Rheinische Friedrich-Wilhelms-Universität Bonn

Bachelor Thesis

Implementing Denotational Semantics of a Functional-Logic Language and a Functional Language with Sets

Fabian Thorand
Student Number: STUDENTNO
STREET NUMBER
D - 53123 Bonn

March 26, 2015

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

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

Eigenständigkeitserklärung

Hiermit erkläre ich, 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

Contents

Preface

This bachelor thesis is based on "Parametricity and Proving Free Theorems for Functional-Logic Languages" by Stefan Mehner, Daniel Seidel, Lutz Straßburger and Janis Voigtländer [14]. They describe two programming languages, namely *CuMin*, a sublanguage of *Curry*, and *SaLT*, a lambda-calculus with explicit set types, together with denotational semantics for these languages.

As both languages are heavily influenced by *Haskell* and *Curry*, the first sections are a short introduction to these two. The remaining sections of the first chapter describe the syntax of CuMin and SaLT, and the differences to the original definition in the paper. The modifications were devised by Fabian Zaiser and me.

In the second chapter I will present the foundations necessary for working with CuMin and SaLT consisting of parsers and type checkers for both languages. Those libraries were developed together with Fabian Zaiser.

The third chapter is dedicated to the denotational semantics of CuMin and SaLT, and the adaptions I needed to perform to accommodate the modified language specifications. Subsequently, I will present my Haskell implementation of the two semantics in the fourth chapter.

Lastly, I will evaluate my implementation with respect to correctness and performance, and compare the results of the two interpreters to each other, using the CuMin to SaLT translation library written by Fabian Zaiser.

Chapter 1

Introduction

1.1 Preliminaries

The most visible feature of a programming language is the *syntax*, being the part that is directly exposed to the users of the language. Equally important is a description of what the syntax is supposed to *mean*, as this might not be obvious from the syntax alone. Consider for example the statement `x = x * 2`. Someone who has worked in imperative languages previously will most likely interpret it as an instruction which doubles the value stored in `x`. However, someone with a purely mathematical background might see this as an equation which has only $x = 0$ as a solution. Therefore, a so called *semantics* is needed to (more or less) formally specify the meaning of programs. In the domain of functional programming languages, one often distinguishes between *operational* and *denotational* semantics. The former defines how a term can be reduced to a value. It is called *operational* as it often can be directly implemented as an interpreter or compiler.

The latter kind of semantics describes a mathematical *denotation* of programs instead, in which syntactic entities are mapped to mathematical objects. It can be leveraged to prove certain properties concerning the results of programs using equational reasoning. For instance, one could prove that two different terms always yield the same values. When one of these terms can be computed more efficiently than the other, this could be used as an optimizing pass in a compiler.

In "Parametricity and Proving Free Theorems for Functional-Logic Languages", Mehner et al. describe two programming languages, namely *CuMin*, a sublanguage of *Curry*, and *SaLT*, a lambda-calculus with explicit set types, together with denotational semantics for theses languages [14].

While traditionally only operational semantics are realized in interpreters or compilers, I will describe how to implement an interpreter following the denotational semantics of CuMin and SaLT instead. For this purpose, I will cover how to represent the mathematical domain of term denotations in Haskell and how different search strategies can be implemented for a possibly infinite search space.

Such an interpreter would be useful when working with denotational semantics. A possible use case is the verification of program transformations. For finite non-determinism, this is achievable by exhaustively enumerating all possible answers; in the infinite case, the equivalence can be verified up to a given search depth.

An actual operational semantics for CuMin appears in Fabian Zaiser's bachelor thesis, developed in parallel to this thesis. Moreover, he presents an implementation of the translation procedure from CuMin to SaLT in [14] and an in-depth analysis of non-determinism in CuMin and SaLT programs.

There are already a couple of existing implementations for Curry, for example PAKCS [3] targeting Prolog and the more modern KiCS2 [5] compiling Curry to Haskell. Furthermore, an operational semantics for Curry has been described in [2]. *FlatCurry* is another sublanguage of Curry, which has been given a denotational semantics before [6]. The denotational semantics presented in [14] aims for compatibility with these and other existing approaches, even though these aspects are beyond the scope of this thesis.

Before describing Curry and SaLT in the last two sections in this chapter, I will first give a short overview of Haskell, which is the language I chose for realizing the interpreters, and Curry, which inspired the syntax and semantics of CuMin.

1.2 Haskell

Haskell is a purely functional programming language with a strong type system. Haskell functions usually behave like mathematical functions, i.e. for a given argument, they always evaluate to the same result. This is achieved by eschewing mutability and side-effects. Moreover, Haskell is lazy, meaning that expressions are not evaluated until their values are actually needed.

1.2.1 Function Definitions

Functions in Haskell are defined in an equational style. A function declaration consists of a type signature and one or more defining equations. The following examples should give some intuition.

$$
\begin{aligned}
&\mathit{add} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}\\
&\mathit{add}\ x\ y = x + y\\
&\mathit{add1} :: \mathsf{Int} \to \mathsf{Int}\\
&\mathit{add1} = \mathit{add}\ 1\\
&\mathit{fac} :: \mathsf{Int} \to \mathsf{Int}\\
&\mathit{fac}\ 0 = 1\\
&\mathit{fac}\ n = n * \mathit{fac}\ (n - 1)
\end{aligned}
$$

In most cases, the compiler is able to infer the types and therefore the type signatures could be omitted, but writing them down usually makes the code more readable and self-documenting. The type signatures show that the *add* function takes two arguments of type Int and returns an Int, *add1* and *fac* each take one argument of type Int and return an Int.

As the definition of *fac* shows, function declarations can be split into multiple defining equations. They are checked in order and the first one matching the argument(s) will be evaluated. Furthermore, Haskell imposes no restriction on recursion, which means that it is perfectly valid for *fac* to call itself again in the function body. On the downside, functions might not terminate, and therefore are not functions in a mathematical sense.

As one can see from *add1*, it is not necessary to list a parameter for each $\rightarrow$ occurring in the type signature. Partially applying *add* to 1 yields $add\ 1 :: \mathsf{Int} \rightarrow \mathsf{Int}$ which matches exactly the type signature of *add1*. $\mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$ can also be read as a function which takes an Int and returns another function $\mathsf{Int} \rightarrow \mathsf{Int}$.

1.2.2 Polymorphism

The following function differs from those previously shown in that its type contains so-called *type variables*, denoted by type identifiers starting with a lowercase letter. The following *const* function always returns its first argument. As the second argument is not used in the function body, the corresponding parameter can be ignored using an underscore.

$$const :: a \rightarrow b \rightarrow a$$
$$const\ x\ \_ = x$$

Type variables can be instantiated to any type when the function is used. Accordingly, *const* takes some a and some b and returns an a. Since *const* has to work for all types a, it has no knowledge about the arguments and can only pass them along. Hence, $const\ x\ y \equiv x$ (for all x and y) follows directly from the type signature[1]. Polymorphic functions can be used like any other function, for example $const\ 1\ \texttt{"hello world"} \equiv 1$.

1.2.3 Algebraic Data Types

Haskell uses so called algebraic data types to represent more complex values. Like functions, algebraic data types can be polymorphic. A declaration of an ADT consists of a name, a list of type variables and a list of constructor definitions separated by vertical bars. They are called *algebraic* because they are composed of *sums* and *products*, where a sum is a choice between several constructors and constructors are a product of their arguments. Consider the following declaration of a singly linked list.

$$\textbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)$$

The List type has a type parameter a and is comprised of a nullary constructor Nil and a binary constructor Cons which contains a value of type a and a value of type List a. The concrete type substituted for a is determined when the List type is actually used. A function calculating the sum of a list of integers then would have the type $sum :: \mathsf{List\ Int} \rightarrow \mathsf{Int}$. In this case, the List type has been instantiated to Int. Alternatively, for calculating the length of a list, the contents of the list are unimportant, therefore the corresponding function has the following signature $length :: \mathsf{List}\ a \rightarrow \mathsf{Int}$. Informally, one could say that the function knows about the structure of the list, but not about the elements stored inside.

To construct actual values, constructors can be used like functions taking the constructor fields as parameters. The list 1, 2, 3 would be written as $\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 3\ \mathsf{Nil}))$. Actually, Haskell has already a built-in list type called $[a]$. Nil is denoted by $[\,]$ and Cons by a colon, but for the rest of the introduction, I will keep using the exemplary definition from above.

[1] $f\ x\ y \equiv x$ is a so called *free theorem* [19] of functions $f :: a \rightarrow b \rightarrow a$.

1.2.4 Pattern Matching

Since there is a way to construct values of algebraic data types, there is also a way of inspecting them. This is what **case** expressions are for. They allow a distinction of the different constructors. The aforementioned *sum* function could be implemented as follows.

$$
\begin{aligned}
&\mathit{sum} :: \mathsf{List\ Int} \to \mathsf{Int} \\
&\mathit{sum}\ \mathit{lst} = \textbf{case}\ \mathit{lst}\ \textbf{of} \\
&\quad \mathsf{Cons}\ i\ \mathit{rst} \to i + \mathit{sum}\ \mathit{rst} \\
&\quad \mathsf{Nil} \qquad\ \to 0
\end{aligned}
$$

The expression between **case** and **of** is also called the *scrutinee*. Firstly, the scrutinee is evaluated, then its value is matched against the patterns in the given order. Variables in patterns serve as a wildcard, and when the constructors match, the values are bound to the variables in the body. In the above example, if the list value is a Cons, then the stored value is bound to i and the tail of the list is bound to *rst*.

In addition to **case** expressions, pattern matching can also be done in the defining equations of a function, as shown in the following definition of a function returning the head of a linked list when it is non-empty and nothing otherwise. Maybe is an option type whose values can either contain Just a value or Nothing.

$$
\begin{aligned}
&\mathit{safeHead} :: \mathsf{List}\ a \to \mathsf{Maybe}\ a \\
&\mathit{safeHead}\ (\mathsf{Cons}\ x\ \mathit{xs}) = \mathsf{Just}\ x \\
&\mathit{safeHead}\ \mathsf{Nil} \qquad\ = \mathsf{Nothing}
\end{aligned}
$$

1.2.5 Auxiliary Bindings

Sometimes it might be useful to bind intermediate results to a name. This is what **let** and **where** bindings are for. **let** bindings are expressions themselves, like $\textbf{let}\ x = 3\ \textbf{in}\ x + x$. In contrast, **where** bindings are part of a defining equation of a function or a case alternative and define the names after the actual expression, as shown in the following example.

$$
\begin{aligned}
&\mathit{doubleFac} :: \mathsf{Int} \to \mathsf{Int} \\
&\mathit{doubleFac}\ n = x + x\ \textbf{where}\ x = \mathit{fac}\ n
\end{aligned}
$$

Both variants can be used to define multiple variables at once by either listing the definitions at the same indentation level or by separating them with semicolons.

1.2.6 Higher-Order Programming

Functions taking other functions as parameters are called *higher-order functions*. This feature is an important part of the expressiveness of functional programming languages. A simple example for a higher-order function is the following.

$$
\begin{aligned}
&\mathit{doTwice} :: (a \to a) \to a \to a \\
&\mathit{doTwice}\ f\ x = f\ (f\ x)
\end{aligned}
$$

This function applies its function argument twice to its second argument. For example, $\mathit{doTwice}\ \mathit{add1}\ 1 \equiv \mathit{add1}\ (\mathit{add1}\ 1) \equiv 3$.

Sometimes it is tedious to give a name to a function, when it is only used once as an argument for a higher-order function. Therefore, Haskell also supports anonymous functions, also called *lambda abstractions*. An anonymous form of the aforementioned *add1* function would be $\lambda n.\ n + 1$. Lambda functions are allowed to have more than one parameter and their types are inferred automatically.

A more elaborate example of a higher-order function is

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to \mathsf{List}\ a \to b \\
&\mathit{foldr}\ f\ n\ \mathsf{Nil} &&= n \\
&\mathit{foldr}\ f\ n\ (\mathsf{Cons}\ x\ xs) &&= f\ x\ (\mathit{foldr}\ f\ n\ xs)
\end{aligned}
$$

which decomposes a list by having a fixed value for the base case Nil and a function to combine the values in the list with the intermediate result.

The above *sum* function could be implemented as $\mathit{foldr}\ (\lambda m\ n.\ m + n)\ 0$ which is partially applied and has the type $\mathsf{List\ Int} \to \mathsf{Int}$.

1.2.7 Type Classes

Haskell allows for some kind of function overloading via type classes [20]. As an example, consider the following definition of the Eq type class from the Haskell prelude, which defines two operators $(\equiv)$ and $(\not\equiv)$ with polymorphic arguments.

$$
\begin{aligned}
&\textbf{class}\ \mathsf{Eq}\ a\ \textbf{where} \\
&\quad (\equiv) :: a \to a \to \mathsf{Bool} \\
&\quad x \equiv y = \mathit{not}\ (x \not\equiv y) \\
&\quad (\not\equiv) :: a \to a \to \mathsf{Bool} \\
&\quad x \not\equiv y = \mathit{not}\ (x \equiv y)
\end{aligned}
$$

These operators can be implemented for a concrete type by writing an *instance* of the type class. In the above example, there are default implementations which are used when an instance declaration omits the definition of a function of the type class.

In this case, the default implementations are mutually recursive, meaning that a call would diverge regardless of the arguments.

While the instances for some built-in type classes like Eq can be automatically derived by the compiler, it is also possible to define the instances manually, as the following example shows.

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Eq}\ a \Rightarrow \mathsf{Eq}\ (\mathsf{List}\ a)\ \textbf{where} \\
&\quad \mathsf{Nil} &&\equiv \mathsf{Nil} &&= \mathsf{True} \\
&\quad (\mathsf{Cons}\ x\ xs) &&\equiv (\mathsf{Cons}\ y\ ys) &&= (x \equiv y) \wedge (xs \equiv ys) \\
&\quad \_ &&\equiv \_ &&= \mathsf{False}
\end{aligned}
$$

Since the default implementation of $(\not\equiv)$ is $\mathit{not}\ (x \equiv y)$, it suffices to implement $(\equiv)$.

Note that the function[2] definitions in a type class can be recursive. Furthermore, instances can be constrained. In this example, List a is only a member of the type class when a is, since $(\equiv)$ is needed in the second case of the above definition.

[2] operators, like $(\equiv)$, are also considered functions

In addition to instance declarations, type class constraints can be imposed to type signature of functions. The $nub :: \mathsf{Eq}\ a \Rightarrow [a] \to [a]$ function, for example, removes duplicate elements from a list and consequently needs to be able to check pairs of elements for equality. The $\mathsf{Eq}\ a$ constraint allows $(\equiv)$ and $(\not\equiv)$ to be used on the elements of the argument list inside the function body.

1.2.8 Laziness

As mentioned before, Haskell has *lazy*[3] semantics meaning that evaluation of expressions is deferred until the value is actually needed. The following code serves as an example.

$$const\ x\ \_ = x$$
$$loop = loop$$

The *const* function always returns its first argument and *loop* diverges as it is unconditionally recursive. In strict languages, such as C or Java, *print* (*const* 1 *loop*) would diverge as well because the arguments would be evaluated before making the call. In Haskell on the other hand, the second argument of *const* is discarded. Therefore, its value is never demanded and *print* (*const* 1 *loop*) will print a one to the standard output.

1.3 Curry

Curry [9] is a so called functional logic programming language. It combines features known from functional programming with the core aspects of logic programming such as logic variables (sometimes also called free variables) and non-determinism. The syntax and language features of Curry are in many ways similar to Haskell. This section is designed as a small overview of the essential logic programming features.

1.3.1 Non-Determinism

Non-determinism means that a given expression may evaluate to more than one value or no value at all whereof usually each satisfies some given constraints. It can be introduced by overlapping function equations, which is exemplified by the following code snippet defining a non-deterministic choice between two values. In contrast to Haskell, all matching equations are non-deterministically evaluated as opposed to only the first one.

$$choose\ x\ \_ = x$$
$$choose\ \_\ y = y$$

Now *choose* 1 2 non-deterministically yields both 1 and 2 as answers. There is also a predefined operator (?) with the same functionality.

Curry provides the constrained equality operator $(=:=) :: a \to a \to \mathsf{Success}$ for constraining the outcome of a non-deterministic computation. It succeeds when both arguments can be evaluated to the same value and returns success, otherwise it fails, returning no value. The constrained expression operator $(\&>) :: \mathsf{Success} \to a \to a$

[3] actually *non-strict*, but most implementations, in particular GHC, realize this through laziness

```
data Person = Alice | Bob | Eve | Peter | Paul
parent Eve  = Alice
parent Eve  = Bob
parent Paul = Eve
parent Paul = Peter
children a      | parent b =:= a                          = b where b     free
grandparent a | parent a =:= b & parent b =:= c = c where b, c free
```

Listing 1.1: Example of a Curry program

returns the second argument on success, and otherwise returns no value. The expression **let** $n = \mathit{choose}\ 1\ 2$ **in** $(1 + n =:= 2)\ \&> n$ checks whether $1 + 1$ or $1 + 2$ is 2, returning the solution on success.

1.3.2 Logic Variables

In Curry, variables can be *free* in that they are not bound to a value when they are defined. Their value is determined implicitly by the constraints imposed through the expressions referencing them. Logic variables are introduced by **let** or **where** bindings using the keyword **free**. For instance, **let** x **free in** e or e **where** x **free** declare x as a logic variable in the scope of e. When evaluating an expression, Curry solves the logic variables and returns all possible values if there is more than one solution satisfying the constraints.

The example in listing 1.1 shows how to use logic variables in combination with non-determinism. The *parent* function assigns two parents for each person (except for the top level of the family tree). The inverse mapping is given by *children*. Some person b is a child of person a, if *parent* b is a again. A more elaborate usage of logic variables is *grandparent*. Some person c is a grandparent of a, if there is another person b such that b is parent of a and c is parent of b.

1.4 CuMin

CuMin [14] stands for Curry Minor and is a small sublanguage of Curry which none the less has the basic features required for logic programming, i.e. non-determinism and logic variables. Many Curry programs can be expressed in CuMin, although it lacks a lot of syntactic sugar like lambda-abstractions, where-clauses and left-hand side pattern matching. Let-bindings are not recursive; recursion is only possible for top-level bindings.

Fabian Zaiser and I added syntax for algebraic data types and the corresponding generalized case expressions which were not part of the original language definition. For an example of how Curry is translated to CuMin, compare listings 1.1 and 1.2. Figure 1.2 shows the symbolic syntax of CuMin. In addition to the symbolic syntax notation used throughout this chapter, there is also an ASCII representation which looks similar to actual Curry or Haskell source code and follows relatively straight-forward from the symbolic syntax (see figure 1.1).

Symbolic Syntax	ASCII Equivalent
$\forall$	`forall`
$\Rightarrow$	`=>`
$\rightarrow$	`->`
$\equiv$	`==`
f_{τ_1,τ_2}	`f<:tau1,tau2:>`

Figure 1.1: ASCII representation of symbolic syntax

```
data Person = Alice | Bob | Eve | Peter | Paul
parent :: Person → Person
parent p = case p of
  Eve  → choose_Person Alice Bob
  Paul → choose_Person Eve   Peter
  x    → failed_Person
children :: Person → Person
children a = let b :: Person free in case a ≡ parent b of
  { True → b; False → failed_Person }
grandparent :: Person → Person
grandparent a = let b :: Person free in let c :: Person free in
  case b ≡ parent a of
    True → case c ≡ parent b of
      { True → c; False → failed_Person }
    False → failed_Person
```

Listing 1.2: Example program 1.1 translated to CuMin

1.4.1 Language Features and Syntax

A program is a list of function bindings and data declarations. The syntax of algebraic data types is based on Haskell and similar to the one shown in the examples above. Higher-kinded type variables are not supported, i.e. while **data** Example f = Example (f Int) is valid Haskell code, it is neither valid for CuMin nor for SaLT.

