cdot\}$ is a consequence of the semantics in [11].

Let us consider another example: the *length* function in CuMin.

$$\begin{array}{l} \mathit{length} :: \forall a.\mathsf{List}\ a \to \mathsf{Nat} \\ \mathit{length}\ \mathit{list} = \mathbf{case}\ \mathit{list}\ \mathbf{of} \\ \quad \mathsf{Nil} \qquad\quad \to 0 \\ \quad \mathsf{Cons}\ x\ \mathit{xs} \to 1 + \mathit{length}_a\ \mathit{xs} \end{array}$$

Translating it to SaLT yields the following function, after simplification.

$$\begin{array}{l} \mathit{length} :: \forall a.\{\mathsf{List}\ a \to \{\mathsf{Nat}\}\} \\ \mathit{length} = \{\lambda \mathit{list} :: \mathsf{List}\ a \to \mathbf{case}\ \mathit{list}\ \mathbf{of} \\ \quad \mathsf{Nil} \qquad\quad \to \{0\} \\ \quad \mathsf{Cons}\ x\ \mathit{xs} \to \mathit{length}_a \ggg\!= \lambda f \to \\ \qquad f\ \mathit{xs} \ggg\!= \lambda l \to \{1 + l\}\} \end{array}$$

One can "factor out" a non-recursive function g again.

$$\begin{array}{l} \mathit{length} :: \forall a.\{\mathsf{List}\ a \to \{\mathsf{Nat}\}\} \\ \mathit{length} = g\ \mathit{length} \end{array}$$

$$
\begin{aligned}
&g :: \forall a.\{\mathsf{List}\ a \to \{\mathsf{Nat}\}\} \to \{\mathsf{List}\ a \to \{\mathsf{Nat}\}\}\\
&g = \lambda \mathit{length}' \to \{\lambda \mathit{list} :: \mathsf{List}\ a \to \textbf{case}\ \mathit{list}\ \textbf{of}\\
&\quad \mathsf{Nil} \quad\quad\ \to \{0\}\\
&\quad \mathsf{Cons}\ x\ xs \to \mathit{length}' \gg\!= \lambda f \to\\
&\quad\quad f\ xs \gg\!= \lambda l \to \{1 + l\}\}
\end{aligned}
$$

As before, one can show an "extraction property" of g:

$$
\begin{aligned}
&g \circ (\{\cdot\} \circ (\{\cdot\}\circ))\\
&\ \cong \quad \text{-- definition of } (\circ),\ \beta\text{-reduction}\\
&\lambda \mathit{length}' :: (\mathsf{List}\ a \to \mathsf{Nat}) \to\\
&\{\lambda \mathit{list} :: \mathsf{List}\ a \to \textbf{case}\ \mathit{list}\ \textbf{of}\\
&\quad \mathsf{Nil} \quad\quad\ \to \{0\}\\
&\quad \mathsf{Cons}\ x\ xs \to \{\{\cdot\} \circ \mathit{length}'\} \gg\!= \lambda f \to\\
&\quad\quad f\ xs \gg\!= \lambda l \to \{1 + l\}\}\\
&\ \cong \quad \text{-- first monad law}\\
&\lambda \mathit{length}' :: (\mathsf{List}\ a \to \mathsf{Nat}) \to\\
&\{\lambda \mathit{list} :: \mathsf{List}\ a \to \textbf{case}\ \mathit{list}\ \textbf{of}\\
&\quad \mathsf{Nil} \quad\quad\ \to \{0\}\\
&\quad \mathsf{Cons}\ x\ xs \to \{\mathit{length}'\ xs\} \gg\!= \lambda l \to \{1 + l\}\}\\
&\ \cong \quad \text{-- first monad law}\\
&\lambda \mathit{length}' :: (\mathsf{List}\ a \to \mathsf{Nat}) \to\\
&\{\lambda \mathit{list} :: \mathsf{List}\ a \to \textbf{case}\ \mathit{list}\ \textbf{of}\\
&\quad \mathsf{Nil} \quad\quad\ \to \{0\}\\
&\quad \mathsf{Cons}\ x\ xs \to \{1 + \mathit{length}'\ xs\}\}\\
&\ \cong \quad \text{-- case rule}\\
&\lambda \mathit{length}' :: (\mathsf{List}\ a \to \mathsf{Nat}) \to\\
&\{\lambda \mathit{list} :: \mathsf{List}\ a \to \{\textbf{case}\ \mathit{list}\ \textbf{of}\\
&\quad \mathsf{Nil} \quad\quad\ \to 0\\
&\quad \mathsf{Cons}\ x\ xs \to 1 + \mathit{length}'\ xs\}\}\\
&\ \cong \quad \text{-- rewrite with combinators}\\
&(\{\cdot\} \circ (\{\cdot\}\circ)) \circ (\lambda \mathit{length}' :: (\mathsf{List}\ a \to \mathsf{Nat}) \to\\
&\lambda \mathit{list} :: \mathsf{List}\ a \to \textbf{case}\ \mathit{list}\ \textbf{of}\\
&\quad \mathsf{Nil} \quad\quad\ \to 0\\
&\quad \mathsf{Cons}\ x\ xs \to 1 + \mathit{length}'\ xs)\\
&\ \cong \quad \text{-- choosing } g' := \lambda \mathit{length}' :: (\mathsf{List}\ a \to \mathsf{Nat}) \to \ldots\\
&(\{\cdot\} \circ (\{\cdot\}\circ)) \circ g'
\end{aligned}
$$

