Implementing Reflective Metaprogramming for Rosetta using InterpreterLib

Philip J. Weaver

Submitted to the Department of Electrical Engineering & Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master's of Science

Thesis Committee:

Dr. Perry Alexander: Chairperson

Dr. Arvin Agah

Dr. Nancy Kinnersley

Date Defended

2007/08/31

The Thesis Committee for Philip J. Weaver certifies
That this is the approved version of the following thesis:

Implementing Reflective Metaprogramming for Rosetta using InterpreterLib

Committee:

_____
Chairperson

_____

_____

_____
Date Approved

Abstract

Reflection and Metaprogramming offer a variety of benefits, including compile-time optimization, generic programming, and generative programming. I bring reflective metaprogramming to Rosetta, a systems-level design language focused on heterogeneous design. I compare Rosetta to other languages with reflective, metaprogramming, and staged programming features, then present a unique approach to implementing reflective metaprogramming. My solution is unique because i) it uses InterpreterLib, a Haskell library that implements Modular Monadic Semantics; ii) unlike most reflective metaprogramming languages, Rosetta is not an executable programming language; and iii) I implement reflection as a standalone elaborator, not as an integration into an existing interpreter.

Acknowledgements

Dr. Perry Alexander sparked my interest in functional programming when I learned Scheme and ML from him as an undergraduate student, and because of this I have continued to be fascinated with functional programming, particularly pure functional languages and monads. He has been my mentor and friend for two years, during which he has prepared me for the real world.

It was Garrin Kimmell who suggested I write the preprocessor and metaprogram that eventually became integrated into AlgC, and so I must give him credit for sparking my interest in generative programming and metaprogramming. Perry suggested that I investigate Rosetta's reflective system, and from this I have become extremely interested in generative programming, generic programming, metaprogramming, multi-staged programming and reflection.

Our lab, the Systems Level Design Group (SLDG), includes myself, Dr. Alexander, Nicolas Frisby, Garrin Kimmell, Mark Snyder and Jennifer Streb. Each of them has provided valuable technical and emotional support throughout my two years in the graduate program at KU. Throughout this paper, "we" typically refers to this group of people. For example, when I say "we maintain InterpreterLib", I am giving credit to this group. Garrin started InterpreterLib, and Nick has extended it greatly; in particular, Nick is responsible for the algebra combinators and generics support.

In addition to maintaining InterpreterLib, we also develop and maintain the toolchain for Rosetta, known as Raskell. We all chip in on maintaining the AST and parser. Mark is in charge of the typechecker, I write the elaborator, Garrin works on synthesis, Nick works on simulation, and Jeni works on verification.

I want to also thank Dr. Arvin Agah and Dr. Nancy Kinnersley for taking the time to serve on my thesis committee. Dr. Agah served as my undergraduate advisor, and Dr. Kinnersley as my teaching advisor, so I thank them both for their support.

Contents

Chapter 1

Introduction

One of the most fundamental things I learned as an undergraduate student is the importance of abstractions in Computer Science and Computer Engineering. The topics and techniques presented in this paper all have one thing in common: they are all intended to raise the level of abstraction in order to make development easier for the user. Furthermore, they facilitate team development in which persons can work in parallel on individual components of a complex system.

I begin with Haskell, a high-level functional language, and InterpreterLib, which is implemented in Haskell. InterpreterLib allows the user to design language processing tools at a very high level of abstraction. Furthermore, it facilitates development by allowing tools to be built as complex collections of smaller components, and allowing these highly modular tools to be generically modified, manipulated and extended.

Reflection allows the user to write generic algorithms for solving classes of problems and then specialize these algorithms for each particular problem. Furthermore, a reflective language does not require that the user learn an additional language, but simply write code in the existing language at a higher level of abstraction, treating the language itself as data that can be manipulated.

This thesis covers my implementation of reflection in Rosetta, a systems-level design language intended to facilitate heterogeneous design at a higher level of abstraction than current solutions. Rosetta allows teams to design in multiple domains and bring together information from these domains all in a single environment. We intend for Rosetta to allow developers to make important design decisions earlier on in the development process, reducing cost and speeding development. I have implemented Reflection in Rosetta as a standalone elaborator, written in InterpreterLib, that is invoked between two stages of typechecking. The

elaborator is a rewriter that resolves staging annotations and interpretation constructs statically. I also extended the typechecker to support reflection without modifying the original typechecker code.

1.1 InterpreterLib

Developing language interpreters and analyzers can be difficult and time-consuming. This is especially true when the target language is still in development and evolving alongside the tools. To alleviate this problem, we provide InterpreterLib [34, 40, 41], a Haskell library for writing modular, extensible language processing tools, such as interpreters, compilers, and other forms of analysis. It is developed and maintained by the Systems Level Design Group (SLDG) at the University of Kansas. InterpreterLib implements Modular Monadic Semantics (MMS), a technique for defining language syntax and semantics as a collection of smaller, composable languages. Abstract syntax for a language is represented as *syntactic functors*, and *semantic algebras* define denotations from a syntactic space to a value space in Haskell. InterpreterLib also provides several extensions to MMS, including *algebra combinators* and *generics*. They allow the user to reuse existing algebras to define complex semantics without modifying the original algebras.

InterpreterLib has proved its usefulness in in SLDG lab. We use InterpreterLib to write all of the tools for Rosetta, including the elaborator for reflection. Analyses are written independently, generically, and as complex compositions of smaller algebras, allowing us to work in parallel with each other on a constantly evolving language.

1.2 Metaprogramming

Metaprogramming is a broad category of programming techniques for writing programs that manipulate programs. In metaprogramming, the *metalanguage* operates on the *target language* or *object language*. The target language is exposed to the metalanguage as a data structure that programs written in the the metalanguage can operate on. Some common meta-programs are compilers, in which the metalanguage is source code and the target language is assembly, a netlist, or some other target, and the C/C++ preprocessor, in which the metalanguage consists of preprocessor directives and the target language is C/C++.

A language is *reflective* if the metalanguage and target language are the same. In

a high-level reflective language, the abstract syntax is exposed as a data structure within the language.[1] This allows reflective programs to treat code as data and generically rewrite and generate code.

Metaprogramming with reflection is cleaner and more powerful than using a preprocessor or macro expansion system. Preprocessors require the user to learn a new language, whereas reflection is implemented as an extension to an existing language. Macros can mostly only be used for text replacement and substitution. Reflective programs are more powerful because they can inspect and manipulate code, rather than just generate code.

Reflection can occur at compile time or run time. This thesis focuses on high-level compile-time reflection that offers the following benefits:

- Partial evaluation, conditional compilation, and optimization. Statically known parameters may be evaluated at compile time, debugging code may be conditionally compiled, and compile-time optimizations may be performed, thus offering performance gains with little effort from the programmer.
- Generative programming. The algorithmic construction of programs leads to greater code reuse and reliability.
- Generic programming. Algorithms can be written generically for a large set of tasks, at a high level of abstraction, and then specialized to certain tasks at compile time with no run-time performance penalty.

Reflection can be implemented using staging annotations. A *quasiquote* construct turns off evaluation and directs the compiler to treat its argument as syntactic data rather than semantic code. This allows that syntax to be operated on just like any other data in a program. An *unquote* or *splice* turns evaluation back on and inserts the result into a syntactic quasiquoted term. Statically staged reflective languages include datatype definitions for representing their own abstract syntax, along with constructors for creating abstract syntax and some mechanism for manipulating abstract syntax. Finally, some sort of *run* or *eval* construct translates syntax into a semantic value by directing the compiler to evaluate the syntax as code in the language at compile time.

[1] Reflection can also refer to a very low-level technique, in which programs operate on their own binary or byte code in memory at run-time. This is an entirely different kind of reflection that is not the topic of this thesis.

1.3 Rosetta

Rosetta [1] is a systems-level design language supporting heterogeneous specification. This includes both functional and non-functional constraints, such as timing, resource utilization, security and safety. Rosetta is meant to facilitate higher-level design than current specifications languages, and to allow the user to find and resolve issues earlier on in the design process. Rosetta tools are written in Haskell and developed and maintained by the Systems Level Design Group (SLDG) at the University of Kansas.

A Rosetta specification consists of a series of declarations. Basic declarations are written in the following syntax:

```
identifier :: type is definition;
```

Identifiers can be declared without being defined, in which case the optional `is` clause can be left out or written as `is constant`. Rosetta features dependent types [1], so types are first-class values in Rosetta and this declaration syntax can be used to declare both term variables and type variables. More information on dependent types can be found starting on page 462 of Pierce's book [30].

Rosetta supports 2-stage static reflection. Alexander provides an overview of Rosetta's reflective system in chapter 13 [1], which is summarized in section 5.3 of this paper. Section 6.2 discusses the elaborator, a set of modules in the Rosetta toolchain that implement reflection in Rosetta.

Rosetta provides algebraic datatypes, which are used to represent Rosetta's own abstract syntax in order to support reflection. Algebraic datatypes are introduced for Haskell in section 2.2.2. Algebraic datatypes in Rosetta differ only slightly from Haskell and are discussed in section 5.3.

The more advanced features of Rosetta are not necessary for understanding the work presented in this paper.

1.4 Organization and Contributions

The purposes and organization of this paper are as follows:

- Introduce the features of Haskell that are needed to understand this thesis (chapter 2).

- Provide an introduction and tutorial of MMS and InterpreterLib (chapters 3 and 4).
- Demonstrate InterpreterLib by defining an interpreter for a toy language, and extending it using algebra combinators and generics (chapter 4).
- Introduce metaprogramming, multi-stage programming, and reflection, and give an overview of these features in several languages, including Rosetta (chapter 5).
- Discuss two InterpreterLib implementations of metaprogramming: i) multi-stage programming for the toy language; and ii) staged reflection for Rosetta (chapter 6).

InterpreterLib was written before I started my research, though I have helped maintain and improve it over the last two years. My biggest contributions to InterpreterLib have been AlgC and MonadBuilder. AlgC is a preprocessor for generating boilerplate code for InterpreterLib. I improved it by allowing it to directly parse recursive AST definitions in Haskell and automatically generate a non-recursive AST for InterpreterLib, significant boilerplate code, and a function to convert from the recursive AST to non-recursive AST. MonadBuilder automates the building of specific combinations of monads using Nick Frisby's HoFix solution (section 4.3).

I studied metaprogramming with InterpreterLib, including the switch algebra and reification, and wrote examples similar to those in section 6.1 with Garrin Kimmell for two of our conference papers [40, 41]. This led to the elaborator, a standalone analysis in the Raskell toolchain that resolves reflection, which I alone wrote. Its implementation is discussed in section 6.2.