To keep the type checker simple, CuMin requires explicit instantiations of polymorphic values which are denoted by `<:T1,...:>` in the ASCII representation or by subscripts (i.e. $fun_{\tau_1,\ldots}$) in the mathematical notation. If a value is monomorphic, the type instantiation brackets may be omitted. At the declaration site of a function, all type variables in a function signature must be captured with an explicit `forall`. Other than that, function and constructor application works just as expected when coming from Haskell. In contrast to the first language specification, constructors may now be partially applied.

Case expressions look similar to Haskell, but with some restrictions. Patterns can be either a constructor followed by one variable name for each argument, or a catch-all (wildcard) pattern consisting of just a variable name. The case alternatives can either be grouped by indentation or be enclosed by braces and separated by semicolons instead. If a

$$
\begin{aligned}
P &::= D; P \mid D \\
D &::= f :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau; f\ \overline{x_n} = e \\
&\quad \mid \mathbf{data}\ \mathit{TyCon}\ \overline{\alpha_m} = \overline{\mathit{Con}_i\ \overline{\tau_{i_j}}} \\
\tau &::= \alpha \mid \tau \to \tau' \mid \mathit{TyCon}\ \overline{\tau_m} \\
e &::= x \mid f_{\overline{\tau_m}} \mid \mathit{Con}_{\overline{\tau_m}} \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e \\
&\quad \mid \mathbf{let}\ x :: \tau\ \mathbf{free\ in}\ e \mid \mathbf{failed}_\tau \\
&\quad \mid n \mid e_1\ e_2 \mid e_1 + e_2 \mid e_1 \equiv e_2 \\
&\quad \mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathit{Con}\ \overline{x_n}}\} \\
&\quad \mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathit{Con}\ \overline{x_n}}; x \to e\}
\end{aligned}
$$

Figure 1.2: Syntax of CuMin (adapted from [14])

$\mathit{coin} :: \mathsf{Nat}$	$\mathit{dc1} :: \mathsf{Nat}$	$\mathit{dc2} :: \mathsf{Nat}$
$\mathit{coin} = \mathit{choose}_{\mathsf{Nat}}\ 0\ 1$	$\mathit{dc1} = \mathit{coin} + \mathit{coin}$	$\mathit{dc2} = \mathbf{let}\ x = \mathit{coin}\ \mathbf{in}\ x + x$

Listing 1.3: Example for sharing in CuMin, slightly adapted from [14]

catch-all pattern is used, it must be the last one and has to be preceded by at least one constructor pattern.

Let bindings can bind only one variable at a time and are used to declare logic variables (**let** $x :: \tau$ **free in** e) as well as regular bindings (**let** $x = e_1$ **in** e_2). Logic variables are not allowed to have arbitrary types, since there is no reasonable way to enumerate all values of a function type $\tau_1 \to \tau_2$. Instead, their types are restricted to members of the Data class which behaves a lot like a regular Haskell type class with one parameter, except that it cannot be explicitly declared but instead is implicitly derived by the typing rules described below. The syntax for type declarations is similar to that of Haskell; $\forall a\ b.\mathsf{Data}\ a \Rightarrow b \to (a, b)$ is a valid CuMin type signature, for example.

Let bindings are important to enforce sharing of non-deterministic values, as demonstrated by the example in listing 1.3.

dc1 evaluates to all three values 0, 1 and 2 because the two occurrences of *coin* are independent of each other, whereas *dc2* only returns 0 and 2 because the value for x is chosen at the time of the binding and not at the time where x occurs. Hence, the body either evaluates to $0 + 0$ or $1 + 1$. The other way to ensure sharing are function calls, as the arguments are non-deterministically chosen at call time.

Strongly related to non-determinism is the $\mathbf{failed}_\tau$ primitive, which indicates a non-deterministic computation without any result. Non-deterministic choice is not a primitive, because it can be implemented in terms of logic variables. The $\mathit{choose} :: \forall a.a \to a \to a$ function is defined in the prelude of the language implementation presented in the next chapter.

The only primitive type (apart from $\to$) is the type of non-negative integers Nat because of the way it was defined in the original paper. It comes with the built-in addition operator $+$. Every literal number in CuMin source code is of type Nat. Technically, there is no reason why numbers are restricted to non-negative values and there is only an addition operator, but we decided to maintain the first language specification in this regard. Since we now have algebraic data types, custom Nat implementations can be easily implemented

by users of the language.

Another usage of the above-mentioned Data class is the structural equality operator $(\equiv) :: \forall a.\mathsf{Data}\ a \Rightarrow a \to a \to \mathsf{Bool}$ (== in ASCII). Notably, this is a generalization from the original paper, where $(\equiv)$ was only defined for Nat. The behavior on Nat remains unchanged. Additionally, two values of an algebraic data type are said to be structurally equal when the constructors match and each pair of arguments is structurally equal. If at least one argument is a failure, then the whole comparison fails. Arguments are compared left to right and are short-cutting. Especially, $\mathsf{Cons}_{\mathsf{Nat}}\ 1\ \mathsf{Nil}_{\mathsf{Nat}} \equiv \mathsf{Cons}_{\mathsf{Nat}}\ 2\ \mathbf{failed}_{\mathsf{List\ Nat}}$ evaluates to False. Structural equality is restricted to Data values, because there is no sensible way two compare two arbitrary functions.

1.4.2 Typing Rules

There are four kinds of typing judgements relevant for CuMin. The two kinds of auxiliary judgements shown in figure 1.3 state what constitutes contexts and types. The third kind of rules in figure 1.4 defines which types have Data instances and the other judgements in figure 1.5 describe well-typed CuMin expressions. All typing judgements are defined with respect to some fixed CuMin program which provides the function and data declarations. Formally, all typing judgements needed to be indexed by the actual program, but for better readability, this indexing is omitted.

The addition of ADTs and partial constructor applications required a few changes to the typing rules. The rule for constructor expressions now states that each constructor has the type of a curried function mapping its arguments to its defining ADT which is abstracted over the type parameters of the corresponding data type.

The **case**-rule requires that all case alternatives have the same type τ in their respective contexts given by the patterns. Furthermore, constructor patterns must match the type of the scrutinee.

The Data-rule states that each algebraic data type has a data instance if each of its type arguments either has a Data instance too or it is a phantom parameter. A a type parameter is said to be *phantom* if is not actually used to store data inside one of the constructors. Consider a type like **data** A a = A where the a parameter is not used. Then the type parameter in **data** B a = B (A a) is also phantom.

Note that by instantiating the generalized Type-, *Con*-, **case**- and Data-rules to Bool, List and Pair, the original rules can be directly recovered. The following derivation shows this exemplarily for the Pair constructor.

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{Pair}_{\tau_1,\tau_2} :: \tau_1 \to \tau_2 \to \mathsf{Pair}\ \tau_1\ \tau_2 \qquad \dfrac{\ldots}{\Gamma \vdash e_1 :: \tau_1}}{\Gamma \vdash \mathsf{Pair}_{\tau_1,\tau_2}\ e_1 :: \tau_2 \to \mathsf{Pair}\ \tau_1\ \tau_2} \qquad \dfrac{\ldots}{\Gamma \vdash e_2 :: \tau_2}}{\Gamma \vdash \mathsf{Pair}_{\tau_1,\tau_2}\ e_1\ e_2 :: \mathsf{Pair}\ \tau_1\ \tau_2}$$

While the rules discussed above only define which types and terms are valid, the following definition extends the concept of being *well-typed* to whole programs.

Definition 1.1. A CuMin program is well-typed if

1. for each n-ary function declaration $f :: \forall\overline{\alpha_m}.(\mathsf{Data}\ \alpha_{i_1}, \ldots \mathsf{Data}\ \alpha_{i_j}) \Rightarrow \tau; f\ \overline{x_n} = e$,

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

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

$$\frac{\forall i \in \{1, \ldots, m\} \,.\, \Gamma \vdash \tau_i \in \mathsf{Type}}{\Gamma \vdash \mathsf{D}\ \overline{\tau_m} \in \mathsf{Type}}$$
for every algebraic data type $\mathsf{D}\ \overline{\alpha_m}$

Figure 1.3: Rules for auxiliary typing judgements

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

$$\frac{\Gamma \vdash \forall i \in \{1, \ldots, m\} \,.(\tau_i \in \mathsf{Data} \vee \alpha_i \text{ is phantom parameter})}{\Gamma \vdash \mathsf{D}\ \overline{\tau_m} \in \mathsf{Data}}$$
for every algebraic data type $\mathsf{D}\ \overline{\alpha_m}$

Figure 1.4: Rules for being a data type adapted from [14]

a) the type τ is comprised of at least n function arrows, i.e. $\tau = \tau_1 \to \ldots \to \tau_n \to \tau'$ and

b) the typing judgement $\alpha_1, \ldots, \alpha_m, \alpha_{i_1} \in \mathsf{Data}, \ldots, \alpha_{i_j} \in \mathsf{Data}, x_1 :: \tau_1, \ldots, x_n :: \tau_n \vdash e :: \tau'$ can be derived

2. each argument in a constructor definition only consists of valid types (as defined in figure 1.3)

1.5 SaLT

SaLT shares most of the basic syntax features with CuMin, but in contrast to the latter, non-determinism is not hidden, but explicitly annotated by a built-in Set type. In order To work with sets, there are two additional syntax features. A singleton set containing $x :: \tau$ is denoted by $\{x\}$ (and has the type $\mathsf{Set}\ \tau$ or $\{\tau\}$) and an indexed union $\bigcup_{x \in e}(f\ x)$ is denoted by $e \gg\!= \lambda x :: \tau . f\ x$ for some expression $e :: \mathsf{Set}\ \tau$ and a function $f :: \tau \to \mathsf{Set}\ \tau'$. The latter lambda abstraction can be replaced by a partial application of f (with no arguments), resulting in $s \gg\!= f$. Formally, the indexed union operator has the type $(\gg\!=) :: \mathsf{Set}\ a \to (a \to \mathsf{Set}\ b) \to \mathsf{Set}\ b$. The full syntax of SaLT is presented in figure 1.6.

Additionally, there are no let bindings and parameters may not be listed on the left hand side of a function definition. In exchange, there are lambda abstractions, which have a similar syntax to Haskell, but need an explicit type annotation, and only one variable may be bound per lambda abstraction. Instead of **let** $x :: \tau$ **free in** e, SaLT provides the **unknown** $:: \forall a.\mathsf{Data}\ a \Rightarrow \mathsf{Set}\ a$ primitive to reify the set of all values inhabiting the given data type.

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash \mathbf{failed}_\tau :: \tau \qquad \Gamma \vdash n :: \mathsf{Nat}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash e :: \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e :: \tau} \qquad \frac{\Gamma, x :: \tau_1 \vdash e :: \tau \qquad \Gamma \vdash \tau_1 \in \mathsf{Data}}{\Gamma \vdash \mathbf{let}\ x :: \tau_1\ \mathbf{free\ in}\ e :: \tau}$$

$$\frac{\Gamma \vdash \tau_{i_1} \in \mathsf{Data} \qquad \ldots \qquad \Gamma \vdash \tau_{i_j} \in \mathsf{Data}}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau\ [\overline{\tau_m\ /\ \alpha_m}]}$$
for every function $(f :: \forall \overline{\alpha_m}.(\mathsf{Data}\ \alpha_{i_1}, \ldots, \mathsf{Data}\ \alpha_{i_j}) \Rightarrow \tau; f\ \overline{x_n} = e)$

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

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

$$\Gamma \vdash \mathsf{C}_{\overline{\tau_m}} :: \tau'_1\ [\overline{\tau_m\ /\ \alpha_m}] \to \ldots \to \tau'_n\ [\overline{\tau_m\ /\ \alpha_m}] \to \mathsf{D}\ \overline{\tau_m}$$
for every constructor $\mathsf{C}\ \overline{\tau'_n}$ of a data type $\mathsf{D}\ \overline{\alpha_m}$

$$\frac{\Gamma \vdash e :: D_l\ \overline{\tau_m} \qquad \forall k.\ \begin{cases} \Gamma, x :: D_l\ \overline{\tau_m} \vdash e' :: \tau & \text{if } Alt_k = x \to e' \\ \Gamma, \overline{x_{i_j} :: \tau_{i_j}\ [\overline{\tau_m\ /\ \alpha_m}]} \vdash e' :: \tau & \text{if } Alt_k = C_i\ \overline{x_{i_j}} \to e' \end{cases}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{Alt_k}\} :: \tau}$$
for every declaration $\mathbf{data}\ D_l\ \overline{\alpha_m} = \overline{C_i\ \overline{\tau_{i_j}}}$

Figure 1.5: Typing rules for CuMin adapted from [14]

$$\begin{aligned}
P &::= D; P \mid D \\
D &::= f :: \forall \overline{\alpha_m}.(\overline{\mathsf{Data}\ \alpha_{i_j}}) \Rightarrow \tau; f = e \\
&\quad \mid \mathbf{data}\ \mathit{TyCon}\ \overline{\alpha_m} = \overline{\mathit{Con}_i\ \overline{\tau_{i_j}}} \\
\tau &::= \alpha \mid \tau \to \tau' \mid \{\tau\} \mid \mathit{TyCon}\ \overline{\tau_m} \\
e &::= x \mid f_{\overline{\tau_m}} \mid \mathit{Con}_{\overline{\tau_m}} \mid \lambda x :: \tau.\ e \mid n \\
&\quad \mid \{e\} \mid e_1 \gg\!= e_2 \mid e_1 + e_2 \mid e_1 \equiv e_2 \\
&\quad \mid e_1\ e_2 \mid \mathbf{unknown}_\tau \mid \mathbf{failed}_\tau \\
&\quad \mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathit{Con}\ \overline{x_n}}\} \\
&\quad \mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathit{Con}\ \overline{x_n}}; x \to e\}
\end{aligned}$$

Figure 1.6: Syntax of SaLT (adapted from [14])

```
data Person = Alice | Bob | Eve | Peter | Paul
choose :: ∀α.Set α → Set α → Set α
choose = λas :: Set α. λbs :: Set α.
  unknown_Bool ≫= λc :: Bool. case b of
    False → as
    True  → bs
parent :: Person → Set Person
parent = λp :: Person. case p of
  Eve → choose_Person {Alice} {Bob}
  Paul → choose_Person {Eve}  {Peter}
  x    → failed_Set Person
children :: Person → Set Person
children = λa :: Person. unknown_Person ≫= λb :: Person.
  parent b ≫= λp :: Person. case a ≡ p of
    {True → {b}; False → failed_Set Person}
grandparent :: Person → Set Person
grandparent = λa :: Person. unknown_Person ≫= λb :: Person.
  unknown_Person ≫= λc :: Person. parent a ≫= λpa :: Person.
    case b ≡ pa of
      True → parent b ≫= λpb :: Person. case c ≡ pb of
        {True → {c}; False → failed_Set Person}
      False → failed_Set Person
```

Listing 1.4: Example program 1.2 translated to SaLT

These language features are sufficient to express every CuMin program in SaLT as already shown in [14]. For example, a logic variable **let** $x :: \tau$ **free in** e can be translated to $\mathbf{unknown}_\tau \gg\!= \lambda x :: \tau.\, e'$ where e' is the translation of e.

Listing 1.4 shows a translation of the example program from above to SaLT. Note that this is not the SaLT code which would result from the translation method presented in the original paper.

1.5.1 Typing Rules

The typing rules of SaLT are mostly identical to CuMin, apart from the rules needed for lambda abstractions and set operations. Since data types behave identically in CuMin and SaLT, the auxiliary and Data typing judgements from figures 1.3 and 1.4 can be reused. A rule stating that Set τ is a type must be added. The additional typing rules are shown in figure 1.7. Since SaLT has no let-bindings, those typing judgements are not used.

Definition 1.2. A SaLT program is well-typed if

1. as in [14], for each function declaration $f :: \forall\overline{\alpha_m}.(\mathsf{Data}\ \alpha_{i_1}, \ldots \mathsf{Data}\ \alpha_{i_j}) \Rightarrow \tau; f = e$, the typing judgement $\alpha_1, \ldots, \alpha_m, \alpha_{i_1} \in \mathsf{Data}, \ldots, \alpha_{i_j} \in \mathsf{Data} \vdash e :: \tau$ holds, and

2. each argument in a constructor definition only consists of valid types (as defined in figure 1.3)

$$\frac{\Gamma \vdash \tau \in \mathsf{Type}}{\Gamma \vdash \{\tau\} \in \mathsf{Type}} \qquad \frac{\Gamma, x :: \tau_1 \vdash e :: \tau_2}{\Gamma \vdash \lambda x :: \tau_1.\, e :: \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash \mathbf{unknown}_\tau :: \{\tau\}}$$

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \{e\} :: \{\tau\}} \qquad \frac{\Gamma \vdash e_1 :: \{\tau_1\} \qquad \Gamma \vdash e_2 :: \tau_1 \to \{\tau_2\}}{\Gamma \vdash e_1 \gg\!= e_2 :: \{\tau_2\}}$$

Figure 1.7: Additional typing rules for SaLT adapted from [14]

Chapter 2

Language Implementation

Before I can elaborate on the implementation of the denotational semantics, a few additional parts are needed as groundwork, most notably parsers and type-checkers for CuMin and SaLT. Those were developed together with Fabian Zaiser, who implements the operational semantics in his bachelor thesis. In the following sections, I will briefly describe the libraries we created for working with CuMin and SaLT code. The implementation is subdivided into the packages `funlogic-core`, `language-cumin` and `language-salt`. The former contains code that can be reused across CuMin and SaLT, the latter two each consist of types for an AST (abstract syntax tree), a parser for transforming source code into an AST, a pretty printer doing the reverse transformation and a type-checker.

2.1 Representation

Parsed CuMin and SaLT modules are represented by a hierarchy of data types. The root is CoreModule b, which is parameterized over the type of bindings b, because the top level bindings and expression types are the only differences between a CuMin and a SaLT module. Each language package defines its own type synonym Module by filling in the corresponding top level binding type Binding. This abstraction allows writing code which works for syntax trees of both languages.

Each module has a name and two maps containing the ADTs and top level bindings of that module. The data types modeling the abstract syntax tree tightly correspond to the syntax definitions from figures 1.2 and 1.6. Since the syntax for type signatures and data types is the same for CuMin and SaLT, the corresponding types for the abstract syntax tree are contained in `funlogic-core`.

Variables are simply represented by their names. Therefore, special care must be taken during substitution to avoid unintentional capturing of free variables. While there are other more elaborate binding schemes, we figured that this is an unnecessary complication, because we do not need term substitution in the core libraries. Type variables on the other hand are only bound in the top level type signatures and not in the types itself, thus, substitution in types is not a problem either.

2.2 Parsing

For parsing, we rely on the parser combinators [10] provided by `trifecta`[1]. Since we also wanted to have indentation sensitive parsing to make the syntax look more like Haskell, we use the `indentation` [1] package containing additional combinators for `trifecta` for indentation parsing.

The advantage of using parser combinators instead of the traditional approach of parser generators is that it is easier to work with an EDSL[2] in Haskell than to work with the generated code. While the resulting parser might be slightly slower, this is negligible for the expected source file sizes.

Transforming source code to an abstract syntax tree is a two-stage process. Firstly, all top level declarations (i.e. data and function declarations) are parsed and collected. Then in the second stage, all names are checked for uniqueness. There may not be two functions, two data types or two constructors with the same name. On the other hand, a data type is allowed to have the same name as a constructor, because the former is a type level entity, the latter a term level entity.

2.3 Pretty Printing

The inverse operation to parsing is *pretty printing*, transforming an AST into the corresponding source code, which will yield the same AST again when parsed. We use the `ansi-wl-pprint`[3] which provides combinators to compose an abstract document with indentation and formatting annotations.

2.4 Type Checker

Note that our abstract syntax tree has no means to ensure that only well-typed expressions can be represented. While this would be theoretically possible[4], the presence of arbitrary algebraic data types would make this a tedious task. Instead, we wrote a separate type checker which should be invoked after parsing a module.