Using fixed point fusion with $h = \{\cdot\} \circ (\{\cdot\}\circ)$ and g', we can equivalently define $\mathit{length} = \{\{\cdot\} \circ \mathit{length}'\}$ with $\mathit{length}' = g'\ \mathit{length}'$. Again, the strictness of h follows from the semantics in [11]. Inlining g' yields:

$$
\begin{aligned}
&\mathit{length}' :: \mathsf{List}\ a \to \mathsf{Nat}\\
&\mathit{length}' = \lambda \mathit{list} :: \mathsf{List}\ a \to \textbf{case}\ \mathit{list}\ \textbf{of}\\
&\quad \mathsf{Nil} \quad\quad\ \to 0\\
&\quad \mathsf{Cons}\ x\ xs \to 1 + \mathit{length}'\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{length} :: \{\mathsf{List}\ a \to \{\mathsf{Nat}\}\}\\
&\mathit{length} = \{\lambda \mathit{list} :: \mathsf{List}\ a \to \{\mathit{length}'\ \mathit{list}\}\}
\end{aligned}
$$

In this equivalent definition, it is obvious that *length* is deterministic. Furthermore, this is the optimal way of writing *length* and it was derived from the original translation using only well-specified program transformations.

So far, only directly recursive functions were discussed. But the transformation can be adapted to deal with mutually recursive functions. For instance, say there are two functions f and g, recursively calling each other. Then one creates two new functions f' and g' that have fewer sets and are mutually recursive, and defines f and g as “singleton wrappers” for f' and g'. However, the details are beyond the scope of this thesis.

5.6 Limitations and Related Work

While I have presented methods to analyze lots of functions, including recursive ones, I have not discussed functions with higher-order arguments, like *map*. Whether such a function is deterministic can depend on whether its higher-order argument is deterministic or not. So this has to be decided at the call site. Inlining can often solve this problem, but in case of recursive functions, it does not help.

There are other ways of analyzing nondeterminism that do not rely on syntactic transformations. For instance, a type and effect system can be used to track the nondeterminism in the program. Such an approach is described in [7].

However, the goal in this thesis was to analyze the nondeterminism behavior of programs by translating them to SaLT and applying program transformations. While there are some limitations, many CuMin expressions and functions can be shown to be (multi-)deterministic, using the general rules described in this chapter.

Chapter 6

Conclusion

In this thesis, I have explained and implemented an operational semantics for CuMin. The semantics helps to understand intricacies of CuMin's evaluation, such as call-time choice. In the implementation, the actual evaluation and nondeterminism are separated by lazily constructing a computation tree with all the possible nondeterministic results and then traversing it with either breadth-first or depth-first search, optionally limited to a certain depth. In most cases, breadth-first search seems to be the best option since it is complete and the performance is similar to depth-first search.

Moreover, I have described and implemented a translation from CuMin to SaLT. Superfluous sets in the generated SaLT code can often be eliminated using monad laws. Further simplifications are possible by making use of β- and η-reduction. I measured the performance difference between the optimized and unoptimized SaLT code, using the denotational semantics implemented by Fabian Thorand. In all test cases, the simplified code was significantly faster.