Chapter 2

Haskell

Haskell [10, 28, 29] is a declarative functional language that is both lazy and pure. Haskell is known as a purely functional language because is excludes mutable variables. Conventional imperative languages, such as `C`, `C++` and `Java`, define the `=` symbol as an assignment operator that destructively updates the state of a variable. Most functional languages avoid the use of mutable data, but Haskell forbids it entirely. The `=` symbol in Haskell is used to declare the value of a persistent, immutable identifier, whose value can never change.

The limitations of such a strict system seem immediately obvious. However, purely functional languages exhibit an important property known as *referential transparency*. A declaration in Haskell defines a true equality, that wherever the identifier is in scope it can be replaced with the right hand side of its declaration without changing the behavior of a program. For example, in a program with the declarations

```
f = x + y
g = f
```

the expressions `x + y`, `f` and `g` can be used interchangeably, regardless of the values of `x` and `y`.

It is important to understand how variables are bound in Haskell. The declaration `f = x + y` does not evaluate `x` and `y` and bind `f` to their sum; instead, it binds `f` to the expression, or computation, `x + y`. When programming in purely functional languages such as Haskell it is common to view a variable or expression not as a value but as a *computation* that will yield a value if and when it is evaluated. This is especially true when dealing with monadic computations (see section 2.5).

In strict evaluation expressions are evaluated when bound to a variable. Because

function calls bind arguments to formal parameters, the arguments to a strict function are evaluated before the function is applied. This of course means that unused arguments are needlessly evaluated. Also, divergent computations that are never used are still evaluated and thus diverge in a strict language. Most strict languages use non-strict evaluation for control structures such as if statements; the branch to an if statement that is not taken should never be evaluated. Also, a kind non-strict evaluation called *short circuiting* is used in languages such as `C` and `C++` to avoid unnecessary evaluation of arguments to Boolean operators.

Haskell is entirely non-strict because all bound expressions are evaluated only if they are used. Furthermore, Haskell is lazy because evaluation is delayed until the moment that the result of a computation is needed. One way to implement non-strict evaluation is known as *call by name*. In this evaluation strategy function arguments are evaluated every single time they are used. This is of course inefficient because the same computation may be evaluated multiple times, even if it yields the same result each time. Fortunately, referential transparency guarantees that a computation will produce the same result every single time it is evaluated. Thus, Haskell only evaluates computations once and *memoizes* the result at run time; all subsequent uses of a computation use the memoized value. This evaluation strategy is known as *call by need*, and is only possible if referential transparency is guaranteed.

Haskell is a purely functional language, so side effects are forbidden. To the average imperative programmer, this has some disturbing implications, such as no mutable state and no IO. However, almost any side effects can be simulated in Haskell using a monad. I say "simulated" because almost all monads are just abstractions of existing Haskell data structures, particularly lambda abstractions. For example, global state is simulated by the State monad using a lambda abstraction of the type `s` $\rightarrow$ `(s, a)`, a function that takes in the initial state and yields a value and a final state. To simulate side effects by manually managing such abstractions would be extremely tedious and unacceptable, as they would have to be explicitly passed into each function that intends to produce or use the effects. Fortunately with monads in Haskell, the details are hidden from the programmer and side effects occur quite transparently. This is accomplished via type classes.

The remainder of this chapter covers the aforementioned features of Haskell that are necessary for understanding this thesis.

2.1 Functions

A function in Haskell is represented as a *lambda abstraction*. For example, the abstraction `(\x y →x + y)` takes in formal parameters `x` and `y` and yields their sum. One can name this function by declaring it as follows:

```
add = \x y -> x + y
```

However, it is often more convenient to define functions in the following style. This declaration is equivalent to the one above:

```
add x y = x + y
```

Functions can be partially applied, so `add 1` is equivalent to `(\y →1 + y)`. This allows us to define functions such as the following:

```
inc = add 1
```

In functional languages, lambda abstractions are first-class values that may be passed around like any other value. Thus, one can write higher-order functions that are parameterized over other functions. A classic higher-order function is the `map` function, which takes in a function and a list and applies that function to every element in the list. Thus, `map inc` is a function that takes in a list of numbers and adds one to every element in that list.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Lists are discussed in the next section. The type of `map` is discussed in section 2.3.

Operators are just functions that are applied using infix notation, and have different naming restrictions (in general, operators are made of symbols and functions are made of letters). An infix operator may be turned into a prefix operator by surrounding it in parentheses. A function may be turned into an infix function by surrounding it in backquotes. For example, the `div` and `mod` functions, which perform integer division and modulo, are more commonly used in infix notation, such as in the following function that converts a number of seconds into minutes and seconds:

```
f seconds = (sec, min)
    where sec = seconds `mod` 60
          min = seconds `div` 60
```

Function composition is accomplished via the dot `(.)` operator

```
(f . g) x = f (g x)
```

which allows for code to be written in a point-free style. Consider the definition of an `even` function:

```
even x = x `mod` 2 == 0
```

Alternatively, one can use both `(`mod` 2)` and `(==0)` as unary functions and compose them to define `even` in point-free style:

```
even = (==0) . (`mod` 2)
```

2.2 Data Structures

Haskell provides built-in collection types and allows for user defined datatypes, particularly algebraic datatypes. Lazy evaluation allows for infinite data structures and data structures containing divergent computations. We say that the value of a divergent computation is *bottom*, which in Haskell is `undefined`. The term `undefined` is a computation that, if evaluated, will cause Haskell to crash.

2.2.1 Tuples and Lists

Haskell provides two basic data structures for storing collections of multiple values in one value: *lists* and *tuples*. A list is a homogeneous, variable length structure; that is, every element in a list must be of the same type and the length of a list may change. This does not mean that one may change the value of a list, because no variable may be changed in Haskell. Instead, this means that one can construct new lists by adding or removing elements from existing lists, or concatenating two lists. Haskell includes the following list constructs, all of which yield a list: i) `[]`, the null list constructor; ii) `(:)`, the cons operator that adds an element to the beginning of a list; and iii) `(++)`, the concatenation operator that appends one list to another. On the other hand, a tuple is a heterogeneous, fixed length structure, constructed as a comma-separated parenthesized list. The size of a tuple and the type of each element are prescribed in its type, so one cannot write a function that operates on arbitrarily-sized tuples. The type of each element in a list is prescribed in its type. The following are example lists and tuples along with possible ascriptions of their types:

```
[1, 2, 3] ++ [4,5] :: [Int]
1:[2, 3] :: [Int]
('a', False) :: (Char, Bool)
(1, 'b', 'c') :: (Int, Char, Char)
```

The Haskell Prelude includes the following built-in functions for extracting values from lists and tuples. Although each of these functions extracts the first or remaining part of a collection, `head` and `tail` operate on lists of *any size*, whereas `fst` and `snd` operate only on tuples of length 2.

```
head (x:xs) = x
tail (x:xs) = xs

fst (x, y) = x
snd (x, y) = y
```

2.2.2 Algebraic Datatypes

An algebraic datatype is a sum of 0 or more products and is defined using the `data` keyword. Each product combines 0 or more types into a single type and is constructed using a *constructor*. An algebraic datatype may consist of a single product, and thus a single constructor, such as the following, which combines a person's name and age into a single type:

```
data Person = P String Int
```

This defines `P` as a constructor that takes in a `String` and an `Int` and yields a `Person`. Thus, `P :: String` →`Int` →`Person`.

Another specific kind of algebraic datatype is an enumeration, in which all constructors are nullary. The built-in `Bool` type is an example of an enumeration:

```
data Bool = True | False
```

Haskell includes an `Either` type that sums any two types together:

```
data Either a b = Left a | Right b
```

This will be used in chapters 3 and 4 to sum language constructs, evaluation strategies, and semantic spaces.

An algebraic datatype can represent the abstract syntax for a language. The following datatype represents a simple expression language with arithmetic, Boolean, and lambda expressions.

```
data Expr = Num Int
          | Add Expr Expr
          | Tru
          | Fls
          | If Expr Expr Expr
```

```
            | Var String
            | Lambda String Expr
            | App Expr Expr
```

Datatypes may be parameterized over other types. See section 2.3.1 for further discussion of parameterized types. The list type is one of the most common algebraic datatypes. It is the sum of two elements: i) the null list and ii) a product of one element and another list. We can define `List` in Haskell as a parameterized type:

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

This declares constructors `Nil` and `Cons` with the following types:

```
Nil :: List a
Cons :: a -> List a -> List a
```

This `List` type is identical to the built-in `[]`, where `Nil` is `[]` and `Cons` is `(:)`.

Another common data structure is a binary tree, represented as the following type:

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

In this case the constructors `Leaf` and `Branch` have the following types:

```
Leaf :: a -> Tree a
Branch :: Tree a -> Tree a -> Tree a
```

Any algebraic datatype with only one constructor and only one parameter to that constructor may be written as a `newtype`. The syntax for a `newtype` is the same as for a `data`. Datatypes declared using `newtype` are represented more efficiently in Haskell than those declared using `data`.

2.2.3 Infinite Data Structures

Haskell's lazy evaluation allows the user to construct divergent computations, including infinite data structures. Only those computations that are actually used are evaluated. For example, consider the following list that crashes when its 4th argument, `54 ‘div‘ 0`, is evaluated. The result of printing this list to the screen is shown below. Notice that Haskell evaluates the first 3 elements of the list and displays them, then crashes on the 4th.

```
map (54 `div`) [3, 2, 1, 0, -1, -2, -3]

> [18,27,54,*** Exception: divide by zero
```

The only reason Haskell crashes in the above example is because the 4th element is evaluated. We can, however, ask for only the first three elements using the `take` function, in which case the result is successfully calculated:

```
let list = map (54 `div`) [3, 2, 1, 0, -1, -2, -3]
in take 3 list

> [18, 27, 54]
```

The list is bound in a let expression simply to show that variables can be bound to divergent data structures as long as the divergent computation itself is never executed.

The following are all valid infinite lists in Haskell. The `take`, `head` and `tail` functions and the list index operator, `(!!)`, can all be used to extract parts of these infinite lists.

```
ones = 1 : ones
numsFrom n = n : numsFrom (n+1)
squares = map (^2) (numsFrom 0)
```

Any recursively defined algebraic datatype can easily be instantiated into an infinite data structure. The following defines `t n`, an infinite tree with a `(Leaf n)` on the left and an infinite tree `(t (n+1))` on the right.

```
t n = Branch (Leaf n) (t (n + 1))
```

2.3 Types

Haskell is strictly typed; that is, the type of every term in Haskell must be known statically. The Haskell type checker infers the type of each term unless the type is explicitly ascribed by the programmer.