The type checker is implemented using a monadic computation [18] which provides an environment consisting of a global and a local scope and handles errors[5]. The global scope contains the types of constructors and functions as well as the Data instances, whereas the local scope keeps track of the types of local variables, the presence of type variables and local Data constraints. The type checking is canceled after the first type mismatch has been found.

[1] `https://hackage.haskell.org/package/trifecta`

[2] embedded domain specific language

[3] `https://hackage.haskell.org/package/ansi-wl-pprint`

[4] Guillemette and Monnier present a type safe intermediate representation of a small functional language with bindings in [8]

[5] *Monads* are a way of structuring functional programs by introducing a notion of sequentiality and by handling effects. Examples of effects that can be modeled with monads are exceptions and mutable state.

2.4.1 Algebraic Data Types and Data Instances

Firstly, algebraic data types are checked by ensuring that the type variables used in the constructor definitions have previously been declared and that each type constructor application has the correct arity, i.e. the number of arguments matches the number of type parameters. For example List Nat Nat would be invalid since the List type constructor only takes one type argument.

Secondly, the Data instances are derived. For that, the type checker must first recursively determine which type parameters are actually used (i.e. which are non-phantom) inside the declaration. The set of non-phantom types is determined by a fixpoint iteration. In each step, every argument that either appears directly as constructor argument or as a non-phantom (as known so far) parameter of a data type is marked as non-phantom. Furthermore, if $(\rightarrow)$ or Set occurs in a non-phantom position, the Data instance for the affected type is removed altogether.

This construction will always yield a fixpoint after a finite number of steps, since there is only a finite number of data types and parameters, and the construction is monotone, i.e. once a parameter has been marked non-phantom, this mark is never removed and once a Data instance has been removed, it is never added again. Moreover, it yields the largest set of Data instances, and if there is more than one, it returns the one with a minimal number of constraints.

2.4.2 Top-Level Declarations

Lastly, all function definitions are checked. Similar to the verification of ADTs, all type variables used in the type signature and in the function body have to be listed in the outermost $\forall\cdot$, and each type constructor application has to have the correct arity. In order to check (mutually) recursive functions, the type checker assumes that all top level type signatures are valid when checking a particular function body. Especially, the types derived for the function bodies have to match the given top level definitions. In the case of CuMin, the types of the function parameters are deduced from the type signature before descending into the function body. The code for checking the different syntax elements can be (almost) directly adopted from the typing rules given in figures 1.5 and 1.7, since the extensive use of type annotation allows for a straightforward bottom-up inference. Only local information and the types of subexpressions are needed to infer the type of a larger expression.

For example, the type of a let binding of the form **let** $x = e_1$ **in** e_2 can be inferred by first inferring the type of the expression e_1, and then inferring e_2 with the context extended by $x :: \tau$ (where τ is the type of e_1).

```
go (ELet var e1 e2) = do
    varTy <- checkExp e1
    local (localScope.at var .~ Just varTy) (checkExp e2)
```

If *checkExp* fails to infer a type for e_1, the type checker monad ensures that the failure is propagated outwards, even though this is not directly apparent from the code excerpt. On success, the type of e_1 is bound to *varTy* using the $\leftarrow$-syntax. Note how the **do**-block implicitly introduces a notion of sequentiality. The *local* function modifies the environment

```
data Pair a b = Pair a b
data List a = Nil | Cons a (List a)
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Bool = False | True
and :: Bool → Bool → Bool
choose :: ∀a.a → a → a
const :: ∀a b.a → b → a
either :: ∀a b c.(a → c) → (b → c)
  → Either a b → c
filter :: ∀a.(a → Bool) → List a → List a
flip :: ∀a b c.(a → b → c) → b → a → c
foldr :: ∀a b.(a → b → b) → b → List a → b
fst :: ∀a b.Pair a b → a
guard :: ∀a.Bool → a → a
id :: ∀a.a → a
length :: ∀a.List a → Nat
map :: ∀a b.(a → b) → List a → List b
maybe :: ∀a b.b → (a → b) → Maybe a → b
not :: Bool → Bool
or :: Bool → Bool → Bool
snd :: ∀a b.Pair a b → b
```

Listing 2.1: CuMin Prelude

carried by the monad in the scope of the second argument. In this case, the scope is extended with type information about the variable *var* to check e_2.

The *checkExp* function performs the type inference and checking of expressions. There are analogous functions for checking modules, ADTs and case alternatives. In the case of the above code snippet, the type of the bound variable is inferred before extending the context with the type of the variable during the inference of the body of the let-expression.

2.5 Prelude

We also devised prelude modules (see listing 2.1) for CuMin and SaLT, which contain a small set of useful data types and higher-order combinators to work with them. The prelude is made available by the respective libraries as an abstract syntax tree which has been included at compile time via Template Haskell [17].

Most CuMin and SaLT programs will need the prelude since some of the built-in operators rely on the presence of certain data types, for instance, ($\equiv$) returns Bool and the syntactic sugar for lists needs the List data type.

The SaLT prelude contains almost the same functions as the CuMin prelude (listing 2.1) with two changes: *choose* :: $\forall a.\mathsf{Set}\ a \to \mathsf{Set}\ a \to \mathsf{Set}\ a$ now explicitly mentions the non-determinism in the type signature and there is a new function *sMap* :: $\forall a\ b.(a \to b) \to \mathsf{Set}\ a \to \mathsf{Set}\ b$ for mapping over the elements of a set. The type signatures of the other functions remain the same, with the consequence that those functions are now visibly deterministic.

As a result, programs translated from CuMin to SaLT will not work together with the deterministic prelude. For example, the translated type of *choose* would be $\forall\alpha.\mathsf{Set}\ (\alpha \to \mathsf{Set}\ (\alpha \to \mathsf{Set}\ \alpha))$ instead of $\forall\alpha.\mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha$. Therefore, we provided a second prelude for SaLT which provides exactly the same functions as the CuMin prelude with regard to non-determinism.

As an example, the two different SaLT implementations of *choose* are shown below.

$$
\begin{array}{l}
\mathit{chooseA} :: \forall\alpha.\mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha \\
\mathit{chooseA} = \lambda as :: \mathsf{Set}\ \alpha.\ \lambda bs :: \mathsf{Set}\ \alpha. \\
\quad \mathbf{unknown}_{\mathsf{Bool}} \gg\!= \lambda c :: \mathsf{Bool}.\ \mathbf{case}\ b\ \mathbf{of} \\
\qquad \mathsf{False} \to as \\
\qquad \mathsf{True}\ \to bs
\end{array}
$$

$$
\begin{array}{l}
\mathit{chooseB} :: \forall\alpha.\mathsf{Set}\ (\alpha \to \mathsf{Set}\ (\alpha \to \mathsf{Set}\ \alpha)) \\
\mathit{chooseB} = \{\lambda a :: \alpha.\ \{\lambda b :: \alpha. \\
\quad \mathbf{unknown}_{\mathsf{Bool}} \gg\!= \lambda c :: \mathsf{Bool}.\ \mathbf{case}\ b\ \mathbf{of} \\
\qquad \mathsf{False} \to \{\,a\,\} \\
\qquad \mathsf{True}\ \to \{\,b\,\}\}\}
\end{array}
$$

The second variant uses two singleton sets for the lambda abstractions when there really is just one possibility. *choose* 1 2 in CuMin is translated to $\mathit{chooseB} \gg\!= \lambda f :: (\alpha \to \mathsf{Set}\ (\alpha \to \mathsf{Set}\ \alpha)).\ f\ 1 \gg\!= \lambda g :: (\alpha \to \mathsf{Set}\ \alpha).\ g\ 2$ using the formal translation procedure from [14]. It can be rewritten to $\mathit{chooseA}\ \{1\}\ \{2\}$ using the variant implemented in the native SaLT prelude.

For applying a function to the elements of a set, $(\gg\!=)$ can be used in the following way.

$$
\begin{array}{l}
\mathit{sMap} :: \forall\alpha\ \beta.(\alpha \to \beta) \to \mathsf{Set}\ \alpha \to \mathsf{Set}\ \beta \\
\mathit{sMap} = \lambda f :: (\alpha.\ \beta) \to \\
\quad \lambda xs :: \mathsf{Set}\ \alpha.\ xs \gg\!= \lambda x :: \alpha.\ \{f\ x\}
\end{array}
$$

Each element of the input set is transformed using the function and then wrapped in a singleton set. The type signatures of *sMap* and $(\gg\!=)$ are actually quite similar, if the arguments are swapped: $\mathit{flip}\ \mathit{sMap} :: \forall\alpha\ \beta.\mathsf{Set}\ \alpha \to (\alpha \to \beta) \to \mathsf{Set}\ \beta$. The only difference is in the higher-order argument, which returns a set for $(\gg\!=)$ and just a single element for *sMap*.

Chapter 3

Denotational Semantics

In this chapter, I will describe the denotational semantics of CuMin and SaLT originally presented in [14] and highlight the required modifications to handle the deviations from the initial language specification.

3.1 Motivation for Denotational Semantics

Denotational semantics allows reasoning about functional programs by leveraging mathematical objects [16]. For example, the Haskell type Int could be interpreted as the set of integers $\mathbb{Z}$ and each value of type Int would then be an element of $\mathbb{Z}$. However, since Haskell allows unrestricted recursion, it is possible to write a definition similar to the following.

$$\begin{aligned} &\mathit{diverging} :: \mathsf{Int} \\ &\mathit{diverging} = \mathit{diverging} \end{aligned}$$

Since *diverging* never terminates, it is clearly not an integer value. Therefore, an additional value $\bot$ (*bottom*) is introduced, denoting non-terminating or failed computations. This bottom value is an inhabitant of each type. The correct interpretation of the type Int is $\{\bot\} \cup \mathbb{Z}$. The mapping from syntax to semantics is usually denoted by double square brackets. Hence, the semantics of Int can be formally written as $[\![\mathsf{Int}]\!] = \{\bot\} \cup \mathbb{Z}$. Using bottom, it is now possible to assign a value to non-terminating expressions, for example $[\![\mathit{diverging}]\!] = \bot$.

$$\begin{aligned} &\mathit{double} :: \mathsf{Int} \to \mathsf{Int} \\ &\mathit{double}\ x = x + x \end{aligned}$$

The *double* function has to be represented by a mathematical function $f : [\![\mathsf{Int}]\!] \to [\![\mathsf{Int}]\!]$. Since the function body of *double* uses its argument, *double e* terminates if and only if e terminates. This yields to the following definition for f.

$$f(n) = \begin{cases} \bot & \text{if } n = \bot \\ n + n & \text{if } n \in \mathbb{Z} \end{cases}$$

Using the above findings, the denotational semantics can now be used to derive the result of the computation *double* (*double* 2).

$$\begin{aligned}
&[\![\mathit{double}\ (\mathit{double}\ 2)]\!] \\
=&[\![\mathit{double}]\!]([\![\mathit{double}\ 2]\!]) \\
=&[\![\mathit{double}]\!]([\![\mathit{double}]\!]([\![2]\!])) \\
=&f(f(2)) = 4
\end{aligned}$$

The following sections will give a more rigorous mathematical definition for the denotational semantics of CuMin and SaLT, additionally featuring non-determinism.

3.2 Mathematical Foundations

Before presenting the actual semantics for CuMin and SaLT, I will give an introduction to the necessary mathematical foundations, covering different notions of definedness, and some set-theoretic background.

3.2.1 Definedness Relation

When an expression evaluates to bottom, it is also said to be *undefined*. In the above example, either an expression was defined, i.e. its value was an actual integer, or undefined. In the presence of algebraic data types, there are more nuances of definedness. Consider the expression Just *diverging*. The non-terminating expression is an argument to the constructor, but because of laziness, the expression as whole terminates. A program could distinguish Just and Nothing values, even though the argument of Just might be diverging. Hence, $\bot$ is said to be less defined than Just $\bot$ which is turn is less defined than Just 1.

This relation forms a partial order[1] and is characterized by the following definition 3.1. The statement $\mathbf{a} \sqsubseteq \mathbf{b}$ means than $\mathbf{a}$ is less defined than or equal to $\mathbf{b}$. The definedness relation given by [14] does not cover arbitrary algebraic data types and the relation presented here is a generalization of the original definition as shown by lemma 3.2.

Usually, the definedness relation is given separately for each type, but as inhabitants of different types (except for $\bot$) are incomparable (regarding definedness) anyway, the following definition extends to the whole universe of type inhabitants. This is also closer to the implementation of the interpreter in the next chapter, which does not preserve type information.

Definition 3.1. The definedness relation $\sqsubseteq$ is the smallest partial order satisfying the following conditions.

1. $\bot \sqsubseteq n$ for all $n \in \mathbb{N}$,
2. for all constructors C
   a) $C\ x_1\ \ldots\ x_n \sqsubseteq C\ x'_1\ \ldots\ x'_n$ if and only if $\forall i \in \{1, \ldots, n\}\ .x_i \sqsubseteq x'_i$

[1] A partial order $(P, \sqsubseteq)$ is a reflexive, antisymmetric and transitive relation $\sqsubseteq$ on a set P.

b) $\bot \sqsubseteq C\ x_1\ \ldots\ x_n$ for any $x_1, \ldots, x_n$,

3. given two functions f, g with $\mathrm{dom}(f) = \mathrm{dom}(g)$, $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in \mathrm{dom}(f)$ (pointwise comparison).

Lemma 3.2. *The partial order $\sqsubseteq$ defined above is a generalization of the partial order defined in [14] (denoted by $\sqsubseteq'$ in the following proof).*

Proof. For natural numbers and functions this is clear, since those parts where not changed.

For booleans, $\sqsubseteq' = \{(\bot, \bot), (\bot, \mathsf{False}), (\bot, \mathsf{True}), (\mathsf{False}, \mathsf{False}), (\mathsf{True}, \mathsf{True})\}$ is also the smallest partial order satisfying $\bot \sqsubseteq \mathsf{False}$, $\bot \sqsubseteq \mathsf{True}$, $\mathsf{False} \sqsubseteq \mathsf{False}$ and $\mathsf{True} \sqsubseteq \mathsf{True}$.

For pairs, the above definition corresponds to an entry-wise comparison with an additional least element $\bot$, since $\mathsf{Pair}\ x\ y \sqsubseteq \mathsf{Pair}\ x'\ y'$ if and only if $x \sqsubseteq x'$ and $y \sqsubseteq y'$. The definition for lists is equivalent since $\mathsf{Nil} \sqsubseteq \mathsf{Nil}$ and $\mathsf{Cons}\ x\ xs \sqsubseteq \mathsf{Cons}\ y\ ys$ whenever $x \sqsubseteq y$ and $xs \sqsubseteq ys$. □

3.2.2 Pointed Partially Ordered Sets

The non-determinism in CuMin and SaLT is represented by a special kind of sets, called *pointed partially ordered sets*, or *pointed posets* for short.

Each of the posets occuring in the semantics contains a unique least (w.r.t. $\sqsubseteq$) element denoted $\bot$. Hence, any such poset is also pointed, but I will adapt the terminology from the paper and simply write *poset*. A function between posets is said to be *monotone* if it is order preserving, and *strict* if it preserves the least element.

3.2.3 Lower Subsets

A non-empty subset $A \subseteq P$ of a poset is a *lower set* if for all $x \in A$ and $y \in P$, $y \in A$ whenever $y \sqsubseteq x$, i.e. when it is $\sqsubseteq$-downards closed. The set $\mathcal{L}(P)$ of all lower sets of a poset P is partially ordered by inclusion with $\{\bot\}$ (the singleton set containing $\bot$) being the least element. Hence, $(\mathcal{L}(P), \subseteq)$ is a pointed partially ordered set by itself. The *closure* of some $x \in P$ is denoted by $\downarrow x = \{y \in P \mid y \sqsubseteq x\}$. Note that the $(\downarrow\cdot) : P \to \mathcal{L}(P)$ function is monotone and strict. The $\sqsubseteq$-downwards closure of some nonempty subset $A \subseteq P$ of a poset P is denoted by $\mathrm{cl}(A) = \bigcup_{x \in A} \downarrow x$.

The following lemma shows that in some cases, the constraint of always using lower sets in the denotational semantic can be loosened. This will facilitate some proofs in subsequent sections and is the foundation of the inner workings of the interpreter described in the next chapter.

Lemma 3.3. *Suppose that P_1 and P_2 are posets and $f : P_1 \to \mathcal{L}(P_2)$ is a monotone function.*

1. $\bigcup_{x \in \mathrm{cl}(A)} f(x) = \bigcup_{x \in A} f(x)$ *for all nonempty* $A \subseteq P_1$
2. $\bigcup_{y \in \downarrow x} f(y) = f(x)$ *for all* $x \in P_1$.

Proof. 1. Let $\emptyset \neq A \subseteq P_1$ and $y \in \bigcup_{x \in \mathrm{cl}(A)} f(x)$. Then there is some $x \in \mathrm{cl}(A)$ such that $y \in f(x)$. By definition of cl, there is a $z \in A$ with $x \sqsubseteq z$. Hence, $y \in \bigcup_{x \in A} f(x)$ by monotonicity of f. The other direction follows directly from $A \subseteq \mathrm{cl}(A)$.

2. By applying 1. to $\{x\}$ for all $x \in P_1$.

□

3.3 Type Semantics

Before a meaningful interpretation can be assigned to terms, the posets consisting of the inhabitants of each type have to be defined. I will present here the simplified type semantics from [14] and give an example of how to handle regular recursive data types using a fixed-point construction. More complicated algebraic data types, for example containing function types or polymorphic recursion, require a more elaborate mathematical construction which lies beyond the scope of this thesis. An example of how to handle non-regular data types is the work of Johann and Ghani [11].

While the type semantics presented in this thesis does not cover all data types definable in our extension of the CuMin and SaLT languages, this does not affect the interpreters, as their value representation is untyped[2]. Moreover, in contrast to the original definition, the term semantics for logic variables does not refer to the type semantics. On the contrary, it would be an unnecessary restriction (from the implementation perspective) to disallow non-regular data types.

Definition 3.4 (Denotational type semantics of CuMin)**.** The semantics of some type τ is given by $[\![\tau]\!]_\theta$ with respect to some type environment θ which is a mapping from type variables to *closed* types, in other words, types containing no free type variables.

Environments are written as a vector of assignments $[\overline{x_n \mapsto \tau_n}]$, or as an extension of an existing environment by an additional mapping $\theta[x \mapsto \tau']$. The empty environment is denoted by $\emptyset$. The substitution of free type variables in a type τ using an environment θ is written as $\tau\theta$, meaning that each free type variable α in τ is replaced with the type $\theta(\alpha)$ if the mapping exists.

$$[\![\alpha]\!]_\theta = [\![\theta(\alpha)]\!]_\emptyset$$

$$[\![\mathsf{Nat}]\!]_\theta = \{\bot\} \cup \mathbb{N}$$

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

As a demonstration of how recursive data types are handled, the following two data types are used.

data List a = Nil | Cons a (List a)
data Tree a = Leaf a | Node a (Tree a) (Tree a)

The defining equations utilize the fixed-point combinator $\mu S.f(S)$ which denotes the least fixed-point of some function f depending on S.

$$[\![\mathsf{List}\ \tau]\!]_\theta = \mu S.\,\{\bot, \mathbf{Nil}\} \cup \{\mathbf{Cons\ x\ xs} \mid \mathbf{x} \in [\![\tau]\!]_\theta, \mathbf{xs} \in S\}$$

$$[\![\mathsf{Tree}\ \tau]\!]_\theta = \mu S.\,\{\bot\} \cup \{\mathbf{Leaf\ x} \mid \mathbf{x} \in [\![\tau]\!]_\theta\} \cup \{\mathbf{Node\ x\ l\ r} \mid \mathbf{x} \in [\![\tau]\!]_\theta, \mathbf{l} \in S, \mathbf{r} \in S\}$$

[2] Actually, type information is preserved partially, but only used for presentation purposes.