Finally, the translation procedure was used to better understand the nondeterminism of CuMin programs. To this end, I demonstrated by way of examples some general syntactic transformations that "extract" singleton sets from the corresponding SaLT code. Again, monad laws and β-reduction are useful, together with inlining of function definitions. The notion of multideterminism was defined, using the *sMap* combinator, and it was demonstrated how multideterministic functions can be analyzed. I also addressed recursive functions, for which inlining is not an option. However, by means of the fixpoint fusion rule, one can prove for many recursive functions that they are (multi-)deterministic as well.

Bibliography

[1] Michael D. Adams and Ömer S. Ağacan. "Indentation-sensitive Parsing for Parsec". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 121–132. ISBN: 978-1-4503-3041-1. DOI: `10.1145/2633357.2633369`. URL: `http://doi.acm.org/10.1145/2633357.2633369`.

[2] Elvira Albert et al. "Operational semantics for declarative multi-paradigm languages". In: *Journal of Symbolic Computation* 40.1 (2005). Reduction Strategies in Rewriting and Programming special issue, pp. 795–829. ISSN: 0747-7171. DOI: `10.1016/j.jsc.2004.01.001`. URL: `http://www.sciencedirect.com/science/article/pii/S0747717105000313`.

[3] S. Antoy and M. Hanus. "Compiling Multi-Paradigm Declarative Programs into Prolog". In: *Proc. of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*. Springer LNCS 1794, 2000, pp. 171–185.

[4] B. Brassel and F. Huch. "On a Tighter Integration of Functional and Logic Programming". In: *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapure, November 2007*. Vol. 4807. LNCS. Springer, 2007, pp. 122–138.

[5] Bernd Braßel et al. "KiCS2: A New Compiler from Curry to Haskell". In: *Functional and Constraint Logic Programming - 20th International Workshop, WFLP 2011, Odense, Denmark, July 19th, Proceedings*. 2011, pp. 1–18. DOI: `10.1007/978-3-642-22531-4_1`. URL: `http://dx.doi.org/10.1007/978-3-642-22531-4_1`.

[6] Michael Hanus. "Functional Logic Programming: From Theory to Curry". In: *Programming Logics - Essays in Memory of Harald Ganzinger*. 2013, pp. 123–168. DOI: `10.1007/978-3-642-37651-1_6`. URL: `http://dx.doi.org/10.1007/978-3-642-37651-1_6`.

[7] Michael Hanus and Frank Steiner. "Type-based Nondeterminism Checking in Functional Logic Programs". In: *Proceedings of the 2Nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '00. Montreal, Quebec, Canada: ACM, 2000, pp. 202–213. ISBN: 1-58113-265-4. DOI: `10.1145/351268.351292`. URL: `http://doi.acm.org/10.1145/351268.351292`.

[8] Graham Hutton and Erik Meijer. *Monadic Parser Combinators*. 1996.

[9] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 333–343. ISBN: 0-89791-692-1. DOI: `10.1145/199448.199528`. URL: `http://doi.acm.org/10.1145/199448.199528`.

[10] Stefan Mehner. "CuMin Report". (personal communication).

[11] Stefan Mehner et al. “Parametricity and Proving Free Theorems for Functional-Logic Languages”. In: *16th International Symposium on Principles and Practice of Declarative Programming, Canterbury, United Kingdom, Proceedings*. Ed. by Olivier Danvy. ACM Press, Sept. 2014. DOI: `10.1145/2643135.2643147`. URL: `http://www.janis-voigtlaender.eu/MSSV14.html`.

[12] Erik Meijer, Maarten Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: Springer-Verlag, 1991, pp. 124–144.

[13] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[14] Janis Voigtländer. “Asymptotic Improvement of Computations over Free Monads”. In: *Mathematics of Program Construction, Marseille, France, Proceedings*. Ed. by Christine Paulin-Mohring and Philippe Audebaud. Vol. 5133. LNCS. Springer-Verlag, July 2008, pp. 388–403. DOI: `10.1007/978-3-540-70594-9_20`. URL: `http://www.janis-voigtlaender.eu/Voi08d.html`.

[15] Philip Wadler. “A Prettier Printer”. In: *Journal of Functional Programming*. Palgrave Macmillan, 1998, pp. 223–244.

[16] Philip Wadler. “The Essence of Functional Programming”. In: Prentice Hall, 1992, pp. 1–14.

[17] Philip Wadler. “Theorems for free!” In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. ACM. 1989, pp. 347–359.