The `+` operator is overloaded in Haskell, thus the above `add` function is too. One can ascribe the type of `add` to be less polymorphic than the type inferred by Haskell. First, one can ascribe the types of `x` or `y`, allowing Haskell to infer the result of the function. This can be accomplished with either of these two declarations:

```
add (x :: Int) (y :: Int) = x + y
add x y = (x :: Int) + (y :: Int)
```

Or, one can restrict the overloaded `+` operator to a specific type[1]:

```
add x y = ((+) :: Int -> Int -> Int) x y
```

The most common way to ascribe the type of a function is to declare its signature immediately before declaring its value:

```
add :: Int -> Int -> Int
```

If the `add` function is ascribed such a specific type, and `inc` is defined in terms of `add` as it is above, then Haskell will also infer a specific type, `Int` $\rightarrow$ `Int`, for the the `inc` function.

Haskell includes a unit type, `()`. It includes a single term, `()`. So, `() :: ()`. The unit type is used as the result of a computation that does not yield a value.

2.3.1 Kinds

As the name implies, a type constructor constructs a type from zero or more other types. The simplest type constructors are the nullary types such as `Int` and `Bool`. A common unary type constructor is `[]`, which constructs a list type from a base type. One can apply `[]` to `Int` to yield `[Int]`, the type of a list of integers.

Type constructors are applied to types much as functions are applied to terms. Just as a function has a type, a type constructor has a *kind*. All nullary type constructors are of kind `*`, whereas unary constructors are of kind `*` $\rightarrow$ `*` and binary constructors are of kind `*` $\rightarrow$ `*` $\rightarrow$ `*`. For example, in section 2.2.2, the types `List` and `Tree` are unary type constructors of kind `*` $\rightarrow$ `*`, and all the other types are nullary constructors of kind `*`. Kinds are inferred and almost always hidden from the programmer. However, if necessary the programmer may ascribe kinds using the same syntax for ascribing types.

The simplest way to define a type constructor is via a *type synonym*. This does not really introduce a new datatype as in section 2.2, but rather it defines an alias from one type to another. A simple example is the following unary type, `Name`, which is defined as an alias to `String`:

```
type Name = String
```

[1] In order to ascribe the type of `+`, it must be parenthesized, turning it into a prefix operator.

Type synonyms may take on other kinds besides `*`, as in the following example that declares a type for representing an associative list. The keys in the associative list are fixed as `String`, but the values are parameterized.

```
type AList val = [(String, val)]
```

This type is often used to represent an environment when writing a language interpreter, in which variable names are bound to values.

2.3.2 Parametric Polymorphism

Haskell supports parametric polymorphism through universally quantified type variables. The actual type of the `map` function above is

```
forall a b . (a -> b) -> [a] -> [b]
```

which says that this function is defined for all types `a` and `b`.

Universally quantified variables are implied, and the `forall` is usually left off. However, it is necessary if the user wants to use a type variable to ascribe the type of a term within a functions definition. This is sometimes necessary in the highly modular, polymorphic code that can be written using InterpreterLib. In such cases, the "forall" is necessary so that Haskell knows that the ascribed types within the definition are the same as the types ascribed to the function itself.

2.3.3 Ad-hoc Polymorphism

Ad-hoc polymorphism, also known as overloading, is implemented in Haskell using type classes. A type class is a collection of types for which some functions are defined.

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y     = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y     = not (x == y)
```

This class declares that if `a` is an instance of `Eq`, then functions `(==)` and `(/=)` exist with the given types. Default definitions can be provided so that individual instances only have to overwrite one of the functions. One can then define ad-hoc instances of `Eq` for specific types. The Haskell Prelude includes built-in instances of `Eq` for all primitive types, tuples up to length 12, and lists. As an example,

the instance of `Eq Int` might look something like this, where `primIntEq` could be a non-overloaded function defined specifically over integers:

```
instance Eq Int where
  (==)      = primIntEq
```

Also, instances of `Eq` for 2-tuples and lists may be defined as follows:

```
instance (Eq a, Eq b) => Eq (a, b) where
  (a, b) == (c, d)   = a == c && b == d

instance Eq a => Eq [a] where
  []     == []       = True
  (x:xs) == (y:ys)   = x == y && xs == ys
```

The `=>` symbol literally reads as an implication. So, the last instance above states that if `a` is an instance of `Eq`, then `[a]` is an instance of `Eq` with the given definition of `(==)`.

Oftentimes a function depends on the existence of a class instance. For example, the `elem` function that checks if a value exists in a list, and thus has type `a → [a] → Bool`, may only be defined if an instance of `Eq a` exists. This *class constraint* is therefore part of the type of `elem`:

```
elem :: (Eq a) => a -> [a] -> Bool
```

In order to display any value of a particular type, there must be a way to convert that type to a string. The `Show` class defines a `show` function to do this. Instances of `Show` are defined for most built-in types.

```
class Show a where
  show :: a -> String
```

2.4 Functors

A functor is a type `f` of kind `* → *` with a structure preserving function `fmap` of type `(a → b) → f a → f b`. The class declaration for a functor in Haskell is:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Notice that a list is a functor and `fmap` is just a generalization of `map` for arbitrary functors. There are several ways to interpret the `fmap` function. One is that it *lifts* a function of type `(a → b)` into a functor. Another is that it takes in a

function and a data structure and maps that function everywhere it can in that data structure. Let's define an instance of `Functor` for the `Tree` datatype from section 2.2.2:

```
instance Functor Tree where
  fmap f (Leaf a)              = Leaf (f a)
  fmap f (Branch left right)   = Branch (fmap f left) (fmap f right)
```

Notice the strategy for mapping a function onto a tree, which is the same as the strategy for mapping onto a list. The function `f` is applied at the non-recursive nodes and mapped into the recursive nodes. This strategy actually differs from the one used to define functors for language processors in the next chapter.

The `fmap` function always preserves the structure of a functor. Every leaf remains as a leaf and every branch remains as a branch of the same size. However, the type in the functor, referred to as the *hole* or *carrier*, may change. Consider the function `fmap (>0)`, which replaces positive numbers with `True` and all other numbers with `False` in *any* functor. This function converts the carrier from a numeric type to a Boolean type.

Section 2.2.3 provided examples of infinite data structures. Just like the `map` function, the `fmap` function can operate on infinite data structures. If `t` is an infinite tree such as the one defined in section 2.2.3, then one can construct another infinite tree, `fmap even t`, which maps the predicate function `even` onto the infinite tree `t`.

2.5 Monads

Monads capture computational effects in a purely functional language. Every monad is a functor whose carrier is a value. A monadic computation consists of two things: i) computation effects; and ii) the result of the computation, a value. To *execute* or *run* a monadic computation means to apply the effects and yield a value. So for a monad `m`, we say that monadic computations are of type `m a`, which may encapsulate one or more effects, and when executed yield values of type `a`. A monad `m` may be an instance of one or more monad classes.

This paper only discusses monads in enough detail to understand the implementations of reflection presented in this thesis. Wadler provides an excellent introduction to monads [39] to supplement this section. The following monads are relevant to this paper:

Monad	Computational Effect
Identity	None
Maybe	Failure
State s	Global state of type s
Reader e	Local context of type e
Writer w	Monoid of type w
IO	File and user IO, system calls, etc.

Every monad implements the following interface:

```
class Monad m where
  return  ::  a -> m a
  (>>=)   ::  m a -> (a -> m b) -> m b
```

The function `return` lifts a pure value into the monad, producing a computation that will yield that value without any effects. The *bind* operator executes a monadic computation and then applies a monadic function to the result, yielding a new monadic computation. So, `x >>= f` executes `x`, including its effects, yielding a value, which it then applies `f` to. The bind operator is very much a *sequence* operator, like a semicolon in an imperative language, except that it propagates resultant values from one computation to another. To sequence effect-only computations, ignoring the values they may or may not produce, another bind operation exists:

```
x >> y   =   x >>= \_ -> y
```

This operator more closely resembles the semicolon from imperative languages, because in an imperative language each statement produces only an effect and no value.

The second argument to `>>=`, the function `f`, must yield a monadic computation. If `f` is a pure function of type `a` $\rightarrow$ `b`, then we can make it monadic by applying `return` after it. This is a very common operation, and a function has been defined for it:

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = m >>= return . f
```

Every monad literally is a functor. This `liftM` function is actually `fmap`; it lifts a function into a monad just like it lifts any function into a functor. It is useful because it allows us to apply a pure function to the value within a monad in the context of the computational effects that are currently encapsulated in that monad; that is, the function will receive a value that has been calculated from those computational effects.

Although concise, the bind notation presented above can be difficult to use. Fortunately, Haskell provides a more natural "do" notation that translates directly to the bind notation. The simplest example of this notation is the following, which is shorthand for `x >>= f`:

```
do x' <- x
   f x'
```

If we want to sequence three monadic computations - `x`, `y`, and `z` - together and provide the result of `x` to the function `z` we would have to use a lambda abstraction like this: `x >>= (\x' →(y >> z x'))`. This is much cleaner when written using the do notation:

```
do x' <- x
   y
   z x'
```

For large, complicated functions the do notation can be far easier to understand.

So far we have only seen how to sequence computational effects, but not how to *produce* them. Each monad comes with a set of *non-proper morphisms*, functions that manipulate and use the monad's computational effects. Additionally, most monads have a "run" function of type `m a` → `a` that executes the monad, yielding a value that is calculated using the monadic effects.

Instances of `Monad` should obey the following laws[2]:

1. `return a >>= f = f a`
2. `m >>= return = m`
3. `(m >>= f) >>= g = m >>= (x→f x >>= g)`

2.5.1 Identity Monad

The Identity monad, the simplest monad of all, produces no side effects. Without these computational effects, the bind operator is simply function composition. The function `runIdentity` simply pulls the value out of the monad.