As an example for the concept of fixed-points, consider the following set produced by the construction of $[\![\mathsf{List\ Nat}]\!]$.

$$\{\bot, \mathbf{Nil}, \mathbf{Cons}\ \bot\ \bot, \mathbf{Cons}\ \bot\ \mathbf{Nil}, \mathbf{Cons}\ \bot\ (\mathbf{Cons}\ \bot\ \bot), \ldots,$$
$$\mathbf{Cons}\ 0\ \bot, \mathbf{Cons}\ 0\ \mathbf{Nil}, \mathbf{Cons}\ 0\ (\mathbf{Cons}\ \bot\ \bot), \ldots$$
$$\vdots \quad \}$$

Definition 3.5 (Denotational type semantics of SaLT)**.** The type semantics for SaLT $[\![\tau]\!]_\theta$ is similar to the one for CuMin, apart from functions and the additional set type.

$$[\![\alpha]\!]_\theta = [\![\theta(\alpha)]\!]_\emptyset \qquad [\![\tau \to \tau']\!]_\theta = \{\mathbf{f} : [\![\tau]\!]_\theta \to [\![\tau']\!]_\theta \mid \mathbf{f} \text{ monotone}\}$$

$$[\![\mathsf{Nat}]\!]_\theta = \{\bot\} \cup \mathbb{N} \qquad [\![\mathsf{Set}\ \tau]\!]_\theta = \mathcal{L}\,([\![\tau]\!]_\theta)$$

$$[\![\mathsf{List}\ \tau]\!]_\theta = \mu S.\, \{\bot, \mathbf{Nil}\} \cup \{\mathbf{Cons\ x\ xs} \mid \mathbf{x} \in [\![\tau]\!]_\theta, \mathbf{xs} \in S\}$$

$$[\![\mathsf{Tree}\ \tau]\!]_\theta = \mu S.\, \{\bot\} \cup \{\mathbf{Leaf\ x} \mid \mathbf{x} \in [\![\tau]\!]_\theta\} \cup \{\mathbf{Node\ x\ l\ r} \mid \mathbf{x} \in [\![\tau]\!]_\theta, \mathbf{l} \in S, \mathbf{r} \in S\}$$

3.4 Generator Functions

For the interpreter, it is desirable to restrict sets resulting from the term semantics to be finite, if possible. In the original term semantics, logic variables introduce infinite sets due to using the type semantics. Instead, generator functions depending on a step index are used to approximate the type semantics using finite sets. The approach to implement logic variables in terms of non-determinism has first been described for Curry by Antoy and Hanus in [4]. Definition 3.6 presents the generator functions for types in Data. Due to the restriction to Data values, this formalization does not suffer the technical difficulties encountered when defining the type semantics. Afterwards, lemma 3.7 proves exemplarily that the generated sets are indeed an approximation for the given type semantics (see definition 3.4).

Definition 3.6 (Generator Functions)**.** The family of functions gen_τ is defined for $\tau \in \mathsf{Data}$ and produces an approximating lower set $\mathrm{gen}_\tau(\theta, i)$ for the type environment θ as explained before and some step index i restricting the size of the resulting sets. For all free type variables α in τ the mapping $\theta(\alpha)$ must exist and $\theta(\alpha) \in \mathsf{Data}$ is required.

$$\begin{aligned}
\mathrm{gen}_\alpha\ (\theta, i) &= \mathrm{gen}_{\theta(\alpha)}(\emptyset, i) \\
\mathrm{gen}_{\mathrm{Nat}}\ (\theta, i) &= \{\bot, 0\} \cup \{n \in \mathbb{N}_+ \mid \log_2 n < i\} \\
\mathrm{gen}_{\mathsf{D}\ \overline{\tau_m}}(\theta, 0) &= \{\bot\} \\
\mathrm{gen}_{\mathsf{D}\ \overline{\tau_m}}(\theta, i+1) &= \{\bot\} \cup \{\mathsf{C}\ x_1 \ldots x_n \mid \mathsf{C}\ \overline{\tau'_n} \text{ is constructor of } \mathsf{D}\ \overline{\alpha_m}, \\
&\qquad x_1 \in \mathrm{gen}_{\tau'_1}([\overline{\alpha_m \mapsto \tau_m\theta}], i), \ldots, x_n \in \mathrm{gen}_{\tau'_n}([\overline{\alpha_m \mapsto \tau_m\theta}], i)\}
\end{aligned}$$

The following example shows the sets produced by the generator function for the list of naturals, which already served as an example above, for the step indices one and two.

$$
\begin{aligned}
&\mathrm{gen}_{\mathsf{List\ Nat}}(\emptyset, 1) \\
=&\{\bot\} \cup \{\mathbf{Nil}\} \cup \{\mathbf{Cons\ x\ xs} \mid \mathbf{x} \in \mathrm{gen}_{\mathsf{Nat}}(\emptyset, 0), \mathbf{xs} \in \mathrm{gen}_{\mathsf{List\ Nat}}(\emptyset, 0)\} \\
=&\{\bot\} \cup \{\mathbf{Nil}\} \cup \{\mathbf{Cons\ x\ xs} \mid \mathbf{x} \in \{\bot, 0\}, \mathbf{xs} \in \{\bot\}\} \\
=&\{\bot, \mathbf{Nil}, \mathbf{Cons}\ \bot\ \bot, \mathbf{Cons}\ 0\ \bot\} \\
\subseteq&[\![\mathsf{List\ Nat}]\!]_\emptyset
\end{aligned}
$$

The set for the successor step index can be defined in terms of the previous one

$$
\begin{aligned}
&\mathrm{gen}_{\mathsf{List\ Nat}}(\emptyset, 2) \\
=&\{\bot\} \cup \{\mathbf{Nil}\} \cup \{\mathbf{Cons\ x\ xs} \mid \mathbf{x} \in \mathrm{gen}_{\mathsf{Nat}}(\emptyset, 1), \mathbf{x}s \in \mathrm{gen}_{\mathsf{List\ Nat}}(\emptyset, 1)\} \\
=&\{\bot\} \cup \{\mathbf{Nil}\} \cup \{\mathbf{Cons\ x\ xs} \mid \mathbf{x} \in \{\bot, 0, 1\}, \mathbf{x}s \in \mathrm{gen}_{\mathsf{List\ Nat}}(\emptyset, 1)\} \\
=&\{\bot, \mathbf{Nil}, \mathbf{Cons}\ \bot\ \bot, \mathbf{Cons}\ \bot\ \mathbf{Nil}, \mathbf{Cons}\ \bot\ (\mathbf{Cons}\ \bot\ \bot), \mathbf{Cons}\ \bot\ (\mathbf{Cons}\ 0\ \bot), \\
&\quad \mathbf{Cons}\ 0\ \bot, \mathbf{Cons}\ 0\ \mathbf{Nil}, \mathbf{Cons}\ 0\ (\mathbf{Cons}\ 0\ \bot), \mathbf{Cons}\ 0\ (\mathbf{Cons}\ 0\ \bot), \\
&\quad \mathbf{Cons}\ 1\ \bot, \mathbf{Cons}\ 1\ \mathbf{Nil}, \mathbf{Cons}\ 1\ (\mathbf{Cons}\ 1\ \bot), \mathbf{Cons}\ 1\ (\mathbf{Cons}\ 1\ \bot)\} \\
\subseteq&[\![\mathsf{List\ Nat}]\!]_\emptyset
\end{aligned}
$$

The following lemma 3.7 supports the claim that the generator functions indeed approximate the type semantics.

Lemma 3.7. *The type semantics denoted by* $[\![\tau]\!]_\theta$ *is approximated by the generator functions* gen_τ *for* $\tau \in \mathsf{Data}$ *if for all free type variables* α *in* τ *the mapping* $\theta(\alpha)$ *exists and* $\theta(\alpha) \in \mathsf{Data}$*.*

$$[\![\tau]\!]_\theta = \bigcup_{i \in \mathbb{N}} \mathrm{gen}_\tau(\theta, i)$$

Proof. By structural induction on Data types and the environment θ being empty or not. Specifically, either the type argument or the environment is structurally reduced in each inductive case.

If $\tau = \alpha$ By assumption, $\theta(\alpha)$ exists, is closed and is a Data type. The inductive hypothesis can be applied for the empty environment, yielding $[\![\alpha]\!]_\theta = [\![\theta(\alpha)]\!]_\emptyset = \bigcup_{i \in \mathbb{N}} \mathrm{gen}_{\theta(\alpha)}(\emptyset)(i)$.

If $\tau = \mathsf{Nat}$ By definition, $\bigcup_{i \in \mathbb{N}} \mathrm{gen}_{\mathsf{Nat}}(\theta, i) \subseteq [\![\mathsf{Nat}]\!]_\theta$.

On the other hand, $n \in \mathrm{gen}_{\mathsf{Nat}}(\theta', \lceil \log_2(n+1) \rceil)$ for each $n \in \mathbb{N}$. Hence, the two interpretations are equal.

If $\tau = \mathsf{List}\ \tau'$ Proof by structural induction on list constructors using the inductive hypothesis, stating that for all lists $\mathbf{l}$, $\mathbf{l} \in [\![\mathsf{List}\ \tau']\!]_\theta \iff \exists j \in \mathbb{N}.\mathbf{l} \in \mathrm{gen}_{\mathsf{List}\ \tau'}(\theta, j)$.

If $\mathbf{l} = \mathbf{Nil}$: By definition, $\mathbf{Nil} \in [\![\mathsf{List}\ \tau']\!]_\theta$ and $\mathbf{Nil} \in \mathrm{gen}_{\mathsf{List}\ \tau'}(\theta, 1)$.

If $\mathbf{l} = \mathbf{Cons\ x\ xs}$: Suppose that $\mathbf{l} \in [\![\mathsf{List}\ \tau']\!]_\theta$. By definition, then also $\mathbf{xs} \in [\![\mathsf{List}\ \tau']\!]_\theta$. Let $j \in \mathbb{N}$ such that $\mathbf{xs} \in \mathrm{gen}_{\mathsf{List}\ \tau'}(\theta, j)$ as given by the inner inductive hypothesis. By the outer inductive hypothesis, $[\![\tau']\!]_\theta = \bigcup_{i \in \mathbb{N}} \mathrm{gen}_{\tau'}(\theta, i)$.

$$\begin{aligned}
[\![x]\!]^i_{\theta,\sigma} &= \downarrow\sigma(x) & [\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]^i_{\theta,\sigma} &= \bigcup_{\mathbf{x}\in[\![e_1]\!]^i_{\theta,\sigma}} [\![e_2]\!]^i_{\theta,\sigma[x\mapsto\mathbf{x}]} \\
[\![n]\!]^i_{\theta,\sigma} &= \downarrow\mathbf{n} & [\![\textbf{let } x :: \tau \textbf{ free in } e]\!]^i_{\theta,\sigma} &= \bigcup_{\mathbf{x}\in\mathrm{gen}_\tau(\theta,i)} [\![e]\!]^i_{\theta,\sigma[x\mapsto\mathbf{x}]} \\
[\![\textbf{failed}_\tau]\!]^i_{\theta,\sigma} &= \{\bot\} & [\![e_1\ e_2]\!]^i_{\theta,\sigma} &= \bigcup_{\mathbf{f}\in[\![e_1]\!]^i_{\theta,\sigma}} \bigcup_{\mathbf{x}\in[\![e_2]\!]^i_{\theta,\sigma}} \mathbf{f}\ \mathbf{x}
\end{aligned}$$

$$[\![e_1 + e_2]\!]^i_{\theta,\sigma} = \left\{\mathbf{a} +^\bot \mathbf{b} \mid \mathbf{a} \in [\![e_1]\!]^i_{\theta,\sigma}, \mathbf{b} \in [\![e_2]\!]^i_{\theta,\sigma}\right\}$$

$$[\![e_1 \equiv e_2]\!]^i_{\theta,\sigma} = \left\{\mathbf{a} \equiv \mathbf{b} \mid \mathbf{a} \in [\![e_1]\!]^i_{\theta,\sigma}, \mathbf{b} \in [\![e_2]\!]^i_{\theta,\sigma}\right\}$$

$$[\![\mathsf{C}_{\overline{\tau_m}}]\!]^i_{\theta,\sigma} = \downarrow(\lambda\,\mathbf{x}_1.\cdots\downarrow(\lambda\,\mathbf{x}_n.\downarrow(\mathbf{C}\ \mathbf{x}_1\ \ldots\ \mathbf{x}_n))\cdots)$$

for each constructor $\mathsf{C}\ \overline{\tau_n}$ in P

$$\begin{aligned}
[\![f_{\overline{\tau_m}}]\!]^0_{\theta,\sigma} &= \{\bot\} \\
[\![f_{\overline{\tau_m}}]\!]^{i+1}_{\theta,\sigma} &= \downarrow\left(\lambda\,\mathbf{a}_1.\cdots\downarrow\left(\lambda\,\mathbf{a}_n.[\![e]\!]^i_{[\overline{\alpha_m\mapsto\tau_m\theta}],[\overline{x_n\mapsto\mathbf{a_n}}]}\right)\cdots\right)
\end{aligned}$$

with $f :: \forall\alpha_1 \ldots \alpha_m.(\ldots) \Rightarrow \tau; f\ x_1 \ldots x_n = e$ in P

$$[\![\textbf{case } e \textbf{ of } \{\overline{Alt_k}\}]\!]^i_{\theta,\sigma} = \bigcup_{\mathbf{x}\in[\![e]\!]^i_{\theta,\sigma}} \begin{cases}
\bot & \text{if } \mathbf{x} = \bot \text{ or } \mathbf{x} \text{ matches no pattern} \\
[\![e']\!]^i_{\theta,\sigma[\overline{x_n\mapsto\mathbf{a}_n}]} & \text{if } \mathbf{x} = \mathbf{C}\ \overline{\mathbf{a}_n} \text{ and matches } (\mathsf{C}\ \overline{x_n} \to e') \text{ for some constructor } \mathsf{C}\ \overline{\tau_n} \text{ in } P \\
[\![e']\!]^i_{\theta,\sigma[x\mapsto a]} & \text{if } \mathbf{x} \text{ matches } (x \to e') \text{ and no other alternatives}
\end{cases}$$

Figure 3.1: Denotational term semantics for CuMin (adapted from [14])

Hence, there is a $j' \in N$ such that $\mathbf{x} \in \mathrm{gen}_{\tau'}(\theta, j')$. Then $\mathbf{Cons\ x\ xs} \in \mathrm{gen}_{\mathsf{List}\ \tau'}(\theta, \max\{j, j'\} + 1)$.
Now suppose that there is some $0 < j \in \mathbb{N}$ such that $\mathbf{l} \in \mathrm{gen}_{\mathsf{List}\ \tau'}(\theta, j)$. By definition, $\mathbf{x} \in \mathrm{gen}_{\tau'}(\theta, j-1)$ and $\mathbf{xs} \in \mathrm{gen}_{\mathsf{List}\ \tau'}(\theta, j-1)$. By the inductive hypotheses, $\mathbf{x} \in [\![\tau']\!]_\theta$ and $\mathbf{xs} \in [\![\mathsf{List}\ \tau']\!]_\theta$
Let f denote the function such that $[\![\mathsf{List}\ \tau']\!]_\theta = \mu S.f(S)$. Then $\mathbf{Cons\ x\ xs} \in f([\![\mathsf{List}\ \tau']\!]_\theta)$. Hence, $\mathbf{Cons\ x\ xs} \in [\![\mathsf{List}\ \tau']\!]_\theta$ because $[\![\mathsf{List}\ \tau']\!]_\theta$ is a fixed-point of f.

otherwise The equivalence can be shown analogously for other recursive data types.

□

3.5 Term Semantics

After having defined the meaning of types, I will now present the equations assigning meaningful values to terms. Since the semantics for CuMin and SaLT are similar, this

section mainly covers the term semantics for CuMin and then highlights the differences between them.

3.5.1 CuMin

The denotational term semantics $[\![e]\!]^i_{\theta,\sigma}$ of some well-typed expression e is defined in figure 3.1 with respect to some fixed and well-typed program P. In addition to the type environment θ explained above, the term semantics depend on the term environment σ which is a mapping from term variables to a single semantic value. The step index i restricts the depth of recursive calls, ensuring the semantics is well-defined and that all resulting sets are finite.

The unrestricted term semantics recovers unlimited recursion by taking the union of the term mapping with all possible step indices.

$$[\![e]\!]_{\theta,\sigma} = \bigcup_{i \in \mathbb{N}} [\![e]\!]^i_{\theta,\sigma}$$

The interpretation $[\![e]\!]^i_{\theta,\sigma}$ of an expression e is always a lower set due to the non-determinism inherent to CuMin. If the typing judgement $\Gamma \vdash e :: \tau$ holds, all type variables in Γ are mapped by θ to some closed type and $\sigma(x) \in [\![\tau']\!]_\theta$ for each term variable $x :: \tau'$ in Γ, then the semantic interpretation $[\![e]\!]^i_{\theta,\sigma}$ is a lower subset of $[\![\tau]\!]_\theta$ for all $i \in \mathbb{N}$.

The addition of two mathematical values is given by the *lifted* addition operator $+^\perp$ which evaluates to $\perp$ if one of its arguments is $\perp$ and to the sum of the arguments otherwise. The λ-notation on the right-hand side denotes an anonymous mathematical function, similar to lambda-abstractions in Haskell.

The denotation of $[\![e_1 \equiv e_2]\!]^i_{\theta,\sigma}$ relies on the following definition.

Definition 3.8. The $\equiv$-relation between symbolic values is defined to denote structural equality.

$$(\mathbf{a} \equiv \mathbf{b}) := \begin{cases} \perp & \text{if } \mathbf{a} = \perp \vee \mathbf{b} = \perp \\ \mathbf{a} = \mathbf{b} & \text{if } \mathbf{a}, \mathbf{b} \in \mathbb{N} \\ \mathbf{x}_1 \equiv \mathbf{y}_1 \wedge^\perp \cdots \wedge^\perp \mathbf{x}_n \equiv \mathbf{y}_n & \text{if } \mathbf{a} = \mathbf{C}\ \overline{\mathbf{x}_n} \wedge \mathbf{b} = \mathbf{C}\ \overline{\mathbf{y}_n} \\ & \text{for some constructor } \mathbf{C} \\ \mathbf{False} & \text{otherwise} \end{cases}$$

where the conjunction operator $\wedge^\perp$ is defined as follows.

$$(\mathbf{a} \wedge^\perp \mathbf{b}) := \begin{cases} \mathbf{False} & \text{if } \mathbf{a} = \mathbf{False} \\ \mathbf{b} & \text{if } \mathbf{a} = \mathbf{True} \\ \perp & \text{if } \mathbf{a} = \perp \end{cases}$$

It is assumed to be left-associative and the empty conjunction is defined to be **True**.

For the most part, these equations are nearly identical to those from the underlying paper. In addition to the modifications needed to accommodate algebraic data types, logic variables are now resolved using the aforementioned generator functions, and constructors do not need to be fully applied and instead evaluate to nested lambda terms like functions.

$$
\begin{aligned}
[\![x]\!]^i_{\theta,\sigma} &= \sigma(x) & [\![\lambda x :: \tau.\ e]\!]^i_{\theta,\sigma} &= \lambda\,\mathbf{x}.[\![e]\!]^i_{\theta,\sigma[x\mapsto\mathbf{x}]} \\
[\![n]\!]^i_{\theta,\sigma} &= \mathbf{n} & [\![e_1 + e_2]\!]^i_{\theta,\sigma} &= [\![e_1]\!]^i_{\theta,\sigma} +^{\perp} [\![e_2]\!]^i_{\theta,\sigma} \\
[\![\mathbf{failed}_\tau]\!]^i_{\theta,\sigma} &= \{\perp\} & [\![e_1\ e_2]\!]^i_{\theta,\sigma} &= [\![e_1]\!]^i_{\theta,\sigma}[\![e_2]\!]^i_{\theta,\sigma} \\
[\![\mathbf{unknown}_\tau]\!]^i_{\theta,\sigma} &= \mathrm{gen}_\tau(\theta, i) & [\![e_1 \equiv e_2]\!]^i_{\theta,\sigma} &= [\![e_1]\!]^i_{\theta,\sigma} \equiv [\![e_2]\!]^i_{\theta,\sigma} \\
[\![\{\,e\,\}]\!]^i_{\theta,\sigma} &= {\downarrow}[\![e]\!]^i_{\theta,\sigma} & [\![e_1 \gg\!= e_2]\!]^i_{\theta,\sigma} &= \bigcup_{\mathbf{x}\in[\![e_1]\!]^i_{\theta,\sigma}} \left([\![e_2]\!]^i_{\theta,\sigma}\ \mathbf{x}\right)
\end{aligned}
$$

$$
\begin{aligned}
[\![f_{\overline{\tau_m}}]\!]^0_{\theta,\sigma} &= \perp & [\![\mathsf{C}_{\overline{\tau_m}}]\!]^i_{\theta,\sigma} &= \lambda\,\mathbf{x}_1.\cdots\lambda\,\mathbf{x}_n.\mathbf{C}\ \mathbf{x}_1\ \ldots\ \mathbf{x}_n \\
[\![f_{\overline{\tau_m}}]\!]^{i+1}_{\theta,\sigma} &= [\![e]\!]^i_{\overline{[\alpha_m\mapsto\tau_m\theta]},\emptyset} & &\text{for each constructor } \mathsf{C}\ \overline{\tau_n} \text{ in } P \\
\multicolumn{4}{l}{\text{with } f :: \forall\alpha_1\ldots\alpha_m.(\ldots) \Rightarrow \tau; f\ x_1\ldots x_n = e \text{ in } P}
\end{aligned}
$$

$$
[\![\mathbf{case}\ e\ \mathbf{of}\ \{\overline{Alt_k}\}]\!]^i_{\theta,\sigma} = \begin{cases}
\perp & \text{if } [\![e]\!]^i_{\theta,\sigma} = \perp \text{ or } [\![e]\!]^i_{\theta,\sigma} \text{ matches no pattern} \\
[\![e']\!]^i_{\theta,\sigma[\overline{x_n\mapsto\mathbf{a}_n}]} & \text{if } [\![e]\!]^i_{\theta,\sigma} = \mathbf{C}\ \overline{\mathbf{a}_n} \text{ and matches } (\mathsf{C}\ \overline{x_n} \to e') \\
 & \text{for some constructor } \mathsf{C}\ \overline{\tau_n} \text{ in } P \\
[\![e']\!]^i_{\theta,\sigma[x\mapsto a]} & \text{if } [\![e]\!]^i_{\theta,\sigma} \text{ matches } (x \to e') \text{ and no} \\
 & \text{other alternatives}
\end{cases}
$$

Figure 3.2: Denotational term semantics for SaLT (adapted from [14])

As before, it is easy to see that instantiating the generalized rules to the special cases from the paper recovers the original equations. The following proof exemplarily shows this for the Cons constructor. The symbolic cons constructor is denoted by a colon.

Proof.

$$
\begin{aligned}
[\![\mathsf{Cons}_\tau\ x\ xs]\!]^i_{\theta,\sigma} &= \bigcup_{\mathbf{f}\in[\![\mathsf{Cons}_\tau\ x]\!]^i_{\theta,\sigma}} \bigcup_{\mathbf{xs}\in[\![xs]\!]^i_{\theta,\sigma}} \mathbf{f}\ \mathbf{xs} \\
&= \cdots = \bigcup_{\mathbf{g}\in[\![\mathsf{Cons}_\tau]\!]^i_{\theta,\sigma}} \bigcup_{\mathbf{x}\in[\![x]\!]^i_{\theta,\sigma}} \bigcup_{\mathbf{f}\in\mathbf{g}\ \mathbf{x}} \bigcup_{\mathbf{xs}\in[\![xs]\!]^i_{\theta,\sigma}} \mathbf{f}\ \mathbf{xs} \\
&= \bigcup_{\mathbf{x}\in[\![x]\!]^i_{\theta,\sigma}} \bigcup_{\mathbf{f}\in{\downarrow}(\lambda\,\mathbf{t}.{\downarrow}(\mathbf{x}:\mathbf{t}))} \bigcup_{\mathbf{xs}\in[\![xs]\!]^i_{\theta,\sigma}} \mathbf{f}\ \mathbf{xs} \\
&= \bigcup_{\mathbf{x}\in[\![x]\!]^i_{\theta,\sigma}} \bigcup_{\mathbf{xs}\in[\![xs]\!]^i_{\theta,\sigma}} {\downarrow}(\mathbf{x}:\mathbf{xs}) \\
&= \{\perp\} \cup \left\{\mathbf{x}:\mathbf{xs} \mid \mathbf{x}\in[\![x]\!]^i_{\theta,\sigma}, \mathbf{xs}\in[\![xs]\!]^i_{\theta,\sigma}\right\}
\end{aligned}
$$

The third and fourth equations are derived using lemma 3.3. The last equation follows from the fact that $[\![x]\!]^i_{\theta,\sigma}$ and $[\![xs]\!]^i_{\theta,\sigma}$ are already lower sets by themselves. □

3.5.2 SaLT

Again, the equations for the denotational term semantics $[\![e]\!]^i_{\theta,\sigma}$ of well-typed SaLT terms e in figure 3.2 are defined with respect to some fixed well-typed program P. In contrast

to the denotational semantics of CuMin, the interpretation of SaLT terms only relies on lower sets for the set-typed expressions $\{\, e \,\}$ and $e_1 \gg\!= e_2$.

Since the lower sets are necessarily required for limit processes, the following definition of the unrestricted term semantics is only valid for set typed expressions.

$$[\![e]\!]_{\theta,\sigma} = \bigcup_{i\in\mathbb{N}} [\![e]\!]^i_{\theta,\sigma}$$

If the typing judgement $\Gamma \vdash e :: \tau$ holds, all type variables in Γ are mapped by θ to some closed type and $\sigma(x) \in [\![\tau']\!]_\theta$ for each term variable $x :: \tau'$ in Γ, then the semantic interpretation $[\![e]\!]^i_{\theta,\sigma} \in [\![\tau]\!]_\theta$ for all $i \in \mathbb{N}$.

In general, the semantics of CuMin and SaLT are equivalent for all CuMin programs and their translations to SaLT as given by [14]. Since SaLT makes non-determinism explicit, it provides some opportunities to remove unnecessary non-determinism, for example when there is only one maximal element in a lower set.

3.6 Examples and Applications

In this section I present some examples of how the denotational term semantics can be applied to prove program properties.

3.6.1 Set Union

The *choose* function has been suggestively named (∪) in [14], and as shown below, it indeed evaluates to the union of the two arguments.

Lemma 3.9. $[\![choose_\tau\ e_1\ e_2]\!]^i_{\theta,\sigma} = [\![e_1]\!]^i_{\theta,\sigma} \cup [\![e_2]\!]^i_{\theta,\sigma}$ *for* $i \geqslant 2$ *where choose is defined in the prelude as follows.*

$$\begin{aligned} &choose :: \forall\alpha.\alpha \to \alpha \to \alpha \\ &choose\ x\ y = \mathbf{let}\ c :: \mathsf{Bool}\ \mathbf{free} \\ &\quad \mathbf{in\ case}\ c\ \mathbf{of}\ \{\mathsf{False} \to x; \mathsf{True} \to y\} \end{aligned}$$

Proof. Suppose $i \geqslant 1$, then

$$\begin{aligned} &[\![choose_\tau\ e_1\ e_2]\!]^{i+1}_{\theta,\sigma} \\ =\ &\bigcup_{\mathbf{f}\in[\![choose_\tau\ e_1]\!]^{i+1}_{\theta,\sigma}} \bigcup_{\mathbf{b}\in[\![e_2]\!]^{i+1}_{\theta,\sigma}} \mathbf{f\ b} \\ =\ &\bigcup_{\mathbf{g}\in[\![choose_\tau]\!]^{i+1}_{\theta,\sigma}} \bigcup_{\mathbf{a}\in[\![e_1]\!]^{i+1}_{\theta,\sigma}} \bigcup_{\mathbf{f}\in\mathbf{g}\ \mathbf{a}} \bigcup_{\mathbf{b}\in[\![e_2]\!]^{i+1}_{\theta,\sigma}} \mathbf{f\ b} \end{aligned}$$

applying lemma 3.3 twice and defining $\theta' := [\alpha \mapsto \tau]$ allows to reduce this to

$$\begin{aligned} =\ &\bigcup_{\mathbf{a}\in[\![e_1]\!]^{i+1}_{\theta,\sigma}} \bigcup_{\mathbf{b}\in[\![e_2]\!]^{i+1}_{\theta,\sigma}} [\![\mathbf{let}\ c :: \mathsf{Bool}\ \mathbf{free\ in\ case}\ c\ \mathbf{of}\ \{\ldots\}]\!]^i_{\theta',[x\mapsto\mathbf{a},y\mapsto\mathbf{b}]} \\ =\ &\bigcup_{\mathbf{a}\in[\![e_1]\!]^{i+1}_{\theta,\sigma}} \bigcup_{\mathbf{b}\in[\![e_2]\!]^{i+1}_{\theta,\sigma}} \bigcup_{\mathbf{x}\in\mathrm{gen}_{\mathsf{Bool}}(\theta',i)} [\![\mathbf{case}\ c\ \mathbf{of}\ \{\mathsf{False} \to x; \mathsf{True} \to y\}]\!]^i_{\theta',[x\mapsto\mathbf{a},y\mapsto\mathbf{b},c\mapsto\mathbf{x}]} \end{aligned}$$

again by lemma 3.3 and by the definition of of $[\![\mathbf{case}\ldots\mathbf{of}\ldots]\!]^i_{\theta,\sigma}$

$$
\begin{aligned}
&= \bigcup_{\mathbf{a}\in[\![e_1]\!]^{i+1}_{\theta,\sigma}} \bigcup_{\mathbf{b}\in[\![e_2]\!]^{i+1}_{\theta,\sigma}} \left([\![x]\!]^i_{\theta',[x\mapsto\mathbf{a},y\mapsto\mathbf{b},c\mapsto\mathbf{False}]} \cup [\![y]\!]^i_{\theta',[x\mapsto\mathbf{a},y\mapsto\mathbf{b},c\mapsto\mathbf{True}]}\right)\\
&= \bigcup_{\mathbf{a}\in[\![e_1]\!]^{i+1}_{\theta,\sigma}} [\![x]\!]^i_{\theta',[x\mapsto\mathbf{a}]} \cup \bigcup_{\mathbf{b}\in[\![e_2]\!]^{i+1}_{\theta,\sigma}} [\![y]\!]^i_{\theta',[y\mapsto\mathbf{b}]}\\
&= \bigcup_{\mathbf{a}\in[\![e_1]\!]^{i+1}_{\theta,\sigma}} \downarrow\mathbf{a} \cup \bigcup_{\mathbf{b}\in[\![e_2]\!]^{i+1}_{\theta,\sigma}} \downarrow\mathbf{b}\\
&= [\![e_1]\!]^{i+1}_{\theta,\sigma} \cup [\![e_2]\!]^{i+1}_{\theta,\sigma}
\end{aligned}
$$

□

Analogously, the binary choice combinator from the SaLT prelude behaves like set union.

Lemma 3.10. $[\![choose_\tau\ e_1\ e_2]\!]^i_{\theta,\sigma} = [\![e_1]\!]^i_{\theta,\sigma} \cup [\![e_2]\!]^i_{\theta,\sigma}$ *for all* $e_1, e_2 :: \{\tau\}$ *and all* $i \geqslant 2$ *where choose is defined as*

$$
\begin{aligned}
&choose :: \forall\alpha.\mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha\\
&choose = \lambda xs :: \mathsf{Set}\ \alpha.\ \lambda ys :: \mathsf{Set}\ \alpha.\\
&\quad \mathbf{unknown}_{\mathsf{Bool}} \gg\!= \lambda c :: \mathsf{Bool}.\\
&\qquad \mathbf{case}\ c\ \mathbf{of}\ \{\mathsf{False} \to xs; \mathsf{True} \to ys\}
\end{aligned}
$$

Proof. By equational reasoning similar to lemma 3.9. □

Note that the formal translation of the CuMin *choose* function has the type $\forall\alpha.\mathsf{Set}\ (\alpha \to \mathsf{Set}\ (\alpha \to \mathsf{Set}\ \alpha))$, whereas the definition above has the type $\forall\alpha.\mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha \to \mathsf{Set}\ \alpha$. Yet, as shown by the lemmas, both function actually behave the same in that they both return the union of their arguments when applied to non-deterministic expressions.

3.6.2 Impact of Step Indices

The impact of the step index on a computation is best shown using a recursive function like the following implementation of *length*.

$$
\begin{aligned}
&length :: \forall a.\mathsf{List}\ a \to \mathsf{Nat}\\
&length\ xxs = \mathbf{case}\ xxs\ \mathbf{of}\\
&\quad \mathsf{Nil} \to 0\\
&\quad \mathsf{Cons}\ x\ xs \to 1 + length_a\ xs
\end{aligned}
$$

When the step index is too low, the function fails to produce a result at all.

$$
\begin{aligned}
& [\![length_{\mathsf{Nat}}[1,2,3]_{\mathsf{Nat}}]\!]^1_{\theta,\sigma} \\
= \cdots = & \bigcup_{\mathbf{f} \in [\![length_{\mathsf{Nat}}]\!]^1_{\theta,\sigma}} (\mathbf{f}\ (\mathbf{Cons}\ 1\ (\mathbf{Cons}\ 2\ (\mathbf{Cons}\ 3\ \mathbf{Nil})))) \\
= \cdots = & [\![\mathbf{case}\ xxs\ \mathbf{of}\ \ldots]\!]^0_{[a \mapsto \mathsf{Nat}],[xss \mapsto \mathbf{Cons}\ 1\ (\mathbf{Cons}\ 2\ (\mathbf{Cons}\ 3\ \mathbf{Nil}))]} \\
= \cdots = & [\![1 + length_a xs]\!]^0_{[a \mapsto \mathsf{Nat}],[x \mapsto 1, xs \mapsto \mathbf{Cons}\ 2\ (\mathbf{Cons}\ 3\ \mathbf{Nil}), xss \mapsto \mathbf{Cons}\ 1\ (\ldots)]} \\
= \cdots = & \left\{ \mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a} \in \{\perp, 1\}, \mathbf{b} \in [\![length_a xs]\!]^0_{\ldots} \right\} \\
= \cdots = & \left\{ \mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a} \in \{\perp, 1\}, \mathbf{b} \in \{\perp\} \right\} = \{\perp\}
\end{aligned}
$$

A length function using Peano numbers[3] would be able to produce a partial result before failing. However, for all $i > 3$, it can been shown that

$$[\![length_{\mathsf{Nat}}[1,2,3]_{\mathsf{Nat}}]\!]^i_{\theta,\sigma} = [\![length_{\mathsf{Nat}}[1,2,3]_{\mathsf{Nat}}]\!]^{i+1}_{\theta,\sigma} = \{\perp, 3\}$$

and the infinite union $[\![length_{\mathsf{Nat}}[1,2,3]_{\mathsf{Nat}}]\!]_{\theta,\sigma} = \bigcup_{i \in \mathbb{N}} [\![length_{\mathsf{Nat}}[1,2,3]_{\mathsf{Nat}}]\!]^i_{\theta,\sigma}$ collapses to the finite set $\{\perp, 3\}$.

3.6.3 Call-Time Choice

Another finding supported by the denotational semantics is the sharing of non-deterministic values as exerted by let-bindings and function applications. A term variable can only be bound to exactly one value before evaluating the corresponding subexpression. Since the term environment σ maps from term variables to a single value, references to variables always yield deterministic results.

As a demonstration, consider the two examples from listing 1.3. First note that by the above lemma $[\![coin]\!]^{i+2}_{\theta,\sigma} = [\![choose_{\mathsf{Nat}}\ 0\ 1]\!]^{i+1}_{\theta,\sigma} = \{\perp, 0, 1\}$. Now the semantics can be utilized to proof the claim from the first chapter concerning the different results of *dc1* and *dc2*.

$$
\begin{aligned}
& [\![dc1]\!]^{i+3}_{\theta,\sigma} = [\![coin + coin]\!]^{i+2}_{\emptyset,\emptyset} \\
= & \left\{ \mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a}, \mathbf{b} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset} \right\} \\
= & \left\{ \mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a}, \mathbf{b} \in \{\perp, 0, 1\} \right\} \\
= & \{\perp, 0, 1, 2\}
\end{aligned}
$$

$$
\begin{aligned}
& [\![dc2]\!]^{i+3}_{\theta,\sigma} = [\![\mathbf{let}\ x = coin\ \mathbf{in}\ x + x]\!]^{i+2}_{\emptyset,\emptyset} \\
= & \bigcup_{\mathbf{x} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} [\![x + x]\!]^{i+2}_{\emptyset,[x \mapsto \mathbf{x}]} \\
= & \bigcup_{\mathbf{x} \in \{\perp,0,1\}} \left\{ \mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a}, \mathbf{b} \in [\![x]\!]^{i+2}_{\emptyset,[x \mapsto \mathbf{x}]} \right\} \\
= & \bigcup_{\mathbf{x} \in \{\perp,0,1\}} \left\{ \mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a}, \mathbf{b} \in \downarrow[x \mapsto \mathbf{x}](x) \right\} \\
= & \bigcup_{\mathbf{x} \in \{\perp,0,1\}} \left\{ \mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a}, \mathbf{b} \in \{\perp, \mathbf{x}\} \right\} \\
= & \{\perp, 0, 2\}
\end{aligned}
$$

[3] Natural numbers consisting of some zero Z and a successor operation S, i.e. $2 = S(SZ)$

The addends in the first example can each take the value zero or one, leading to four combinations with three different outcomes (disregarding bottom values). In the second example, either both addends are zero or both are one, resulting in only two different outcomes.