2.5.2 Maybe Monad

The following datatype, defined in the Prelude, can be made into a monad:

[2]Should I remove this from the thesis?

```
data Maybe a = Nothing | Just a
```

In the Maybe monad, computations either fail, yielding `Nothing`, or succeed and yield `Just a`, where `a` is the value produced. The `Monad` instance for `Maybe` is:

```
instance Monad Maybe where
  return = Just
  Nothing >>= _   = Nothing
  Just a >>= f    = f a
```

When a computation in the Maybe monad fails, the failure takes over and all subsequent computations are ignored. Let's go back to the example from section 2.2.3:

```
map (54 `div`) [3, 2, 1, 0, -1, -2, -3]
```

We can write a function which takes in a number and divides 54 by it, but fails when the number is 0:

```
f 0 = Nothing
f n = Just (54 `div` n)
```

So if we map `f` onto the list we get back the following:

```
[Just 3, Just 2, Just 1, Nothing, Just -1, Just -2, Just -3]
```

We can intersperse the bind operator, `>>`, between each computation using the `sequence` function, so that if any of the computations in

```
map f [3, 2, 1, 0, -1, -2, -3]
```

fail, the entire computation will fail. The type of `sequence` is:

```
sequence :: Monad m => [m a] -> m [a]
```

Or we can use the `mapM` function, where `mapM f` is the same as `sequence . map f`:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

2.5.3 State Monad

The State monad captures computations that involve a global state. It is represented as a lambda abstraction that is parameterized over an initial state and yields a value and a final state. The lambda is wrapped up in a data constructor using a `newtype`:

```
newtype State s a = State (s -> (a, s))
runState (State x) = x
```

The `runState` function simply pulls the constructor off, yielding a function that is parameterized over the initial state. Thus, `runState m x` means "execute m with an initial state x". The state can be any type in Haskell, such as an integer or an associative list mapping names to values. The instance of `Monad` for `State s` is:

```
instance Monad (State s) where
  return a    = State (\s -> (a, s))
  x >>= f     = State (\s -> let (v, s') = runState x s
                             in runState (f v) s')
```

So, `return a` is a monadic computation that preserves the state and will yield the value `a` when executed. Clearly, the bind operation is more complicated. To bind `x` and feed its result to `f`, we must yield a monadic computation that does the following:

1. Execute `x` with the current state, which yields a value and a new state.
2. Apply `f` to that value, which yields a monadic computation.
3. Run that monadic computation on the new state.

This causes the stateful effects of `x` to be felt during the execution of `f v`. This definition of bind really does sequence effects!

Without ways of extracting and updating the global state, the State monad is nothing but the Identity monad. The following non-proper morphisms provide this necessary functionality.

```
class Monad m => MonadState s m where
  get :: m s
  put :: s -> m ()

instance MonadState s (State s) where
  get    = State (\s -> (s, s))
  put s  = State (\_ -> ((), s))
```

So, `get` is a monadic computation that preserves the state and yields a value equal to the state, so that if we bind a "get" we will receive the current state. And `put s` is a monadic computation that ignores the current state and overrides it with `s`, and does not yield any value. Because `put` only produces effect and does not yield

any value, it often appears on the left-hand side of `>>`, the bind operation that ignores the result of its left-hand side.

Consider the following function that takes in an integer and sets the state to this integer only if the integer is greater than the current state. This function yields a computation of type `m ()` because it only modifies the state and does not return any value.

```
setMax n = do x <- get
              when (n > x) (put n)
```

The function `when` is simply defined as

```
when condition m = if condition then m else return ()
```

If we sequence `setMax` across a list of numbers, the result will be a monadic computation whose state is the maximum value in that list, assuming we provide an initial state that is small enough.

```
runState (mapM setMax [1,5,0,4,1,3]) 0

> ((), 5)
```

The user sometimes only wants back the final state or the final value, such as in the previous example, so the following functions provide these options:

```
evalState m s = fst (runState m s)
execState m s = snd (runState m s)
```

So, the built-in `maximum` function can be re-written using the State monad:

```
maximum [] = undefined
maximum (x:xs) = execState (mapM setMax xs) x
```

The State monad can be used to write an efficient, yet easy-to-read implementation of the Fibonacci sequence. This recursive definition of `fib` is of course horribly inefficient:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

However, the following efficient implementation, which tracks the previous two values in the sequence by passing them into the function, is somewhat messy:

```
fibHelper (x, y) 0 = x
fibHelper (x, y) 1 = y
```

```
fibHelper (x, y) n = fib (y, x+y) (n-1)

fib = fibHelper (1, 1)
```

Instead, we can implement the Fibonacci sequence by hiding the previous two values in the sequence, the pair of `x` and `y`, in a State monad. The following `fibOnce` function updates the state of the `(x, y)` pair and yields no value:

```
fibOnce = do (x, y) <- get
             put (y, x + y)
```

We can calculate any number in the Fibonacci sequence by replicating `fibOnce`:

```
runFib n = execState (replicateM n fibOnce) (1, 1)
```

where `replicateM n` takes in a monadic computation and executes it `n` times. It uses `>>` to bind each computation, so it ignores the value of each computation, but propagates the effects. Notice that both solutions have to feed the initial pair, `(1, 1)`, to their helper function, but in the monadic solution's helper function, `fibOnce`, the pair is hidden in the monad.

2.5.4 Environment Monad

The Environment monad, called `Reader` in Haskell, is similar to the State monad except that a change to the environment is localized to a monadic computation. This leads to a slightly simpler definition than the State monad because computations do not yield a new state, only a value. Whenever a computation finishes, the environment effectively returns to what it was before the computation executed. So, the Environment monad is represented as a lambda abstraction that is parameterized over the initial environment and yields a value.

```
newtype Reader e a = Reader (e -> a)
runReader (Reader x) = x

instance Monad (Reader e) where
  return a   = Reader (\e -> a)
  r >>= f    = Reader (\e -> runReader (f (runReader r e)) e)
```

So, `return a` is a monadic computation that ignores the environment and will simply yield `a` when it is executed. And, `r >>= f` is a monadic computation that executes `r` in the current environment, then executes `f` on that result in the same environment. Of course, `r` may contain a computation that changes the environment, but such changes will not be seen by `f` since they are local to `r`.

The functions specific to the Environment monad are `ask`, which is analogous to `get` for the State monad, and `local`, which evaluates a monadic computation in a modified environment.

```
class Monad m => MonadReader e m where
  ask :: m e
  local :: (e -> e) -> m a -> m a

instance MonadReader e (Reader e) where
  ask        = Reader (\e -> e)
  local f c  = Reader (\e -> runReader c (f e))
```

The `ask` function is simply a monadic computation that yields the current environment. The `local` function takes in a function that modifies the environment, and a monadic computation. It applies that function to the environment and then executes the monadic computation in that new environment.

2.5.5 IO Monad

The IO monad is used to implement a large number of effectful operations. These include file and user IO, random number generation, time of day operations, system calls, external C/C++ calls, and so on. Unlike other monads, the IO monad cannot be implemented in Haskell; its implementation is hidden within the compiler. It is also the only monad that does not have a safe run function, as IO-monadic operations cannot be escaped. There are rare exceptions to this rule, such as externally calling a pure C/C++ function.

Every compiled Haskell program has a top-level `main` function of type `IO ()`, since no program can do anything useful without performing some kind of IO.

Some common IO functions are

```
putStr :: String -> IO ()                -- print string to screen
getLine :: IO String                     -- read string from keyboard
readFile :: FilePath -> IO String        -- read file into a string
writeFile :: FilePath -> String -> IO () -- write string to a file
system :: String -> IO ExitCode          -- execute shell command
getArgs :: IO [String]                   -- read command-line args
```

As an example, the following main function reads a filename from the command line, reads in that file, and prints the first line of that file to the screen.

```
main = do
  args <- getArgs
```

```
case args of
  (name:_) -> readFile name >>= putStr . head . lines
  _ -> error "I␣need␣an␣argument"
```

The `lines` function simply splits a string at every newline character, yielding a list of strings.

2.5.6 Writer Monad

The Writer monad adds results to a *monoid* as a side effect while calculating values. For this thesis, lists are the only monoids used, for which the add operation is concatenation. When a Writer monad computation is executed, it yields a pair of the resultant value and the generated list. The datatype for representing the Writer monad is the following:

```
newtype Writer w a = Writer (a, w)
runWriter (Writer x) = x
```

The Writer monad is commonly used to produce string output as a side effect while computing values. The Writer monad only produces such output; it does not display it to any file descriptor or device as the IO monad does.

For brevity, the implementation of the Writer monad is not provided, and it is assumed that `w` is always a list. There are two functions associated with the Writer monad that are used in this thesis:

```
tell :: (MonadWrtier w m) => w -> m ()
tell w = Writer ((), w)

listen :: (MonadWrtier w m) => m a -> m (a, w)
listen x = Writer (let (a, w) = runWriter x
                   in ((a, w), w))
```

The `tell` function appends to the current list. The `listen` function "listens" to a monadic computation, extracting both the value and the current list from it, and yields a monadic computation whose value is the pair of that previous value and list. The `listen` function is essentially just like `runWriter` except that it does not escape out of the monad.

In the following example, the Writer monad is used to produce a log of actions that occur during a computation. Every time a multiplication occurs on the factorial function below, a message is appended to the Writer monad. This message can be a string, or just the value returned by the recursive call.

```
fact 0 = return 1
fact n = do x <- fact (n-1)
            tell [x]
            return (n * x)
```

Thus, the value of `runWriter (fact 7)` is the factorial of 7 paired with a list of all values "told" to the monad:

```
(5040,[1,1,2,6,24,120,720])
```

2.5.7 Monad Transformers

A monad `m` may be an instance of many different specific monads stacked on top of each other. Functions can use return and bind generically without knowing specifically which monad they are using or how they are stacked, and they may use non-proper morphisms from multiple different monads simultaneously. This is accomplished using *monad transformers* [6, 20], type classes that define transformations between monads.

Consider the following function that generates uniquely tagged strings using the State monad, and "tells" them in a Writer monad. The class constrains in its type indicate that `m` is both a State monad that carries an integer and a Writer monad that carries a list of integers.

```
f :: (MonadState Int m, MonadWriter [Int] m) => m String
f = do n <- get
       put (n + 1)
       tell [n]
       return ("x" ++ show n)
```

So, if the initial state is 0, then `f >> f >> f` should yield a final state of `3`, the list `[0, 1, 2]`, and the value `"x2"`. Each of the monadic run functions has a transformer version ending in `T`. For example, either of the following can be used to execute `f >> f >> f`:

```
run0 = runWriter (runStateT (f >> f >> f) 0)
run1 = runState (runWriterT (f >> f >> f)) 0
```

In `run0`, `runStateT` yields a state-value pair, which is then used as the value in the value-list pair result of `runWriter`, so the value of `run0` is `(("x2",3),[0,1,2])`. In `run1`, `runWriterT` first yields its pair, which is then used as the value in the state monads pair, so the value of `run1` is `(("x2",[0,1,2]),3)`.

2.5.8 Value Recursion

The let expression in Haskell is a recursive let, often called *letrec* in other language. All identifiers declared in a let are in scope to each other and themselves, regardless of the other in which they are made, just like all the declarations that appear in a Haskell module are available to each other.