To show how the semantics for CuMin and SaLT relate, the same example is given below for the SaLT translation of these two expressions. Now *coin* has the type Set Nat, and since the translation procedure preserves the semantics [14], $[\![coin]\!]^{i+2}_{\theta,\sigma} = [\![choose_{\mathsf{Nat}}\ 0\ 1]\!]^{i+1}_{\theta,\sigma} = \{\bot, 0, 1\}$ still holds.

$$
\begin{aligned}
&dc1 :: \mathsf{Set\ Nat} \\
&dc1 = coin \gg\!= \lambda a :: \mathsf{Nat}.\ coin \\
&\qquad\qquad \gg\!= \lambda b :: \mathsf{Nat}.\ \{\, a + b \,\}
\end{aligned}
\qquad\qquad
\begin{aligned}
&dc2 :: \mathsf{Nat} \\
&dc2 = coin \gg\!= \lambda c :: \mathsf{Nat}. \\
&\qquad\qquad (\lambda x :: \mathsf{Nat}.\ \{\, x + x \,\})\ c
\end{aligned}
$$

The following calculation with the SaLT semantics eventually yields the same results as the CuMin term semantics, albeit the SaLT syntax resembles the actual calculations much more closely due to the explicit non-determinism.

$$
\begin{aligned}
[\![dc1]\!]^{i+3}_{\theta,\sigma} &= [\![coin \gg\!= \lambda a :: \mathsf{Nat}.\ coin \gg\!= \lambda b :: \mathsf{Nat}.\ \{\, a + b \,\}]\!]^{i+2}_{\emptyset,\emptyset} \\
&= \bigcup_{\mathbf{a} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} [\![\lambda a :: \mathsf{Nat}.\ coin \gg\!= \lambda b :: \mathsf{Nat}.\ \{\, a + b \,\}]\!]^{i+2}_{\emptyset,\emptyset}\ \mathbf{a} \\
&= \cdots = \bigcup_{\mathbf{a} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} [\![coin \gg\!= \lambda b :: \mathsf{Nat}.\ \{\, a + b \,\}]\!]^{i+2}_{\emptyset,[a \mapsto \mathbf{a}]} \\
&= \cdots = \bigcup_{\mathbf{a} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} \bigcup_{\mathbf{b} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} [\![\{\, a + b \,\}]\!]^{i+2}_{\emptyset,[a \mapsto \mathbf{a}, b \mapsto \mathbf{b}]} \\
&= \cdots = \bigcup_{\mathbf{a} \in \{\bot,0,1\}} \bigcup_{\mathbf{b} \in \{\bot,0,1\}} \downarrow(\mathbf{a} + \mathbf{b}) \\
&= \{\bot, 0, 1, 2\}
\end{aligned}
$$

$$
\begin{aligned}
[\![dc2]\!]^{i+3}_{\theta,\sigma} &= [\![coin \gg\!= \lambda c :: \mathsf{Nat}.\ (\lambda x :: \mathsf{Nat}.\ \{\, x + x \,\})\ c]\!]^{i+2}_{\emptyset,\emptyset} \\
&= \bigcup_{\mathbf{c} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} [\![\lambda c :: \mathsf{Nat}.\ (\lambda x :: \mathsf{Nat}.\ \{\, x + x \,\})\ c]\!]^{i+2}_{\emptyset,\emptyset}\ \mathbf{c} \\
&= \cdots = \bigcup_{\mathbf{c} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} [\![(\lambda x :: \mathsf{Nat}.\ \{\, x + x \,\})\ c]\!]^{i+2}_{\emptyset,[c \mapsto \mathbf{c}]} \\
&= \cdots = \bigcup_{\mathbf{c} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} ([\![\lambda x :: \mathsf{Nat}.\ \{\, x + x \,\}]\!]^{i+2}_{\emptyset,[c \mapsto \mathbf{c}]} \mathbf{c}) \\
&= \cdots = \bigcup_{\mathbf{c} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} [\![\{\, x + x \,\}]\!]^{i+2}_{\emptyset,[c \mapsto \mathbf{c}, x \mapsto \mathbf{c}]} \\
&= \cdots = \bigcup_{\mathbf{c} \in [\![coin]\!]^{i+2}_{\emptyset,\emptyset}} \downarrow(\mathbf{c} + \mathbf{c}) \\
&= \{\bot, 0, 2\}
\end{aligned}
$$

Chapter 4

Implementing the Denotational Semantics

In this chapter I will outline my implementation of the denotational semantics of CuMin and SaLT described in the previous chapter. The first sections cover the general ideas which are important for both languages while the last section is dedicated to the language specific implementations. This separation extends to the implementation, which is subdivided into the three packages `denotational-funlogic`, `denotational-cumin` and `denotational-salt`. The functionality provided by the former is used by the language-specific implementations.

4.1 Modeling Denotational Semantics in Haskell

Before describing the architecture of the implementation, I will briefly describe how monads can be used to abstract over non-determinism in Haskell.

4.1.1 Non-Determinism as a Monad

Monads already occurred in the second chapter, in the implementation of the type checker. The code excerpt, shown again for reference, contains a **do**-block, which is actually syntactic sugar for monads.

```
go (ELet var e1 e2) = do
    varTy <- checkExp e1
    local (localScope.at var .~ Just varTy) (checkExp e2)
```

By desugaring, the definition can be rewritten using $(\gg\!=) :: \mathsf{Monad}\ m \Rightarrow m\ a \to (a \to m\ b) \to m\ b$, also called *bind*.

```
go (ELet var e1 e2) = checkExp e1
  >>= \varTy -> local (localScope.at var .~ Just varTy) (checkExp e2)
```

A monad consists of a type constructor together with the aforementioned bind operation, and a function called *return* $:: \mathsf{Monad}\ m \Rightarrow a \to m\ a$, which allows to wrap single values in the monad. As indicated by the type signatures, monads are implemented using the Monad type class.

Note that the bind operation has the same name as the indexed union operator in SaLT, because the non-determinism found in SaLT also forms a monad with singleton sets ($\{\cdot\} :: a \to \mathsf{Set}\ a$) as return and indexed unions ($(\gg=) :: \mathsf{Set}\ a \to (a \to \mathsf{Set}\ b) \to \mathsf{Set}\ b$) as bind.

The Haskell base library includes the MonadPlus type class for non-deterministic monads providing $mplus :: \mathsf{MonadPlus}\ m \Rightarrow m\ a \to m\ a \to m\ a$ for a choice between two alternatives. Technically, this would be enough for implementing the semantics, but most non-determinism monads implement $\gg=$ and *mplus* as depth-first search.

The basic non-deterministic monad is the list monad representing a choice between multiple outcomes. In this context, *return* x is just the singleton list $[x]$. The monadic bind operation $(\gg=) :: [a] \to (a \to [b]) \to [b]$ specialized to lists maps each element of the input list to multiple results, concatenating these intermediate lists to one list. The bind operation expresses a dependency of results, i.e. to produce the results, it needs to know the input elements. On the other hand, $mplus :: [a] \to [a] \to [a]$ (again specialized to lists) provides a choice between the results of two computations by concatenating them. The following example shows a function which non-deterministically increments its argument.

$$
\begin{aligned}
&maybeAdd :: \mathsf{Int} \to [\mathsf{Int}] \\
&maybeAdd\ n = [n, n+1]
\end{aligned}
$$

Using this function, $[1,3] \gg= maybeAdd$ evaluates to the concatenation of *maybeAdd* 1 and *maybeAdd* 3, which is $[1,2,3,4]$. Actually, *maybeAdd* could be generalized to work with any MonadPlus since it only uses the abstract concept of choice.

$$
\begin{aligned}
&maybeAddM :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{Int} \to m\ \mathsf{Int} \\
&maybeAddM\ n = mplus\ (return\ n)\ (return\ (n+1))
\end{aligned}
$$

The aforementioned list monad also exhibits depth-first search behavior, as the resulting list of $[1,3] \gg= maybeAdd$ first contains the elements produced by *maybeAdd* 1 and only then those produced by *maybeAdd* 3. Likewise, *mplus* implemented as concatenation returns the elements of the first list before the second.

Non-deterministic choice in the set monad in SaLT can be implemented in terms of language features, as demonstrated by the implementation for $choose :: \mathsf{Set}\ a \to \mathsf{Set}\ a \to \mathsf{Set}\ a$. The type of *choose* is the type of *mplus* specialized to Set.

4.1.2 General Architecture

The mapping from terms to the semantic domain depends on an environment consisting of the step index, the type variable mapping and the term variable mapping, and possibly yields non-deterministic results. This can be modeled in Haskell by using the ReaderT monad transformer[1] over some other monad n. The reader monad takes care of threading the environment through the computation and the inner monad provides the non-determinism. There is no canonical implementation of sets since the order of elements matters in Haskell, in particular when sets are infinite. In the latter case, the chosen search strategy determines if all elements of a set can be enumerated or not. Leaving the

[1] A *monad transformer* is a type constructor taking a monad and yielding another monad with combined effects [13].

inner monad unspecified allows plugging in the search strategy best suited to a particular problem.

The definition presented in listing 4.1 is abstracted over the non-determinism monad n, the value type *val* (i.e. the type of inhabitants of the model of the semantic domain) and the type of top-level bindings *bnd* (see section 2.1). The value type itself is parameterized over the non-determinism monad because in CuMin, the domain of function types is inhabited by functions mapping a value to a set of values, and there are explicit set-values in SaLT. Hence, both concrete value types need to refer to the non-determinism monad.

```
data EvalEnv bnd val n
  = EvalEnv
  { _termEnv     :: Map VarName (val n)
  , _typeEnv     :: Map TVName Type
  , _moduleEnv   :: CoreModule bnd
  , _constrEnv   :: Map DataConName TyDecl
  , _stepIdx     :: StepIndex
  , _pruningImpl :: n (val n) -> n (val n)
  }
type Eval bnd val n = ReaderT (EvalEnv bnd val n) n
```

Listing 4.1: Interpreter Environment

In most cases, being fully polymorphic in the value representation *val* does not provide enough structure for some functions. While the models of semantic domains of CuMin and SaLT differ in function and set values, they have bottoms, natural numbers and data constructors in common. The Value type class shown in listing 4.2 can be used to construct these parts of the value models while sill being polymorphic over the value.

```
class Value (v :: (* -> *) -> *) where
  naturalValue :: Integer -> v n
  bottomValue  :: String -> v n
  dataValue    :: DataConName -> [v n] -> [Type] -> v n
```

Listing 4.2: Interface for the concrete implementations of the semantic domain

In addition to the finite step indices, the interpreter provides a mode where it evaluates expressions without restriction on the recursion depth. The StepIndex type can either take a finite (non-negative) value StepNatural n, or the infinite value StepInfinite. The interpreter relies on two functions operating on step indices. The first is *decrement* :: StepIndex $\rightarrow$ StepIndex which indeed decrements the value in the finite case (where zero is a lower bound), but leaves an infinite value unchanged. The second one is *isZero*::StepIndex $\rightarrow$ Bool returning True only for the finite value StepNatural 0.

It is important to note that calling the interpreter with StepInfinite is not the same as evaluating $[\![e]\!]_{\theta,\sigma}$ which is defined to be $\bigcup_{i\in\mathbb{N}}[\![e]\!]^i_{\theta,\sigma}$. The term semantics resulting from the equations with unrestricted recursion would not be well defined in general. In the context of the program

$loop :: \mathsf{Nat}$
$loop = loop$

the expression *loop* has a precise meaning which can be derived as $[\![loop]\!]^i_{\theta,\sigma} = \cdots = [\![loop]\!]^0_{\theta,\sigma} = \{\bot\}$ for all $i \in \mathbb{N}$, but the interpreter would fail to produce any result when called with StepInfinity. In general, due to the lambda calculus with unrestricted recursion being Turing complete, there are no means to detect whether a given expression terminates or not.

To make matters worse, for certain other programs and expressions, it depends on the selected search strategy whether the interpreter returns all possible results or terminates at all. This does not mean that StepInfinity is useless, but it should be used with care when the termination behavior of a given program is unclear.

In addition to the aforementioned properties of the environment, the Haskell representation also contains the module which provides the top-level functions, data types and a mapping of data constructors to their respective types. The latter mapping exists just for convenience and is built from the supplied module in the beginning.

A notable discrepancy between the denotational semantics and the interpreter is the actual representation of sets. All sets occurring in the denotational semantics are lower sets, i.e. for a poset $(P, \sqsubseteq)$, all subsets $A \subseteq P$ are downwards closed under $\sqsubseteq$. To reduce computational overhead, the interpreter takes no precaution that every set is indeed downwards closed. Since all CuMin functions are monotone, this does not change the outcome of the computation. Section 4.4 about pruning covers this matter in detail.

Furthermore, the interpreter internally works with multisets instead of true sets. One reason is, that the monadic operations (i.e. $\gg=$, *mplus*, ...) have no 'knowledge' about the elements inside the computation and therefore cannot filter based on any value-specific criteria (for example equality). But again, except for possible duplicates, this does not change the meaning of the semantics.

4.2 Search Strategies

This section outlines how infinitely large sets can be enumerated in Haskell, and which search strategies I employed in the interpreter.

4.2.1 Handling Infinitely Many Results

In case of finite step indices, the choice of the search strategy does not affect the results, since each set is guaranteed to be finite by construction. Hence, each value will eventually be computed. Otherwise, infinite sets may be introduced by logic variables or unlimited recursion. Depending on the structure of the computation, a depth-first search can get stuck in an infinitely descending branch, never evaluating any other branch.

As an example, consider the evaluation of

$$[\![\textbf{let } b :: \mathsf{Bool} \textbf{ free in let } n :: \mathsf{Nat} \textbf{ free in } \mathsf{Pair}_{\mathsf{Bool},\mathsf{Nat}}\ b\ n]\!]_\theta$$
$$= \bigcup_{\mathbf{b} \in \{\bot, \mathbf{False}, \mathbf{True}\}} \bigcup_{\mathbf{n} \in (\{\bot\} \cup \mathbb{N})} \downarrow(\mathbf{Pair\ b\ n})$$

without step restriction. This expression would be roughly equivalent to $[\mathsf{False}, \mathsf{True}] \gg= \lambda b.\, [0 \ldots] \gg= \lambda n.\, \mathsf{Pair}\ b\ n$ in Haskell. A depth first implementation of $\gg=$ would first commit to $\mathbf{b} = \mathbf{False}$ and then yield all countably infinitely many values of the form **Pair False n**

$$
\begin{array}{lll}
\mathit{interleave}\ \mathit{mzero} & m & \simeq m \\
\mathit{interleave}\ (\mathit{return}\ a\ `\mathit{mplus}`\ m_1) & m_2 & \simeq \mathit{return}\ a\ `\mathit{mplus}`\ \mathit{interleave}\ m_2\ m_1 \\
\mathit{mzero} \gg\!- k & & \simeq \mathit{mzero} \\
(\mathit{return}\ a\ `\mathit{mplus}`\ m) \gg\!- k & & \simeq \mathit{interleave}\ (k\ a)\ (m \gg\!- k)
\end{array}
$$

Figure 4.1: MonadLogic laws

with $\mathbf{n} \in \mathbb{N}$. Hence, pairs where the first component is **True** like **Pair True** 42 are never considered.

Therefore additional combinators for a fair choice between alternatives are needed. This problem has been addressed previously by Oleg Kiselyov et al. resulting in the LogicT monad transformer [12], available via the `logict`[2] package. In particular, they devised two additional functions $(\gg\!-)$ and *interleave*, which have the same types as $\gg=$ and *mplus*, but behave differently in that they yield results in an interleaved fashion, alternating between the left and right branch, and thus allowing to reach each result after finitely many steps. Besides providing combinators for fair search, the Logic[3] monad is also faster than the standard list monad, and can be used as a drop-in replacement.

The combinators provided by `logict` are part of the MonadLogic type class, which enables the use of these combinators for all monad transformer stacks containing LogicT with the appropriate instances. Moreover, the actual behavior of these combinators is determined by a set of laws shown in figure 4.1. The notation $a \simeq b$ denotes that a and b produce identical streams of results.

The above example can now be rewritten to $[\mathsf{False}, \mathsf{True}] \gg\!- \lambda b.\, [0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ b\ n$. Using the laws for MonadLogic, it can be shown that this expression indeed yields each particular result after a finite amount of steps.

$$
\begin{array}{l}
[\mathsf{False}, \mathsf{True}] \gg\!- \lambda b.\, [0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ b\ n \\
\simeq \mathit{mplus}\ (\mathit{return}\ \mathsf{False})\ (\mathit{mplus}\ (\mathit{return}\ \mathsf{True})\ \mathit{mzero}) \\
\quad \gg\!- \lambda b.\, [0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ b\ n \\
\simeq \mathit{interleave}\ ([0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ \mathsf{False}\ n) \\
\quad (\mathit{mplus}\ (\mathit{return}\ \mathsf{True})\ \mathit{mzero} \gg\!- \lambda b.\, [0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ b\ n) \\
\simeq \mathit{interleave}\ ([0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ \mathsf{False}\ n) \\
\quad (\mathit{interleave}\ ([0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ \mathsf{True}\ n) \\
\quad (\mathit{mzero} \gg\!- \lambda b.\, [0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ b\ n) \\
\simeq \mathit{interleave}\ ([0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ \mathsf{False}\ n) \\
\quad (\mathit{interleave}\ ([0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ \mathsf{True}\ n)\ \mathit{mzero}) \\
\simeq \mathit{interleave}\ ([0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ \mathsf{False}\ n)\ ([0 \ldots] \gg\!- \lambda n.\, \mathsf{Pair}\ \mathsf{True}\ n)
\end{array}
$$

Using the fair choice combinators, the computation returns pairs which alternately are False and True in the first component.

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

[3] Logic provides the same effects as LogicT, but is just a regular monad and no monad transformer.

4.2.2 Non-Determinism in the Interpreter

Since the MonadLogic class contains some functions that are not needed for the denotational semantics, I derived a slim version called MonadSearch (see listing 4.3). The functions *branch*, $\gg\!+$ and *peek* correspond to *interleave*, $\gg\!-$ and *msplit*. The latter function allows inspecting the results of a given monadic computation, which will become relevant for pruning.

```
class (Alternative m, MonadPlus m) => MonadSearch m where
  peek   :: m a -> m (Maybe (a, m a))
  branch :: m a -> m a -> m a
  (>>+)  :: m a -> (a -> m b) -> m b
```

Listing 4.3: Interface for non-deterministic search, FunLogic.Semantics.Search

Diffent search strategies can now be implemented by using monads with appropriate definitions for *branch* and $\gg\!+$ as inner monads for Eval (see listing 4.1). The LogicT transformer has depth-first behavior when using $\gg\!=$ and *mplus*, whereas $\gg\!-$ and *interleave* lead to a behavior similar to breadth-first (in the following, it is just called breadth-first) search. The MonadSearch instance for Logic uses the latter combinators. The same module also provides the UnFair wrapper, which takes a monad with a MonadSearch instance and replaces *branch* and $\gg\!+$ by *mplus* and $\gg\!=$ correspondingly in its own instance. Thus, UnFair Logic provides a depth-first search implementation that can be plugged into the interpreter.

A third method is iterative deepening search which enumerates the search space by repeatedly performing depth-first search while increasing the step limit. Since the step index is part of the environment, this strategy cannot be implemented solely in the non-determinism monad. Instead, it is provided as an extra combinator which applies iterative deepening on top of any inner monad implementing MonadSearch by changing the environment in each step, joining the results using *mplus*. The deepening stops when it reaches the step limit from the environment passed to the combinator. The actual implementation is language-specific, because the explicit non-determinism of SaLT is treated differently from CuMin as outlined below. Iterative deepening is the strategy which is closest to the actual definition of the denotational semantics, which for CuMin was $[\![e]\!]_{\theta,\sigma} = \bigcup_{i\in\mathbb{N}} [\![e]\!]^i_{\theta,\sigma}$ and is similar for SaLT. Therefore iterative deepening can only be applied to set typed expressions in SaLT. When there is no step restriction, the search never stops. The deepening combinator has no means to detect whether a computation failed because it hit the recursion limit or because the result space was exhaustively searched.

In the end, the results need to be extracted from a LogicT computation. The `logict` package provides three *observe*$*$ functions to convert a such computation to a list. To minimize the syntactic overhead of replacing the concrete search strategy, I wrapped these functions in a type class named Observable.

4.3 Logic Variables

The semantic interpretation of logic variables (introduced by **let** $x :: \tau$ **free in** ... (CuMin) and $\mathbf{unknown}_\tau$ (SaLT) respectively) relies on a set comprised of all possible inhabitants

of the given type. Therefore, the interpreter solves logic variables by trying every possible combination of values. This is by no means efficient, but it exactly models the denotational semantics, whereas the operational semantics works with a more target-oriented approach.

4.3.1 Difficulties in Enumerating Type Inhabitants

As discussed in the previous section, depth-first search is mostly unable to produce all possible values of a data type with an infinite number of inhabitant (except for simple data types like **data** N = Z | S N). This problem has been tackled by the introduction of generator functions in section 3.4. When evaluating with a finite step index, this guarantees that all generated sets are finite.

Another pitfall are *left-recursive* data types. An algebraic data type is left-recursive if the first constructor (transitively) contains an argument of the same type. The following LRec type is an example for a left-recursive ADT, whereas the RRec type is not left-recursive.

data LRec = LR LRec | LNil
data RRec = RNil | RR RRec

When enumerating the values of an algebraic data type, the constructors have to be traversed in a certain order, in my implementation this order is from left to right. But to produce the LR value, it must first produce a value of type LRec. But when the first branch is always taken, the recursion gets stuck. It is important to note, that using *interleave* does not help in this situation as the recursion takes place in the left branch, and since *interleave* evaluates the left argument first, it fails to produce any value. Furthermore, for each fixed order of traversing the constructors, there is a data type which would provoke a diverging computation.

One possible solution would be shifting the responsibility upon the programmer, who then has to take care not to use any left-recursive data types. Alternatively, one could return an extra bottom value on each level, and therefore ensuring termination. This comes at the cost that now the sets constructed for logic variables are significantly larger than before, and one no longer utilizes the fact that it suffices to perform calculations with maximal elements. The middle course would be a heuristic approach which detects whether the data type is left-recursive (i.e. the first constructor is recursive) and only injects a bottom value in this particular case. This is what I eventually implemented after the first approach turned out to be too inefficient.

4.3.2 Implementation

The following functions responsible for producing these sets are implemented in the module FunLogic.Semantics.Denotational, in the `denotational-funlogic` package. They are all prefixed with *anything* because that was the name in the original paper, before the **let** $x :: \tau$ **free in** e and $\mathbf{unknown}_\tau$ primitives have been introduced.

anything :: (Value *val*, MonadSearch *n*) ⇒ FL.Type → Eval *bnd val n* (*val n*)
: This is the public interface of the set generation. The type signature is general enough to allow this function to be used by both CuMin and SaLT. The actual work is done by the next function.

anything′ :: (. . .) ⇒ HashSet TyConName → Type → Eval *bnd val n* (*val n*)

The function analyzes the given type and returns a non-deterministic computation using the monad n returning a stream of inhabitants of the type depending on the environment. For non Data types, such as functions or sets, the function fails with a runtime error. Natural numbers are delegated to the *anyNatural* function described below. When a type variable is encountered, the function calls itself with the closed type taken from the type environment.

The handling of algebraic data types is a bit more complex due to the problem regarding left-recursion discussed above. The HashSet TyConName parameter mentioned in the type signature keeps track of the data types visited by left-recursion. If *anything′* is asked to return the values of a type already contained in this set, it first yields a bottom value.

Then, the values of each constructor are produced using *anyConstructor*. For the first constructor, the left-recursion set is extended by the current data type.

anyNatural :: (. . .) ⇒ Eval *bnd val n* Integer

For a finite step index i, this function simply returns a finite stream of all natural numbers less than 2^i, beginning with 0. When the step index is infinite, an infinite stream of all natural numbers in their natural order is returned.

anyConstructor :: (. . .) ⇒ HashSet TyConName → [Type] → Map TVName Type → ConDecl → Eval *bnd val n* (*val n*)

The values of each constructor are generated as a conjunction of the arguments using the MonadSearch combinators. This ensures that in the presence of infinite sets, fair search strategies are able to explore the whole value space. Additionally, the function receives a list of types from the polymorphic instantiation, which are stored together with the actual values and a mapping from type variables to their instantiations, since this information is needed for generating the argument values.

For the definitions of LRec and RRec from the example discussed earlier, *anything* would return **LR** ⊥, **LR** (**LR** ⊥), **LR LNil** and **LNil** respectively **RNil**, **RR RNil** and **RR** (**RR** ⊥) for a step index of 2. The **LR** ⊥ value originates from the aforementioned heuristics for handling left-recursive data types, whereas **RR** (**RR** ⊥) and **LR** (**LR** ⊥) result from the recursion limit imposed by the step index.

The full lower subsets of the type semantics of LRec and RRec produced by the generator functions at step index 2 are given below.

$$\mathrm{gen}_{\mathsf{LRec}}(\emptyset, 2) = \{\bot, \mathsf{LNil}, \mathsf{LR}\ \bot, \mathsf{LR}\ \mathsf{LNil}, \mathsf{LR}\ (\mathsf{LR}\ \bot)\}$$
$$\mathrm{gen}_{\mathsf{RRec}}(\emptyset, 2) = \{\bot, \mathsf{RNil}, \mathsf{RR}\ \bot, \mathsf{RR}\ \mathsf{RNil}, \mathsf{RR}\ (\mathsf{RR}\ \bot)\}$$

The larger the step index, the more is gained by only considering maximal elements.

4.4 Pruning Unnecessary Results

Even though the interpreter does not enforce lower sets, they might as well emerge as part of the evaluation process. Consider $choose_{\mathsf{Nat}}\ 1\ \mathbf{failed}_{\mathsf{Nat}}$, which would evaluate to $\{\bot, 1\}$.

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

Listing 4.4: Implementation of the *choose* function from the CuMin prelude

This combined with the multiset behavior can quickly lead to a combinatorial explosion where a lot more calculations than necessary are performed. This is demonstrated by the following example.

4.4.1 Necessity of Pruning

Listing 4.4 shows again the implementation of the *choose* function. When evaluated, the logic variable c takes the two values False and True. Because of call-time choice, the argument variables have a single value assigned when the function body is evaluated, which means it will be evaluated for each pair of input values, each time producing two values. Thus, if $xs :: \tau$ is some expression that evaluates to a multiset with n elements and $ys :: \tau$ evaluates to a multiset with m elements, $\mathit{choose}_\tau\ xs\ ys$ evaluates to a set containing $2nm$ entries. A cascade of n nested calls to *choose* then produces a set of size $\mathcal{O}(2^n)$ with many duplicates. This yields to severe slow-downs on expressions like $\mathit{foldr}_{\mathsf{Nat},\mathsf{Nat}}\ \mathit{choose}_{\mathsf{Nat}}\ \mathbf{failed}_{\mathsf{Nat}}\ [1,\ldots,100]_{\mathsf{Nat}}$. One would expect the interpreter to output 100 results, but the multiset produced by the interpreter actually contains 2^{100} elements. Special care must therefore be taken to avoid such an exponential inflation by removing duplicates and non-maximal elements at some points during the evaluation.

4.4.2 Strategies for Pruning

I have implemented two pruning strategies. Both of them make use of the aforementioned *peek* function (listing 4.3) to inspect the incoming stream of elements. It either returns Nothing if the stream is exhausted or Just (x, xs) to return one result x and a computation xs producing the other answers.

The first approach filters out non-maximal elements by maintaining a list of maximal elements already seen. Any element less than some item in the list is discarded. Otherwise, the element is added to the list and all smaller elements are removed from the list. It relies on the PartialOrd type class which provides two comparisons *leq* and *lt*. Implementations are required to define one, the other is then derived by using the Eq type class.

Only past elements are taken into account, since a larger (w.r.t. the partial order) element can always appear at some later point. But then one has to be careful about how many elements should be cached for pruning. On the one hand, the more elements are cached, the more duplicates are possibly filtered out. But on the other hand, even with few elements, this quickly leads to a high memory consumption and quadratic slow-down when there are many distinct elements.

Especially in combination with breadth-first search which keeps many alternative branches in memory and each of these alternative computations in turn keeps a reference to its respective pruning cache, this strategy consumes a lot of memory (depending on the

branching-factor). An ad-hoc solution to limit the extent to which memory is consumed is to add an upper limit to size of the cache, removing older elements as newer come in. Moreover, it is difficult to choose where the *prune* calls should be inserted during the evaluation, and if it is actually needed at all, since this generally depends on the actual expression being evaluated.

The second approach uses the artificial[4] Ord instance of Value to remove duplicates. While it is not possible to detect non-maximal values using the Ord instance, duplicates can be found more efficiently using the Set[5] type which exhibits logarithmic lookup time compared to the linear scan of the first approach.

The *pruningImpl* field of the environment(see listing 4.1) allows different strategies to be plugged in for the evaluation. The two strategies discussed earlier are implemented in FunLogic.Semantics.Pruning in terms of the functions *pruneNonMaximal* and *pruneDuplicates*.

4.5 Evaluating Expressions

Using the mechanisms explained in the previous sections, I can now present how actual expressions can be evaluated using the implementation of the denotational term semantics.

4.5.1 Representation of Values

The implementation of the CuMin term semantics is now straightforward. Listing 4.5 shows the data type modeling the semantic domain of CuMin. The VNat constructor just wraps Haskell's unbounded integer data type. Bottom values are annotated with a message describing where the value originated. This helped for debugging the interpreter, but this message is omitted by default when printing values to the screen.

Constructors consist of a name and a list of argument values. Additionally, they carry the type instantiation used for the particular value. The latter is unnecessary for the actual computation, but it might be interesting for users, therefore it is configurable whether to include the type information in the visual representation or not.

Functions are modeled as a function mapping a value to a set of values, where the concrete set type is determined by the type parameter of Value. This means that functions are opaque and cannot be inspected by other means than feeding them different inputs. Moreover, the environment used for evaluating the function is threaded through the computation when the function expression is evaluated. There is no possibility to change, for example, the step index after the value has been constructed.

The additional Unique parameter serves to distinguish functions, apart from point-wise comparison. While two function values with different unique IDs may still be equivalent in this sense, it is known for sure that two functions are exactly the same when their two unique fields are identical.

The generation of the unique values is rather ad-hoc and uses the functions from Data.Unique in combination with *unsafePerformIO*. This means, that the interpreter is no longer pure since multiple calls with the same arguments can return different values.

[4] *artificial* in the sense that there is no *natural* total order on values

[5] contained in `Data.Set` from the `containers` package

```
data Value n
  = VCon DataConName [Value n] [Type]
  | VNat Integer
  | VFun (Value n -> n (Value n)) !Unique
  | VBot String
```

Listing 4.5: Representation of the semantic domain of CuMin

However, the unique field is an implementation detail which is used internally to speed up pruning, and should not be considered as a part of the actual output.

The PartialOrd instance for values is, apart from the special handling of function values, exactly the one given by definition 3.1.

4.5.2 Implementing Evaluation

The actual *eval* function is just a translation of the denotational term semantics to Haskell. Indexed unions are represented by calls to the $\gg\!+$ MonadSearch combinator, binding variables in the environment or decrementing the step index is realized by using the $local :: \mathsf{MonadReader}\ r\ m \Rightarrow (r \to r) \to m\ a \to m\ a$ function of the reader monad interface, which transforms the environment using the function from the first argument during the evaluation of the second argument. The expressions to be evaluated and the module passed as environment are assumed to be well-typed. The *eval* function is partial and is only guaranteed to produce results on well-typed programs. Otherwise, it may raise runtime errors.

The most complicated part of *eval* is the handling of functions. It builds a nested chain of VFun closures, one for each argument. The innermost part then evaluates the function body with a fixed environment, taking the decremented step index from the call site.

The excerpt from the interpreter presented in listing 4.6 is responsible for the evaluation of function applications. The outermost *prune* call uses the pruning function from the environment (listing 4.1) and applies it to its argument, in this case the result from the function call. The callee and the argument expression are both evaluated and their results are bound using the MonadSearch combinator $\gg\!+$ to individual values. The *primApp* function handles the actual function application. When something is applied to bottom, the result is bottom, regardless of the argument. In case the callee is a function value, the closure is bound to f, which has the type Value $n \to n$ (Value n) (see listing 4.5). Applying f to the other argument returns a value of type n (Value n) where actually Eval n (Value n) is needed. The latter is a synonym for ReaderT (EvalEnv n) n (Value n). This is where the function $lift :: m\ a \to \mathsf{ReaderT}\ e\ m\ a$ is needed[6], wrapping a computation in the inner monad with the outer monad, thereby *lifting* it through the stack of monad transformers.

4.5.3 Differences for SaLT

The implementation of the semantics of SaLT only differs in a few points. Firstly, the Value type must be able to represent sets, and functions now have no inherent non-determinism.

[6]Actually, *lift* has the more general type $(\mathsf{Monad}\ m) \Rightarrow m\ a \to t\ m\ a$ and is provided by the MonadTrans t type class. Hence, it can be overloaded for any monad transformer.

```
...
eval (CuMin.EApp funE argE) = prune (eval funE
  >>+ \fun -> eval argE
  >>+ \arg -> primApp fun arg)
...

primApp :: MonadSearch n
        => Value n -> Value n -> Eval n (Value n)
primApp (VFun f _) a = lift (f a)
primApp (VBot n) _ = return (VBot n)
primApp _ _ = error "application of non-function type"
```

Listing 4.6: Evaluation of function calls

```
...
  | VFun (Value n -> Value n) !Unique
  | VSet (n (Value n)) !Unique
```

Listing 4.7: Representation of the semantic domain of SaLT compared to CuMin (listing 4.5)

Listing 4.7 shows the differences of the value type compared to CuMin. The Unique argument of VSet serves the same purpose as the one for function values. In general, it is impossible to compare two possibly infinite sets.

Unlike CuMin, non-determinism is explicit in SaLT. Therefore the evaluation monad is no longer a reader transformer over a non-deterministic inner monad, but just a plain Reader[7]. Since the *anything* function from the core library produces a non-deterministic computation, $mapReaderT :: (m\ a \to n\ b) \to \mathsf{ReaderT}\ r\ m\ a \to \mathsf{ReaderT}\ r\ n\ b$ is used to "capture" the non-deterministic inner monad and wrap it in a VSet constructor inside the identity monad. The only location needing fair choice is now the primitive bind operation of SaLT, which can almost directly be translated to a call to $\gg\!+$.

[7] which is actually a reader transformer over the identity monad

Chapter 5

Evaluation

In this chapter I am going to describe how I evaluated and tested my work during development, and how it compares to other approaches.

5.1 REPL

To conveniently work with the denotational semantics, I developed a REPL similar to GHCi for each language, named *cumint* (CuMin Interactive) and *isalt* (Interactive SaLT). The installation instructions can be found in appendix A.

They support loading multiple modules at once, although the functions and data types have to be disjoint. The step index, the search strategy and the pruning function used for the evaluation can be configured interactively. After each evaluation the CPU time consumed in the process is printed to the screen. This helped especially during development to quickly find out how certain changes to the interpreters affected the performance.

Furthermore, when starting the SaLT REPL, one can choose between the native SaLT prelude and the CuMin prelude translated to SaLT. The latter contains a lot of unnecessarily set-typed expressions which consist solely of singleton sets, and is therefore more cumbersome to use when writing directly in SaLT. Nevertheless it is needed for loading CuMin modules that have been automatically translated to SaLT using Fabian Zaiser's `cumin2salt`[1] library which realizes the translation rules from [14] for the abstract syntax trees presented in section 2.1.

The core functionality implemented in `denotational-funlogic` is shared across both REPLs which was made possible by the extensive code reuse between CuMin and SaLT. The specific implementations only define functions for loading modules, type-checking and evaluating expressions. The input is handled by *Haskeline*[2] providing a command history and auto-completion for file paths. While the auto-completion could have been extended to also support identifiers in expressions like GHCi, this would have been to time-consuming for the scope of this thesis. The complete functionality is documented in appendix B.

[1] `https://github.com/fanzier/cumin2salt`

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

5.2 Effectiveness of Pruning

The exponential inflation of the multisets resulting from nested *choose* calls can now be demonstrated using the REPL. With pruning disabled and breadth-first search, the CuMin interpreter yields the following results for a right fold of *choose*.

$$\begin{aligned}&\mathit{foldr}_{\mathsf{Nat},\mathsf{Nat}}\ \mathit{choose}_{\mathsf{Nat}}\ \mathbf{failed}_{\mathsf{Nat}}\ [1,2,3]_{\mathsf{Nat}}\\ &\quad\rightsquigarrow \{1,1,2,1,2,1,3,\bot\}\\ &\mathit{foldr}_{\mathsf{Nat},\mathsf{Nat}}\ \mathit{choose}_{\mathsf{Nat}}\ \mathbf{failed}_{\mathsf{Nat}}\ [1,2,3,4]_{\mathsf{Nat}}\\ &\quad\rightsquigarrow \{1,1,2,1,2,1,3,1,2,1,3,1,2,1,4,\bot\}\end{aligned}$$

This empirically supports the motivation for pruning, as there are 2^n results when folding over a list of length n. By pruning duplicates, these sets shrink to the following.

$$\begin{aligned}&\mathit{foldr}_{\mathsf{Nat},\mathsf{Nat}}\ \mathit{choose}_{\mathsf{Nat}}\ \mathbf{failed}_{\mathsf{Nat}}\ [1,2,3]_{\mathsf{Nat}}\\ &\quad\rightsquigarrow \{1,2,3,\bot\}\\ &\mathit{foldr}_{\mathsf{Nat},\mathsf{Nat}}\ \mathit{choose}_{\mathsf{Nat}}\ \mathbf{failed}_{\mathsf{Nat}}\ [1,2,3,4]_{\mathsf{Nat}}\\ &\quad\rightsquigarrow \{1,2,3,4,\bot\}\end{aligned}$$

By using the pruning method for non-maximal values, even the trailing bottom values can be purged from the result sets at the cost of an approximately quadratic time complexity in the number of maximal elements. The performance characteristics of pruning in combination with different search strategies are evaluated in the next section.

5.3 Performance of Search Strategies and Pruning

In this section, the three search strategies and two pruning methods are assessed for performance on three selected CuMin samples. While the benchmarks are unlikely to yield universally valid results, they nevertheless provide some intuition about when a particular search or pruning strategy is applicable.

5.3.1 Benchmark Architecture

For the following benchmark, it is important to define when a computation counts as completed. When no step limit is imposed, the interpreter might be unable to detect whether there are more results after the first one has been printed. Especially with iterative deepening search, it is generally undecidable whether there are more results available when increasing the depth or not. I therefore decided to stop the evaluation when the first fully defined value has been returned. This ensures that preceding partial results are not counted. The fact that values need to be completely traversed when searching for bottoms guarantees that laziness does not affect the benchmarking results.

The CuMin program used for benchmarking has been translated to SaLT using cumin2salt with optimizations turned on. These optimizations roughly halved the processing time for SaLT in most examples, but are not part of this benchmark.

The benchmarking program is implemented in the `denotational-tests` package which contains a CuMin module (`files/TestEnv.cumin`) containing the expressions to

be evaluated amongst some utility functions and other tests described later on. All top-level bindings starting with "bench" are considered to be benchmarks. They must be monomorphic and not have any arguments.

Each such expression is then evaluated using all combinations of the three different strategies, depth-first search using LogicT with unfair choice, breadth-first search using LogicT and iterative deepening depth-first search with each pruning configuration, being pruning non-maximal elements, pruning duplicate elements and no pruning at all. Since each expression is also translated to SaLT, this amounts to 18 different combinations for each benchmarking expression.

5.3.2 Evaluation of Benchmarking Results

Figure 5.1 shows the (wall clock) execution times of the three benchmarks described below with a logarithmic scale due to the large differences between some values. The times have been measured using the *criterion*[3] library, which has been especially designed for microbenchmarking.

`benchSort` sorts a list by generating all possible permutations, filtering for a sorted one, and therefore heavily relies on non-determinism. Without pruning, it suffers from a similar combinatoric explosion as nested cascades of *choose*.

`benchSubPeanoFree` performs subtraction of two Peano numbers using a logic variable. To calculate $a - b$, it searches for a number d such that $b + d = a$.

`benchSubPeanoDet` performs the same subtraction deterministically by structural recursion, completely avoiding non-determinism except for signaling failure.

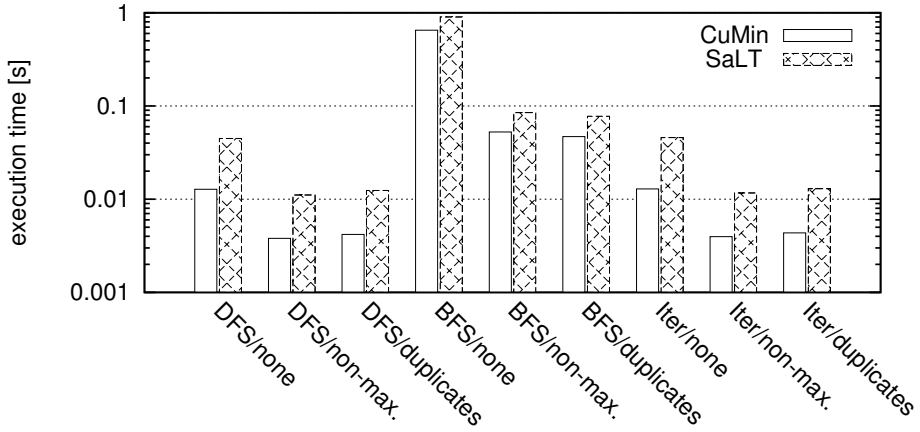
In all cases, the evaluation of CuMin terms has been faster than the evaluation of the translated SaLT code, which is likely to be caused by the wrapping and unwrapping of set constructors needed for explicit non-determinism.