As an example, consider the factorial function defined as two mutually recursive functions:

```
let f x = if x == 0 then 1 else x * (g (x - 1))
    g x = if x == 0 then 1 else x * (f (x - 1))
in f 10
```

As another example, suppose we are defining an interpreter for a language in which `evalDecl` evaluates a single declaration and returns the binding from it, and `evalDecls` evaluates a list of declarations and returns the list of bindings from it.

Mapping `evalDecl` across the list of declarations yields a list of bindings. But we want to evaluate those declarations in the context of the bindings generated by those declarations, thus allowing recursion in the language. Haskell's recursive let expression allows this. The newly generated bindings are prepended to the current environment, and the declarations are each evaluated in this new context.

```
evalDecl env decl = ...

evalDecls env decls =
   let newbindings = map (evalDecl (newbindings++env)) decls
   in newbindings
```

This interpreter can be made monadic by managing the environment in the Reader monad. The `do` notation does not allow the recursion that the let expression does, but an `mdo` construct [4, 5] brings value recursion to monadic computations. The above functions can be rewritten monadically as follows:

```
evalDecl decl = ...

evalDecls decls = mdo
  newbindings <- local (newbindings++) (sequence (map evalDecl decls))
  return bindings
```

Using `mdo` like this is exactly how analyses for Rosetta are written in InterpreterLib, including the reflective system in section 6.2.

Just as `do` is shorthand for the monadic bind, `>>=`, `mdo` is actually shorthand for a more complicated `mfix` construct. This thesis only uses the `mdo` notation, so it is not necessary to understand the `mfix` function.

Chapter 3

Modular Monadic Semantics

Modular Monadic Semantics (MMS) is a technique that has been used to structure denotational semantics for some time now. MMS allows the programmer to define the syntax and semantics for individual families of language constructors, and systematically combine those separate definitions into a complete language. This lends to development of highly modular language processors because features may be added and removed without affecting the rest of the system. Also, denoting to monads in a purely functional language such as Haskell preserves the important property known as referential transparency.

This chapter provides an introduction to MMS that is sufficient for understanding the implementations in this thesis. A more thorough review can be found in the literature [3, 6, 8, 11, 15, 19, 20, 23, 24, 33, 39]

The abstract syntax for a language is represented using *syntactic functors*. Denoting a language is accomplished by mapping an evaluation strategy onto a functor using `fmap`. Instances of `Functor` can be systematically derived from the structure of the functor's datatype declaration. Any two functors may be combined into a *composite functor*. A composite functor is literally just the sum of two functors, and is itself a functor with a `fmap` function.

Value spaces can be constructed in a number of ways: as a basic `Either` type, as an algebraic datatype, or as a composite functor. Each method has its advantages and disadvantages and are discussed in section 3.5.

A *semantic algebra* denotes a syntactic functor to a semantic space. This is the "evaluation strategy" mentioned above. Semantic algebras are combined to yield composite algebras in the same way that functors are combined into composite functors. If an algebra produces any monadic effects, then the semantic space is a

monadic computation that must be executed using the appropriate run function to yield a value.

This chapter introduces MMS by implementing an MMS framework in Haskell and constructing an interpreter for a toy language using this framework. This language is composed of three sub-languages:

- **Arith** - An arithmetic language that denotes to the Identity monad over integers.
- **Func** - A simple untyped lambda calculus that denotes to Reader monad over lambda abstractions. The environment of monad is the local context of variable bindings.
- **Imp** - An imperative language with set, get, and sequence operations that denotes to the State monad over the unit type. The state of the monad is the global list of variable bindings.

This Arith/Func/Imp language is abbreviated AFI. Figure 3.1 shows the relationship between syntactic functors, algebras, and a monadic semantic value spaces, and how the composite AFI language can be created from each independent sub-language.

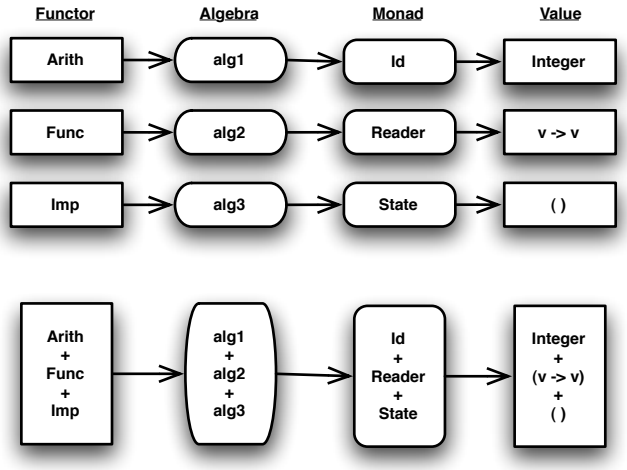

Figure 3.1. Modular Monadic Semantics

3.1 Subtyping

In MMS, complex abstract syntax, semantic algebras and value spaces can each be defined as sums of smaller, unrelated components. This is accomplished using subtyping, treating individual components as subtypes of a larger supertype.

The `SubType` class, defined in figure 3.2, provides an interface for injecting into and projecting out of a supertype. The function `inj` injects a value into the supertype, and the function `prj` projects a value out to a specific subtype. The result of `prj` is a `Maybe` because projection may fail if the argument is not of the assumed type.

The simplest supertype is the sum `Either a b`, of which `a` and `b` are subtypes. Instances of `SubType` for `Either` are provided along with the class in figure 3.2. Note that this implementation assumes that sums are always right-nested; that is, the left side may not be another sum. This literally treats the sum as a list in the type world, where `Either` is analogous to cons, `a` is the head and `b` is the tail. Not surprisingly, this implementation requires an analog of the null list. We can satisfy this requirement by following the convention that the right-most, terminating type in the sum is always `()`, and that no term is ever injected into this terminal type.

```
class SubType a b where
  inj :: a -> b
  prj :: b -> Maybe a

instance SubType a (Either a x) where
  inj = Left
  prj (Left x) = Just x
  prj _ = Nothing

instance SubType b x => SubType b (Either a x) where
  inj = Right . inj
  prj (Right x) = prj x
  prj _ = Nothing
```

Figure 3.2. SubType class

To sum characters and integers together, one would use the type

```
Either Char (Either Int ())
```

The following example injects a character into this sum:

```
x :: Either Char (Either Int ())
x = inj 'a'
```

If we project `x` to a character, we get back the original character:

```
prj x :: Maybe Char
> Just 'a'
```

However, if we project to an integer, the projection fails:

```
prj y :: Maybe Int
> Nothing
```

Applications of `Either` are easier to read if is is implemented as a right-associative infix type constructor. This type constructor will be used in the next section to create a value space.

```
infixr 5 :+:
type (:+:) = Either
```

3.2 Syntactic Functors

In MMS the abstract syntax for a language is represented as a collection of separate functors. Each sub-language in the AFI language is defined as an algebraic datatype of kind $* \rightarrow *$ in figure 3.3. There are two kinds subterms that appear in the right-hand side of each syntactic functor:

- A non-recursive term, represented as a fixed type. Examples include `Int` and `String`.
- A recursive term, whose type is always the carrier of the functor.

The carrier can be anything. When constructing terms in the language, it is the same type as the functor, or is a composite functor as described in the next section. During evaluation, the carrier is the semantic space.

Figure 3.4 defines an instance of `Functor` for each of the algebraic datatypes. Notice how these definitions differ from the functor definitions in section 2.4. Typically the purpose of `fmap f` is to push `f` into a functor and apply it at every non-recursive node. In MMS, however, `fmap f` applies the evaluation function `f` to every recursive subterm, leaves the non-recursive terms unmodified, and never

```
data Arith x = Num Int
             | Add x x

data Func x = Var String
            | Lambda String x
            | App x x

data Imp x = Get String
           | Set String x
           | Seq x x
```

Figure 3.3. Syntactic functors for AFI language

maps the function deeper into the data structure. The act of pushing the evaluation function into the subterms is handled by the catamorphism, which is defined in section 3.6.

```
instance Functor Arith where
  fmap _ (Num x) = Num x
  fmap f (Add x y) = Add (f x) (f y)

instance Functor Func where
  fmap _ (Var s) = Var s
  fmap f (Lambda s x) = Lambda s (f x)
  fmap f (App x y) = App (f x) (f y)

instance Functor Imp where
  fmap _ (Get s) = Get s
  fmap f (Set s x) = Set s (f x)
  fmap f (Seq x y) = Seq (f x) (f y)
```

Figure 3.4. Functor instances for AFI language

3.3 Fixed Points

A language's abstract syntax tree is represented as the fixed point of a functor. To motivate the use of the fixed point, consider the types of the following terms defined using the `Arith` functor:

```
t0 = Num 1
```

```
t1 = Add t0 (Num 2)
t2 = Add t1 (Num 3)
```

There is no restriction on the carrier of `t0` because it contains no recursive subterms. Thus, the type of `t0` may simply be `Arith x`. The term `t1` contains one recursive term, `t0`, so the carrier of `t1` is `Arith x` and the type of `t1` is `Arith (Arith x)`. Finally, the type of `t2` must be `Arith (Arith (Arith x))`. Because `x` can be anything, `t0` and `t1` may have the same type as `t2`. The type of any such terms is valid as long as the `Arith` type constructor appears at least as many times as the depth of the data structure.

It is impossible to use only the `Arith` type constructor to construct a type that allows arbitrarily deep arithmetic terms. To realize this arbitrarily deep recursive data structure, a fixed point of the functor is formed using the following constructed type:

```
newtype Fix f = In (f (Fix f))
out (In x) = x
```

`Fix` turns a functor and yields a nullary type. The `out` function simply strips off the constructor. The kind of `Fix` and the type of `In` are as follows:

```
Fix :: (* -> *) -> *)
In :: f (Fix f) -> Fix f
```

Rather than defining a term by finitely nesting a functor `f`, one can build terms of type `Fix f` using the constructor `In`. For example, the type of `In` simply restricts the carrier of `Arith` to be `Fix Arith`, and yields a term of type `Fix Arith`. All of the following terms are thus of type `Fix Arith`:

```
t0' = In (Num 1)
t1' = In (Add t0' (In (Num 2)))
t2' = In (Add t1' (In (Num 3)))
```

3.4 Composite Functors

Any two functors may be summed together into a composite functor. Figure 3.5 defines `FSum`, a type constructor that sums together two functors to yield another functor, and an instance of `Functor` for this composite functor. The kind of `FSum` is thus

```
(* -> *) -> (* -> *) -> * -> *
```

The instance defines that `FSum f g` is a functor if both `f` and `g` are functors. It simply pushes the `fmap` function into the correct branch.

`FSum` is defined as a newtype instead of a type synonym because type synonyms cannot be partially applied. We need to refer to the type `FSum f g`, which is of kind $* \rightarrow *$, for defining the functor instance. When applying type constructors for a type synonym, however, results must always be of kind $*$. Therefore, we use a newtype with a single constructor to define `FSum`.

```
newtype FSum f g x = S (Either (f x) (g x))

instance (Functor f, Functor g) => Functor (FSum f g) where
  fmap h (S (Left x)) = S (Left (fmap h x))
  fmap h (S (Right x)) = S (Right (fmap h x))
```

Figure 3.5. Composite functor definition

As a prefix type constructor, `FSum` can be somewhat messy. The following right-associative infix type constructor can be defined. It replaces `FSum` just as `(:+:)` replaces `Either`.

```
infixr 5 :$:
type (:$:) = FSum
```

A `SubFunctor` class handles injection and project for functors much like the `SubType` class does for nullary types, and is defined in figure 3.6. Like the `SubType` class, this class assumes that sums are right-nested and terminated by the unit functor, `FUnit`, which is simply defined as

```
data FUnit x
```

and then declared to be a functor using

```
instance Functor FUnit
```

The definition of `fmap` is left undefined because it is never intended to be called. This instance solely exists to satisfy the class constraint.

In InterpreterLib (see chapter 4), the `SubType` and `SubFunctor` classes are merged into a single, consistent interface. Furthermore, the restriction on the structure of composite functors, that they must be right-nested, is made more explicit by embedding the notions of head, tail, cons, and nil for functors into the class constraints.

```
class SubFunctor f g where
  injF :: f a -> g a
  prjF :: g a -> Maybe (f a)

instance SubFunctor f (FSum f g) where
  injF x = S (Left x)
  prjF (S (Left x)) = Just x
  prjF _ = Nothing

instance SubFunctor f g => SubFunctor f (FSum x g) where
  injF x = S (Right (injF x))
  prjF (S (Right x)) = prjF x
  prjF _ = Nothing
```

Figure 3.6. SubFunctor class

One may now combine the Arith, Func, and Imp functors from section 3.2 into a single composite functor of the type

```
type AFI = Arith :$: Func :$: Imp :$: FUnit
```

and construct any of the terms listed in figure 3.7, all of which are of type `Fix AFI`, by using the following injection function:

```
toS = In . injF


num1 = toS (Num 1)
num2 = toS (Num 2)
lam1 = toS (Lambda "x" (toS (Add (toS (Var "x")) num1)))
app1 = toS (App lam1 num2)
seq1 = toS (Seq (toS (Set "x" num1)) (toS (Add (toS (Get "x")) num2)))
```

Figure 3.7. Sample terms in AFI Language

3.5 Value Space

In interpreting a syntactic functor, a semantic algebra denotes to a monadic computation that, when executed using the appropriate run function, yields a value. This value is part of the *value space* that is defined in Haskell. This section discusses three ways of defining a value space: i) An `Either` sum; ii) an algebraic

datatype; and iii) a composite functor. All three methods use the common `SubType` interface, and thus any semantic algebra that denotes to one may also denote to the others without any changes.

The AFI language requires three different values in the value space: integers, functions, and unit. Functions in this value space must be of the general type `v → m v`, where `m` is an instance of the Reader monad, because the Reader monad will be used to capture context for function application (see section 3.6). For increased modularity, the value space should be parameterized over the monad, so that extensions to the language that add additional monad constraints do not require changes in the definition of the value space. This leads to a problem that is not solved in the literature, but is solved in InterpreterLib (see section 4.3). For now, the value space is not parameterized, and a specific monad is used:

```
type M = ReaderT [(String, Value)] (State [(String, Value)])
```

The value space for the AFI language may defined using the `Either` type:

```
newtype Exp = Exp (Value -> M Value)
type Value = Int :+: Exp :+: () :+: ()
```

where the first `()` is used for the value of statements in the Imp language, and the second one terminates the sum. Alternatively, it may be defined as an algebraic datatype:

```
data Value = N Int
           | U
           | F (Value -> M Value)
```

Finally, the value space may be defined as a composite functor. Integers and unit are *constant functors*. Any type of kind `*` can be made into a constant functor of kind `* → *` by using the following datatype that ignores the carrier:

```
newtype Const a x = Const a

instance Functor (Const a) where
  fmap _ (Const v) = Const v
```

Functions can be represented as a sort of "pseudo-functor", a datatype of kind `* → *` for which `fmap` cannot be defined. The reason the value space has been constructed as a functor is to make the injection and projection functions of the `SubFunctor` class available, not to use `fmap`. Thus, one can simply declare that `Exp` is a functor to satisfy the constraint for `SubFunctor` without actually defining `fmap`:

```
newtype Exp x = Exp (x -> M x)
```

```
instance Functor Exp
```

The definition of the value space as a composite functor is thus

```
type Value = Fix (Const Int :$: Exp :$: Const () :$: FUnit)
```

In order to inject into and project out of this this value space as we do other value spaces, the following instance of `SubType` must be defined[1]

```
instance SubFunctor f g => SubType (f (Fix g)) (Fix g) where
    inj = In . injF
    prj = prjF . out
```

Each method has its advantages and disadvantages. The `Either` sum is somewhat simpler and comes with predefined instances of `SubType`, whereas the user must define instances of `SubType` for the algebraic datatype (see figure 3.8). However, a type synonym cannot be cyclic, so one must define a separate newtype for values with recursion in their types, such as functions. This requires the user to explicitly use the `Exp` constructor in the definition of the semantic algebra. With the algebraic datatype, every element in the value space already has a constructor that allows recursive types, so there's no need to define additional newtypes. Defining the value space as a functor does not seem any more useful than using an `Either` type. However, if syntactic spaces and value spaces are defined the same way, then the user has less to learn. Furthermore, if the value space is an extension of or a subset of the syntactic space, which is the case in operational semantics and in this thesis, then is is best defined as a functor.

3.6 Semantic Algebras

An semantic algebra denotes a syntactic functor to a semantic space. It is a function mapping a functor to its carrier, so that any algebra called `phi` has the general type

```
(Functor f) => f a -> a
```

[1]The Haskell typechecker will not use this instance unless the user carefully ascribes the type of `inj` and `prj` when they are called. InterpreterLib, however, does support a working `SubType` interface for composite functors.

```
instance SubType Int Value where
    inj = N
    prj (N x) = Just x
    prj _ = Nothing

instance SubType () Value where
    inj = const U
    prj U = Just ()

instance SubType (Value -> M Value) Value where
    inj = F
    prj (F x) = Just x
    prj _ = Nothing
```

Figure 3.8. Instances of SubType for ADT value space

where the carrier `a` is the semantic space. For example, an evaluation algebra that denotes the arithmetic language to integers would have the type `Arith Int` →`Int`, whereas an algebra that performs an AST transformation on the AFI language would have the type `AFI (Fix AFI)` →`Fix AFI`.

To turn an algebra into an evaluation function that can be applied to fixed-point terms in a language, the catamorphism is used:

```
cata :: (Functor f) => (f a -> a) -> Fix f -> a
cata phi = phi . fmap (cata phi) . out
```

When applied to a fixed-point term, `cata phi` maps the semantic algebra `phi` onto each of the term's recursive subterms and then applies `phi` to the result. This means that semantic algebras are always defined as though the subterms have already been evaluated.

Typically (outside of MMS) an evaluation function applies itself to all recursive subterms, such as

```
eval (Add x y) = eval x + eval y
```

However, in the definition of a semantic algebra the subterms have already been evaluated. So, an algebra to denote the Arith language to integers would be defined as follows:

```
phi :: Arith Int -> Int
phi (Num x) = x
phi (Add x y) = x + y
```

This algebra denotes to a specific value space. A more modular algebra would use injection and projection to denote to *any* composite value space that includes integers. Algebras for each syntactic functor in the Arth/Func/Imp language are defined in figure 3.9. They use the following two functions that handle monadic injection and projection:

```
returnInj = return . inj
prjM m = m >>= return . prj
```

Note that this algebra assumes that projection always succeeds in cases where it could clearly fail. However, the only case where a projection could fail is if a term is not typed correctly. Typechecking the language before evaluating it ensures that these projections always succeed. A typechecker for this language is defined in the appendix and discussed in the next chapter.

Any two algebras that denote to the same value space can be combined into a composite functor that denotes to that value space. By writing algebras that denote to a supertype value space using injection and projection, one can easily write algebras that may always be combined with other algebras. The following function sums two algebras together, yielding a composite algebra for a composite functor:

```
sumAlg :: (f a -> a) -> (g a -> a) -> ((FSum f g) a -> a)
sumAlg phi1 phi2 f = case f of
                       (S (Left x)) -> phi1 x
                       (S (Right x)) -> phi2 x
```

Just like `FSum` is more readable as an infix type constructor, this function is more readable as an infix operator:

```
infixr 5 @+@
(@+@) = sumAlg
```

This function chooses which algebra to apply to a term based on which functor that term is a member of. It is necessary that the composite functor and composite algebra for a language are summed in the same order; that is, because of the order of the functors in the the composite functor `AFI` (section 3.4), the composite algebra for this language must be constructed as

```
evalAlg = phiArith @+@ phiFunc @+@ phiImp @+@ unitAlg
```

where unitAlg corresponds to FUnit and is `undefined` because it is never applied.

Finally, to execute terms in this language, `cata` is applied to `alg` to yield a function of type `Fix AFI` →`m Value`, where `m` is constrained to Reader and State monads.

```
phiArith (Num x) = returnInj x

phiArith (Add x y) = do
  Just xval <- prjM x
  Just yval <- prjM y
  returnInj (xval + yval :: Int)
```

```
phiFunc (Var s) = do
  env <- ask
  case lookup s env of
    (Just x) -> return x
    Nothing -> fail ("undefined␣identifier:␣" ++ s)

phiFunc (Lambda s body) = do
  env <- ask
  returnInj (Exp (\v -> local (const ((s, v):env)) body))

phiFunc (App x y) = do
  Just (Exp f) <- prjM x
  yval <- y
  f yval
```

```
phiImp (Get x) = do
  store <- get
  case lookup x store of
    Just v -> return v
    Nothing -> fail ("Undefined␣variable:␣" ++ x)

phiImp (Set x y) = do
  store <- get
  yval <- y
  put ((x, yval):store)
  returnInj ()

phiImp (Seq x y) = x >> y
```

Figure 3.9. Evaluation semantic algebras

The run functions for Reader and State execute the monadic computation, yielding a final value. The parameter to each run function is `[]`, indicating an empty initial list of bindings.

```
eval = cata evalAlg

runEval :: Fix AFI -> Value
runEval t = evalState (runReaderT (eval t) []) []
```

This `runEval` function can be used to evaluate to any of the terms listed in figure 3.7. This interpreter can easily be extended to include, say, a Boolean language consisting of true, false and if expressions. This process would consist only of the following tasks:

- Define a syntactic functor, `Boolean`, for the Boolean language.
- Define a semantic algebra, `phiBool`, for the Boolean language.
- Extend the composite syntactic space, semantic space and algebra by summing them with `Boolean`, `Const Bool` and `phiBool`, respectively. This is trivial.