Unsurprisingly, pruning deteriorated performance in the deterministic benchmark (figure 5.1c), as only singleton sets occur during evaluation. Likewise, pruning slowed down the evaluation of the second benchmark in figure 5.1b, but for a different reason. The subtraction using a logic variable works by trying out all possible differences of the two values, which are all distinct maximal elements. Hence, pruning only removes the duplicate bottom values resulting from a non-matching difference in that case. However, these are filtered out anyway, as the benchmarking program searches for the first fully defined value. As expected, introducing pruning significantly improved the performance in the sorting benchmark (figure 5.1a). None of the two different pruning strategies can be said to perform significantly better than the other, as this depends on the concrete problem at hand and the choice of search strategy.

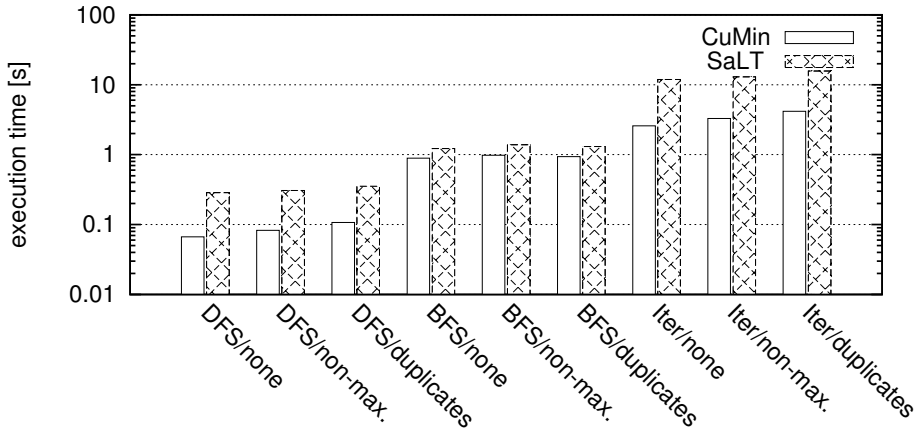
Depth-first search proved to be the fastest implementation for all three benchmarks, albeit, as discussed previously, it is not applicable to all problems. It seems that breadth-first search does not perform well for high branching factors as shown by figure 5.1a whereas iterative deepening search is roughly on par with depth-first search for this problem. On the contrary, iterative deepening performed worst for the other two problems (figures 5.1b

[3] `http://www.serpentine.com/criterion/`

(a) `benchSort`: Sorting by permuting



(b) `benchSubPeanoFree`: Inverse of addition by using a logic variable



(c) `benchSubPeanoDet`: Deterministic implementation of Peano subtraction

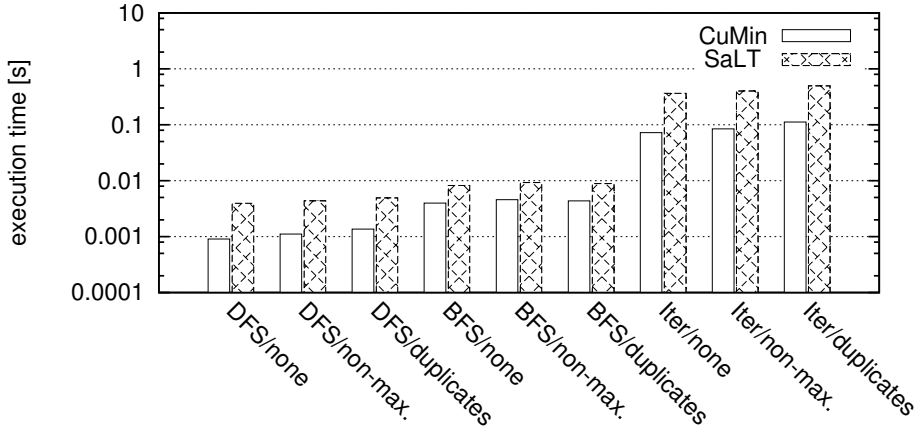


Figure 5.1: Mean (wall clock) execution time of the benchmark programs

and 5.1c), most likely because their branching factor is smaller and therefore less execution paths can be cut off using iterative deepening.

5.4 Testing

During initial development, I examined the correctness of new functionality interactively using the REPL. As the interpreters grew more complex, I also implemented test suites based on *Hspec*[4] for the interpreters. For the most part, these tests consist of known combinations of expressions and the corresponding values they should evaluate to, given by manually applying the denotational semantics.

5.4.1 Language-Specific Tests

For each language, there are three groups of tests. The first kind of tests checks each syntax feature separately to make sure its implementation matches the equations from figures 3.1 and 3.2 at least for certain examples. A second group of tests is dedicated to ensure that the prelude functions work as expected. The last tests verify that the PartialOrd instance of the value types indeed is a partial order, i.e. that it is reflexive, anti-symmetric and transitive. The latter are implemented by checking these three properties for small instances of lists of naturals using *SmallCheck*[5] [15].

SmallCheck provides combinators for generating values up to a given depth and using those to check certain properties, which are simply functions returning a boolean value. This approach is similar to the popular *QuickCheck* [7] library with the only major difference being that QuickCheck generates values using a random distribution whereas SmallCheck is deterministic.

I eschewed *QuickCheck* as randomized testing proved to be too difficult for this scenario, mainly because generating meaningful and correctly typed abstract syntax trees is a problem beyond the goals of my thesis.

5.4.2 Compatibility of the Interpreters

Theoretically, the interpreters for CuMin and SaLT should produce identical results (apart from the order of results) when evaluating CuMin expressions and their respective SaLT translations. Especially for syntactic constructs which are only present in one of the languages, like **let** $x :: \tau$ **free in** e, it is non-trivial to see whether both interpreters are consistent with respect to the translation of CuMin to SaLT.

The test framework to verify this in practice is also implemented in the `denotational-tests` package, using the same CuMin module as the benchmark program. The tests for verifying the compatibility are the top-level expression prefixed by "eqTest". The whole module will then be type-checked and converted to a SaLT module using the previously mentioned *cumin2salt* library. To preempt bugs caused by an erroneous translation, the resulting SaLT program is passed to the type checker again. Finally, all top-level expressions identified as test cases are evaluated in their respective interpreter using both

[4] `https://hackage.haskell.org/package/hspec`
[5] `https://hackage.haskell.org/package/smallcheck-1.1.1`

depth-first and breadth-first search. The initial step index is set to five to limit the test duration. The results produced using the same search strategy are required to consist of the same elements in the same order. For different search strategies, the streams of answers are at least required to yield the same values regardless of the actual order. The following list describes the test I implemented.

`eqTestCaseFailureCase1..3` ensure that catch-all patterns behave identically (i.e. they are only evaluated if all other patterns do not match),

`eqTestLetFree*` cover the translation of logic variables,

`eqTestAddition` checks the combination of a primitive operation with non-determinism,

`eqTestSubtraction` is a more complex example involving logic variables to subtract values,

`eqTestLetBinding1` checks the translation of normal let-bindings (i.e. **let** $x = e_1$ **in** e_2 is equivalent to $(\lambda x :: \tau.\ e_2)\ e_1$ where $e_1 :: \tau$),

`eqTestLetBindingShadowing` ensures that name shadowing behaves identical for let-bindings and lambda abstractions,

`eqTestAncestors*` are built of the ancestors example (listing 1.2) from the introduction,

`eqTestRecursion` checks the behavior of a recursive function.

The biggest drawback of this approach is that it is not immediately clear if there is a bug in the translation library or in the interpreters when there are discrepancies between the results. But these tests are nevertheless useful to prevent introducing bugs by refactoring or rewriting parts of the code. Albeit, I did not find any bugs with them so far that could not be found by testing separately.

Chapter 6

Conclusion

As I have shown in the previous chapters, it is generally possible to implement the denotational semantics of non-deterministic functional-logic languages.

Since the denotational semantics of CuMin and SaLT reduce free variables to non-determinism by enumerating all possible values, the latter was the greatest challenge during the implementation.

Firstly, the representation of sets is non-trivial, especially when they might be infinite. This was alleviated by extending the notion of *step indices* to generator functions, forcing all sets to be finite as long as the step index is finite.

In contrast to the denotational semantics, where the lower sets provide "nice" mathematical properties, the large number of elements is undesired for the interpreters. The monotonicity of CuMin and SaLT functions ensures, that a more defined input leads to a more or equally defined output. Hence, technically, it is enough to work with the maximal elements of these sets, reducing computational overhead.

Sometimes, a computation yields several $\sqsubseteq$-comparable elements or duplicates, where it is advantageous to keep only the most defined ones. I described two different pruning strategies to handle these cases. The first one uses the partial order on values to remove all non-maximal elements from a stream of answers by keeping track of the maximal elements encountered so far. Since the partial order does not provide enough structure for an efficient lookup, this requires a linear scan of these elements. The other strategy only removes duplicate values using an artificial total order on values, allowing for logarithmic lookup time.

Moreover, the choice of the search strategy greatly influences the performance and completeness of the interpreters. While the actual interpreter is not tied to a particular non-determinism monad by requiring a specific interface using a type class, I only provided three strategies for the REPLs.

Depth-first search is the fastest, but only works for expressions with finite results or when the step index is finite. For infinite search spaces, it might commit to an infinite branch and therefore never reach some solutions.

This necessitated another search strategy which was able to cope with infinite streams of results. One such strategy is the *LogicT* monad invented by Kiselyov et al. It provides additional combinators to interleave two streams of results, ensuring that each answer is returned after a finite number of steps, regardless of how many results there are in total.

LogicT performs some kind of breadth-first search on the tree implied by the structure of the computation.

The last search strategy I implemented was iterative deepening search. It uses the same monad as depth-first search, but limits the search depth to an increasing number in each pass. Consequently, each individual depth-first search has to traverse only a finite search space, but the increasing step index guarantees that each possible solution is found in finite time, even though enumerating all results might not terminate.

The benchmarks have shown that, while generally depth-first search is fastest when it is applicable, there is no optimal search strategy applicable to all kinds of problems. Similarly, the benefit of pruning depends both on the evaluated term and the chosen search strategy. As before, their is no optimal pruning strategy that fits all computations.

Currently, the use of my interpreters is limited to expressions with small branching factors because of the way free variables are handled. While the current method has the advantage of directly relating the implementation to the defining equations of the denotational semantics, evaluating expressions containing free variables by instantiating them with all possible value combinations is too slow for larger programs. It seems that there are no obvious solutions without breaking the tight coupling of the interpreters with the denotational semantics.

Nevertheless, the interpreter is useful for verifying program transformations of small programs and for verifying the results produced by an interpreter based on the operational semantics. Furthermore, it is possible to check the equivalence of a CuMin and a SaLT program using Fabian Zaiser's translation library.

Appendix A

Installation Instructions

The implementation has been developed using the Glasgow Haskell Compiler (GHC) version 7.8.4, although it might also work with other 7.8 versions. Except for the base libraries, all dependencies have been installed from Hackage[1] using *cabal*[2] version 1.20. The Haskell Platform, a prefab collection of several commonly used packages, has not been used and it is unlikely that the packages created for this thesis can be compiled using the Platform because of dependency version mismatches.

I developed the libraries using Linux, and do not know if there are any unforeseen compatibility issues with the interpreters when compiled on Windows, due to the lack of a Windows development environment. Furthermore, following step-by-step instructions are tailored to a Linux environment, but should be easy to adapt for Windows if necessary.

1. All required packages are contained in the folder `bachelor-thesis` on the attached CD. The folder has to be copied to a writable location.

2. Open a shell, changing to the target directory.

3. Under ideal conditions, it suffices to execute the contained `init-sandbox.sh` script and wait for all packages to be installed. As the interpreters draw in a large number of transitive dependencies, this process can take while.

   Essentially, the script performs the following actions, which can alternatively run by hand.

   Create a cabal sandbox for the dependencies with `cabal sandbox init`. The sandbox is created in a subdirectory named `.cabal-sandbox`. From now on, the sandbox directory is referred to as `$SANDBOXDIR`.

   For each of these packages (the order matters)

```
1 dependencies/ba-funlogic-common/funlogic-core
2 dependencies/ba-funlogic-common/language-cumin
3 dependencies/ba-funlogic-common/language-salt
```

[1] The Haskell package archive, `http://hackage.haskell.org/`

[2] A tool for downloading and installing Haskell packages.

```
4 dependencies/cumin2salt
5 code/denotational-funlogic
6 code/denotational-cumin
7 code/denotational-salt
8 code/denotational-tests
```

execute the following commands (the current package directory is referred to as `$PKGDIR`)

```
1 $ cd $PKGDIR
2 $ cabal sandbox init --sandbox="$SANDBOXDIR"
3 $ cabal install -j --only-dependencies ↩
    ↪ --constraint="indentation␣+trifecta␣-parsec" ↩
    ↪ --constraint "blaze-markup==0.6.2.0"
4 $ cabal configure
5 $ cabal install
6 $ cabal sandbox add-source "$PKGDIR"
```

The second line references the main sandbox in the package directory, as all packages have to be installed in the same sandbox. Then all the dependencies of the current package are installed. The first *constraint* argument configures the `indentation` package to use `trifecta` instead of `parsec`, the second one is necessary because newer versions of `blaze-markup` break the `lens` package, another dependency. Finally, the commands in lines four and five ensure that all dependencies are installed correctly and eventually install the package into the sandbox. The last command is optional. It registers the package as a source dependency, meaning that dependent packages automatically force a rebuilt when the dependency changed.

After these steps, the `$SANDBOXDIR/bin` directory should contain all executables built from the source code, most notably the `cumint` and `isalt` REPLs as well as `benchmark-runner` and `test-runner` to execute the benchmarks and compatibility tests presented in chapter 5.

Appendix B

REPL Documentation

The REPL executables are named *cumint* and *isalt* for CuMin and SaLT respectively. Both of them accept file names of modules that should be loaded as command line arguments. Additionally, SaLT has the `-prelude <salt/cumin/none>` command line option to choose the prelude. The `cumin` prelude is the CuMin prelude translated to SaLT, and contains a lot of unnecessary non-determinism, but is required to actually load and run translated CuMin programs. For programs written directly in SaLT, `salt` (the default value) provides a prelude where most non-determinism has been eradicated. To start without any prelude, `none` should be used. The different preludes are explained in greater detail in section 2.5.

B.1 Interactive Commands

:load <file> loads an additional module in the REPL session. It may not contain identifiers that are already present in the current session.

:reload tries to reload all successfully loaded modules from disk.

:set <prop> = <value> sets an interpreter property.

:get [prop] prints the value of a property. Without any argument, the command lists all properties and their associated values.

:list displays a list of all functions and data types available in the current session.

:def <name> shows the definition of a function or data type.

:type <expr> infers the type of the given expression.

:quit stops the REPL.

:help prints a help screen explaining the available commands and properties.

<expr> Entering an expression not prefixed by any command evaluates it using the current settings and prints the results to the screen.

Note that command and property names as well as values can be accessed by prefixes as long as they are unique.

B.2 Adjustable Properties

Name	Values	Description
strategy	BFS/DFS/IterDFS	Allows the selection of the search strategy as described in section 4.2.
depth	$n \in \mathbb{N}$ or *inf*	Controls the initial step index used for the evaluation. Using *inf* removes all restrictions imposed on recusion, hence it should be used with care in combination with *DFS*.
numresults	$n \in \mathbb{N}$	Limits the number of results that are printed to screen at once when evaluating large result sets. After each block of n results, the REPL asks for confirmation.
showtypes	yes/no	When enabled, the type instanciations of constructors are also printed to screen and bottom values are annotated with a string indicating where it originates.
pruning	nonmaximal/duplicates/none	Configures the pruning function used during evaluation. The values correspond to the approaches discussed in section 4.4.

Bibliography

[1] Michael D Adams and Ömer S Ağacan. "Indentation-sensitive parsing for Parsec". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. ACM. 2014, pp. 121–132.

[2] E. Albert et al. "Operational Semantics for Declarative Multi-Paradigm Languages". In: *Journal of Symbolic Computation* 40.1 (2005), pp. 795–829.

[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] Sergio Antoy and Michael Hanus. "Overlapping Rules and Logic Variables in Functional Logic Programs". In: *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Sandro Etalle and Miroslaw Truszczynski. Vol. 4079. Lecture Notes in Computer Science. Springer, 2006, pp. 87–101. ISBN: 3-540-36635-0. DOI: 10.1007/11799573_9. URL: http://dx.doi.org/10.1007/11799573_9.

[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*. Ed. by Herbert Kuchen. Vol. 6816. Lecture Notes in Computer Science. Springer, 2011, pp. 1–18. ISBN: 978-3-642-22530-7. DOI: 10.1007/978-3-642-22531-4_1. URL: http://dx.doi.org/10.1007/978-3-642-22531-4_1.

[6] Jan Christiansen, Daniel Seidel, and Janis Voigtländer. "An Adequate, Denotational, Functional-Style Semantics for Typed FlatCurry". In: *Functional and Constraint Logic Programming - 19th International Workshop, WFLP 2010, Madrid, Spain, January 17, 2010. Revised Selected Papers*. Ed. by Julio Mariño. Vol. 6559. Lecture Notes in Computer Science. Springer, 2010, pp. 119–136. ISBN: 978-3-642-20774-7. DOI: 10.1007/978-3-642-20775-4_7. URL: http://dx.doi.org/10.1007/978-3-642-20775-4_7.

[7] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. Ed. by Martin Odersky and Philip Wadler. ACM, 2000, pp. 268–279.

ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: http://doi.acm.org/10.1145/351240.351266.

[8] Louis-Julien Guillemette and Stefan Monnier. "Type-Safe Code Transformations in Haskell". In: *Electr. Notes Theor. Comput. Sci.* 174.7 (2007), pp. 23–39. DOI: 10.1016/j.entcs.2006.10.036. URL: http://dx.doi.org/10.1016/j.entcs.2006.10.036.

[9] Michael Hanus. "Functional Logic Programming: From Theory to Curry". In: *Programming Logics - Essays in Memory of Harald Ganzinger*. Ed. by Andrei Voronkov and Christoph Weidenbach. Vol. 7797. Lecture Notes in Computer Science. Springer, 2013, pp. 123–168. ISBN: 978-3-642-37650-4. DOI: 10.1007/978-3-642-37651-1_6. URL: http://dx.doi.org/10.1007/978-3-642-37651-1_6.

[10] Graham Hutton and Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham, 1996.

[11] Patricia Johann and Neil Ghani. "A principled approach to programming with nested types in Haskell". In: *Higher-Order and Symbolic Computation* 22.2 (2009), pp. 155–189. DOI: 10.1007/s10990-009-9047-7. URL: http://dx.doi.org/10.1007/s10990-009-9047-7.

[12] Oleg Kiselyov et al. "Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl)". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: ACM, 2005, pp. 192–203. ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086390. URL: http://doi.acm.org/10.1145/1086365.1086390.

[13] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 333–343. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199528. URL: http://doi.acm.org/10.1145/199448.199528.

[14] Stefan Mehner et al. "Parametricity and Proving Free Theorems for Functional-Logic Languages". In: *16th International Symposium on Principles and Practice of Declarative Programming* (2014).

[15] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 37–48. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411292. URL: http://doi.acm.org/10.1145/1411286.1411292.

[16] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Dubuque, IA, USA: William C. Brown Publishers, 1986. ISBN: 0-697-06849-2.

[17] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *SIGPLAN Not.* 37.12 (Dec. 2002), pp. 60–75. ISSN: 0362-1340.

[18] Philip Wadler. "The Essence of Functional Programming". In: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. Ed. by Ravi Sethi. ACM Press, 1992, pp. 1–14. ISBN: 0-89791-453-8. DOI: 10.1145/143165.143169. URL: http://doi.acm.org/10.1145/143165.143169.

[19] Philip Wadler. "Theorems for Free!" In: *FPCA*. 1989, pp. 347–359. DOI: 10.1145/99370.99404. URL: http://doi.acm.org/10.1145/99370.99404.

[20] Philip Wadler and Stephen Blott. "How to Make ad-hoc Polymorphism Less ad-hoc". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 1989, pp. 60–76. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75283. URL: http://doi.acm.org/10.1145/75277.75283.