No other changes are necessary. Even the definitions of the `eval` and `runEval` functions can remain the same.

Chapter 4

InterpreterLib

InterpreterLib implements MMS along with some novel and powerful extensions. Two of the more significant extensions are *algebra combinators* and *generics*.

The algebra sum presented in chapter 3, `sumAlg` or `(@+@)`, takes two algebras for *different* functors and yields a composite algebra for a composite functor. Algebra combinators combine multiple algebras for the *same* functor, allowing the user to build complex semantics from smaller, more manageable semantics. These combinators allow algebras to be sequenced, providing the result of one to another, or executed in parallel, selecting which algebra to apply at each node based on contextual information.

InterpreterLib includes a preprocessor called AlgC that generates boilerplate code. In order to use all the features of InterpreterLib, the user must define a significant amount of boilerplate code, which mostly consists of many class instances for each syntactic functor, such as the instances of `Functor` in section 3.2. Rather than define all of these instances manually, the user can write a signature for each functor comparable to the datatype declarations in figure 3.3, and AlgC processes those definitions and generates all necessary boilerplate code.

InterpreterLib fixes the problem with cyclic types introduced in section 3.5 with a solution known as HoFix. HoFix allows the user to take the fixed point of a functor but get another functor back, rather than a nullary type. HoFix code is generated automatically by a preprocessor known as MonadBuilder. This is discussed in section 4.3.

InterpreterLib is far more complex and lengthier than the MMS implementation in the previous chapter, thus Interpreter itself is not provided in this paper. It is available for download on the SLDG website [34].

The interface to InterpreterLib is relatively the same as the MMS implementation in chapter 3. This chapter discusses the interface to InterpreterLib and some of its extensions to MMS, including HoFix, AlgC, algebra combinators and generics. There are many features implemented in InterpreterLib, but only those used in this thesis are presented here. The code presented in this chapter defines and extends the example interpreter from chapter 3 using these features. The following steps are performed:

1. Begin with the AFI language, extended to include Boolean expressions and typed Lambda expressions. This shall be called the BAFI language.
2. Define a typechecker and evaluator for the language. These are defined in the appendix (sections A.1 and A.2).
3. Generically update the typechecker and evaluator to support floating-point numbers and operations.
4. Add increment and decrement operators for integers and define a generic elaborator to translate the operators to the add and subtract operations already implemented in the language.
5. Use a sequence algebra to provide the types from the typechecker to the evaluator.

The composite functor for the BAFI language is

```
type BAFI = Arith :$: Const Bool :$: Func :$: Imp :$: FUnit
```

The typechecking and evaluation algebras are provided in the appendix (sections A.1 and A.2), and the extensions to these algebras are defined in this chapter. In section 6.1, the typechecker and evaluator are extended with support for Lisp-style staged metaprogramming using the switch algebra and generics from InterpreterLib.

4.1 MMS Interface

InterpreterLib defines an algebra as an algebraic datatype, `Algebra`, such that `Algebra f a` and `f a` $\rightarrow$ `a` are isomorphic. The following functions convert between these two datatypes.

```
mkAlg :: (Functor f) => (f a -> a) -> Algebra f a
apply :: Algebra f a -> f a -> a
```

Defining an explicit type for algebras, rather than just allowing any function of type `f a` $\rightarrow$ `a` to qualify, has allowed some parts of InterpreterLib to be implemented more easily. Fortunately, this has not been burdensome to the user, since `mkAlg` and `apply` are quite easy to use.

InterpreterLib provides the `returnInj` function defined above, and a `checkTypeM` function that is similar to `prjM`, except that it strips of the `Just` constructor to yield a value of type `a` rather than `Maybe a`. Projecting to an invalid type, which would otherwise return `Nothing`, instead causes a run-time exception.

```
returnInj :: (Monad m, SubType a v) => a -> m v
checkTypeM :: (Monad m, SubType a v) => m v -> m a
```

InterpreterLib includes useful functions for lifting functions into a monadic supertype. The two most commonly used ones are `liftMSub` and `liftM2Sub`. The first argument to these functions, the function being lifted into the supertype, must have a specific monomorphic type so that Haskell knows exactly which subtype to project to, apply the function to, and re-inject the result from.

```
liftMSub :: (Monad m, SubType b v, SubType a v) =>
            (a -> b) -> m v -> m v

liftM2Sub :: (Monad m, SubType c v, SubType b v, SubType a v) =>
             (a -> b -> c) -> m v -> m v -> m v
```

4.2 Language Syntax and AlgC

AlgC is a preprocessor for InterpreterLib that generates boilerplate from functor signatures. Functor signatures for this chapter's example language are defined in figure 4.2. From each of these signatures AlgC generates the following Haskell code:

- A datatype for the functor like those in section 3.2.
- Instances of Functor, Traversable, Foldable, Applicative, AmbiFunctor, FunctorContext, and FunctorZip.
- A "mk" function for each constructor that can be used to construct terms and inject them into a composite functor.

```
num1 = mkNum 1
num2 = mkNum 2
lam1 = mkLambda "x" TyInt (mkAdd (mkVar "x") num2)
app1 = mkApp lam1 num1
seq1 = mkSeq (mkSet "x" num1) (mkAdd (mkGet "x") num2)

test run = map run [num1, num2, lam1, app1, seq1]

fact = mkSet "fact"
   (mkLambda "n" TyInt
   (mkIf (mkIsZero (mkVar "n")) (mkNum 1)
   (mkMult (mkVar "n") (mkApp (mkGet "fact") (mkDec (mkVar "n"))))))

 pow = mkSet "pow"
   (mkLambda "x" TyFloat
   (mkLambda "n" TyInt
   (mkIf (mkIsZero (mkVar "n")) (mkNum 1)
   (mkMult (mkVar "x")
   (mkApp (mkApp (mkGet "pow") (mkVar "x")) (mkDec (mkVar "n")))))))
```

Figure 4.1. Sample terms

The example terms from figure 3.7, along with the definitions of a factorial function and a power function, are constructed in figure 4.1 using the "mk" functions generated by AlgC.

4.3 HoFix and MonadBuilder

Interpreters built using Modular Monadic Semantics often use the Reader monad to manage an environment of name-to-value bindings for variable declarations and lambda expressions, and the value space includes Haskell functions of the form `v → m v`. For modularity, the value space is parameterized over this monad, `m`. As discussed in section 3.5, this leads to an impossibly typed run function, such as:

```
run :: Lang -> Value (Reader (Value (Reader (Value ...
run t = runReader (cata evalAlg t) []
```

A module in InterpreterLib, HoFix, solves this problem. It provides a type combinators, `HoFix`, that is similar to `Fix` in that it takes the fixed point of two functors,

```
signature Arith = Num Int
                | Float Float
                | Arith * *
                | Sub * *
                | Mult * *

signature Boolean = Tru
                  | Fls
                  | IsZero *
                  | If * * *

signature Func ty = Var String
                  | Lambda String ty *
                  | App * *

signature Imp = Get String
              | Set String *
              | Seq * *

signature Ops = Dec *
              | Inc *
```

Figure 4.2. Functor signatures for AlgC

particularly two monads. Unlike `Fix`, the result of `HoFix` is another functor.

```
newtype HoFix hof a = HoFix { unHoFix :: hof (HoFix hof) a }
```

The following `FunctorIso` class allows the user to define an isomorphism between two monads:

```
class FunctorIso f g | f -> g, g -> f where
  fOut :: f a -> g a
  fInn :: g a -> f a
```

If such an isomorphism exists between `HoFix hof` and a monad `m`, then `HoFix hof` is an equivalent monad. This is captured by the following instance, whose definition is omitted. Similar instances for each specific monad are also defined in InterpreterLib.

```
instance ( Monad m, FunctorIso (HoFix hof) m) => Monad (HoFix hof)
```

Defining a HoFix monad is tedious, so a preprocessor called MonadBuilder generates the Haskell code automatically from a simple definition. The following is parsed by MonadBuilder to yield the HoFix monad used in this chapter.

```
monad EvalMonad = Reader [(String, Value EvalMonad)]
                . State [(String, Value EvalMonad)]
```

From this definition, MonadBuilder generates a monad that includes the above Reader and State instances and defines an isomorphism between this monad and the HoFix of the monad. The user only needs to interact with the generated `runEvalMoand` function:

```
runEvalMonad :: EvalMonad x -> [(String, Value EvalMonad)]
                            -> [(String, Value EvalMonad)]
                            -> (x, [(String, Value EvalMonad)])
runEvalMonad x initR initS = runIdentity (runStateT
                               (runReaderT (fOut x) initR) initS)
```

4.4 Generics

Generic programming is a class of techniques for concise definitions of functions that remain complete even when data types are changed (cite bananas). Like Modular Monadic Semantics, its benefits include highly modular, reusable code. Generic programming can allow the user to define complex traversal functions by only specifying two things: i) a generic default behavior that is applied everywhere;

and ii) an exception to the generic case, in which a specific behavior is applied only when a specific type is encountered. One approach to this is "Scrap your Boilerplate" approach [16,17], which uses a dynamic type-safe cast to allow generic algorithms to be written over recursive datatypes.

Lammel et al. [18] provide another approach to generic programming that represents semantic algebras as Haskell records with a field corresponding to each constructor. Updating the semantics for a single constructor is a simple record update. This allows an algebra to provide a default behavior that can be overridden with different behaviors for specific cases.

InterpreterLib generalizes this approach to functors, providing an implementation of updateable algebras that is built on top of the MMS framework. It relies heavily on type-level programming, using the Haskell type class system to statically resolve the appropriate instantiation of a generic function. Using the type class system this way has been demonstrated with statically-typed heterogeneous lists [14] and pseudo-dependent types [21].

The "Scrap your Boilerplate" approach can be used to update a semantic algebra. However, any composite algebra that an updated algebra is part of must be reconstructed using the algebra sum operator, `@+@`. This operator requires that the user construct a composite algebra in the same order as the composite functor that it denotes.

The generic algebra update provided by InterpreterLib does not make this restriction. For example, the composite typechecker algebra for the BAFI language (defined in section A.1) is constructed as follows:

```
tyAlg = mkAlg tyArith @+@ mkAlg tyBool @+@
        mkAlg tyFunc @+@ mkAlg tyImp @+@ funitAlg
```

The algebras must be summed exactly in this order because the composite functor that this algebra denotes is defined as

```
type BAFI = Arith :$: Const Bool :$: Func :$: Imp :$: FUnit
```

However, with a generic algebra update, the user can specify a default behavior (undefined) and update that default behavior with each of the four algebras in any order:

```
tyAlg = upd tyFunc (upd tyArith (upd tyImp (upd tyBool undefined)))
    where upd = updAlg . const . mkAlg
```

InterpreterLib also provides a generic transformer algebra that is useful for defining AST transformations. Its default behavior is an identity algebra that can be

overridden for specific functors.

The update algebra and transformer algebra are discussed in the next two sections. More information on generics in InterpreterLib can be found in [41].

4.4.1 Update

Any algebra may be treated as a generic algebra that defines the default behavior for a functor. Its behavior may be updated, yielding a new algebra that applies a new behavior to specific constructors, and the old behavior everywhere else. Furthermore, a single algebra in a composite algebra can be generically updated with new behavior, redefining its behavior for its corresponding functor in a composite functor. Monad effects are correctly sequenced by this update.

InterpreterLib provides the following function for updating an algebra:

```
updAlg :: (Algebra g a -> Algebra g a) ->
          Algebra sum a -> Algebra sum a
```

where `sum` must be a composite functor that includes `g`. The update is type-driven; that is, `updAlg f alg` determines which algebra in `alg` to update based on the type of `f`. It then provides the original algebra to `f`, so `f` can apply this original algebra to any constructors whose behavior it does not want to update. In this case, `alg` is the generic default algebra and `f` is the function to override the behavior for the functor `g`. `updAlg` provides the original algebra for the functor `g` to the function `f` so that `f` can choose whether to apply the original algebra or a new behavior depending on, for example, which constructor it encounters.

We can now add support for floating-point numbers to the typechecker by replacing the `tyArith` algebra with `tyArith'`, defined in figure 4.3. This algebra is not just an update of `tyArith`, it is a complete replacement. Thus, when we use `updAlg` to update `tyAlg`, we can ignore the original algebra using `const`. The updated algebra, `tyAlg'`, has the same type as `tyAlg`, so its run function is identical.

The evaluator is updated slightly differently, as shown in figure 4.4. For arithmetic operations of type `TyInt`, the original algebra `phiArith` is used. For arithmetic operations of type `TyFloat`, the algebra `phiArithFloat` is used. `evalAlg` is then updated so that the type of every arithmetic term determines which arithmetic algebra is applied. Notice that this algebra does not need to know the name of the original arithmetic algebra, `phiArith`, which is simply referred to as `oldAlg`. In section 4.5, I show how to provide the results of the typechecker to this algebra

```
tyArith' (Num _) = return TyInt
tyArith' (Float _) = return TyFloat
tyArith' (Add x y) = liftM2 arithHelper' x y
tyArith' (Sub x y) = liftM2 arithHelper' x y
tyArith' (Mult x y) = liftM2 arithHelper' x y

arithHelper' TyInt TyInt = TyInt
arithHelper' TyFloat TyFloat = TyFloat
arithHelper' _ _ = error "Bad␣arguments␣to␣arithmetic␣operator"

tyAlg' = updAlg (const (mkAlg tyArith')) tyAlg
```

Figure 4.3. Typechecker with float support

at each node in the AST.

```
type FloatOp = Float -> Float -> Float

phiArithFloat (Float x) = returnInj x
phiArithFloat (Add x y) = liftM2Sub ((+) :: FloatOp) x y
phiArithFloat (Sub x y) = liftM2Sub ((-) :: FloatOp) x y
phiArithFloat (Mult x y) = liftM2Sub ((*) :: FloatOp) x y

evalAlg' ty = updAlg (select ty) evalAlg where
    select TyInt oldAlg = oldAlg
    select TyFloat _ = mkAlg phiArithFloat
```

Figure 4.4. Type-directed evaluator with float support

4.4.2 Transform

The type of `updAlg` is restricted so that the resulting composite algebra must have the same type as the original algebra. InterpreterLib defines another generic algebra updater, `transAlg`, that does not have this restriction. Its default behavior is an identity algebra, and it takes in a list of algebras that replace the default behavior for the specific functors that those algebras apply to. The type of `transAlg` and its non-monadic counterpart, `transPureAlg`, are as follows:

```
transAlg :: (Monad m) => AlgebraList fs (m (Fix sum'))
                         -> Algebra sum (m (Fix sum'))
```

```
transPureAlg :: (Traversable fs) => AlgebraList fs (Fix sum')
                                    -> Algebra sum (Fix sum')
```

A "list" of algebras, `AlgebraList`, is constructed with `(@:@)` and `emptyAlgList`, which are analogous to `(:)` and `[]` for lists. The type of `(@:@)` is as follows:

```
(@:@) :: (Functor fs, Functor f) =>
         Algebra f a -> AlgebraList fs a -> AlgebraList (f :$: fs) a
```

It takes in an algebra that denotes the functor `f` and a list of algebras that denote a composite functor `fs`, yielding an algebra list for `f :$: fs`.

Rather than write a denotation for the increment and decrement operators of the example language, one can elaborate `Inc` to `Add (Num 1)` and `Dec` to `Add (Sub 1)`. Note that these operators are thus only defined for integers, not floating-point numbers. The entire elaborator is defined generically in figure 4.5. The `phiElab` function rewrites the increment and decrement operators. The `elabAlg` algebra applies an identity algebra everywhere except when it encounters the `Ops` functor, to which it applies `phiElab`. This elaborator uses is not monadic, so it uses `transPureAlg` and does not require any monadic run function. Note that the types of `elabAlg` and `elab` explicitly eliminate the `Ops` functor from the language.

```
phiElab :: (SubFunctor Arith sum) => Ops (Fix sum) -> Fix sum
phiElab (Inc x) = mkAdd (mkNum 1) x
phiElab (Dec x) = mkSub (mkNum 1) x

elabAlg :: Algebra (Ops :$: LangS) Lang
elabAlg = transPureAlg (mkAlg phiElab @:@ emptyAlgList) where

elab :: Fix (Ops :$: LangS) -> Lang
elab = cata elabAlg
```

Figure 4.5. Elaborator for Inc and Dec operators

4.5 Algebra Combinators

Algebra combinators allow the user to define complex algebras as a composition of simpler algebras. Unlike the algebra sum operator `(@+@)` that simply combines algebras for separate functors into a composite algebra for a composite functor, algebra combinators combine multiple algebras for the same functor, allowing

rich and complex semantics to be defined. Algebra combinators are particularly useful for capturing dependencies between analyses. The *sequence* combinator chains two algebras together, providing the results of one to the other. The *switch* combinator selects between modes of interpretation based on context.

More information on algebra combinators in InterpreterLib can be found in [13, 40, 41].

4.5.1 Sequence

The sequence algebra combinator takes in two algebras, applies the first algebra entirely, and then passes the result of the first algebra at every node in the AST to the second algebra. Specifically, for every node in the AST, the result of that node and the result of all subterms are made available to the second algebra. The second algebra is then parameterized over these two results, and we say that is is *indexed* by them. To sequence `alg1` followed by `alg2`, the general types of these two algebras must be

```
alg1 :: Algebra f (m a)
alg2 :: a -> f a -> Algebra f b
```

If `alg1` is one of the typechecking algebras from above, and the sequence is applied to the term

```
mkIf mkTru (mkFloat 1) (mkFloat 2)
```

then `alg2` will see `TyFloat` and `(If TyBool TyFloat TyFloat)` for the `a` and `f a` terms, respectively. Note that both algebras operate on the same original AST; the second algebra simply has the results of the first algebra available to it while it traverses the original AST. Creating the sequence algebra is as simple as

```
alg1 `seqMAlg` alg2
```

where `seqMAlg` has the following type:

```
seqMAlg :: Algebra f (m a) -> (a -> f a -> Algebra f b)
           -> Algebra f (m b)
```

Finally, InterpreterLib defines a transformer for the sequence algebra called `runSeqT`. It allows the two algebras to use the same monads with different constraints, so that a sequence algebra is never hindered by algebras with conflicting monad constraints. To execute a sequence algebra, the following steps are performed:

1. Apply `cata` to the sequence algebra

2. Apply `runSeqT`

3. Apply each of the monad run functions for `alg1`

4. Apply each of the monad run functions for `alg2`

The evaluator in the previous section depends on the type of arithmetic terms to determine whether to evaluate the subterms as integers or floats. The resulting types from the typechecker algebra are provided to this evaluator using the sequence algebra defined in figure 4.6. The evaluator only needs the type of each term, and not the subterms, so that parameter is ignored in the sequence. Finally, `cata` and each of the run functions are applied, yielding a final value.

```
seqEvalAlg = tyAlg' `seqMAlg` (\ty _ -> evalAlg' ty)

runEval' :: Lang -> Value EvalMonad
runEval' t = fst t5 where
    t1 = cata seqEvalAlg t
    t2 = runSeqT t1
    t3 = runReaderT t2 []
    t4 = evalState t3 []
    t5 = runEvalMonad t4 [] []
```

Figure 4.6. Typechecker sequenced with type-directed evaluator

There is one significant restriction on the sequence combinator. The first algebra must perform a *static* analysis; that is, it cannot change the structure of the AST, either by reducing or expanding it. Because the second algebra must see the results of the first algebra for every node in the *original* AST, the structure of the AST must be preserved. Typechecking is a static analysis, but evaluation reduces the AST and could not be used as the first algebra in a sequence.

4.5.2 Switch

```
switchAlg :: (MonadSwitch r m, Functor f) =>
             (r -> Algebra f (m a)) -> Algebra f (m a)

runSwitchT :: SwitchAT sw m a -> sw -> m a
```

Chapter 5

Multi-Staging and Reflection

Multi-stage programming splits execution of a program into one or more stages, the first being static compile-time stages and the last being a runtime stage. Code generation and execution, which are normally only performed at runtime, can be performed during compilation. This allows for optimization and specialization of generic algorithms.

Staging can be performed automatically by the compiler. Nielson and Nielson show how to perform binding-time analysis (BTA) automatically [27]. This analysis determines which variables are known statically and which are known dynamically, and can then be used to build a partial evaluator [2, 12, 22, 31] by directing the compiler to evaluate statically known expressions at compile time (first stage) and dynamically known expressions at run time (second stage).

Staging can also be implemented by adding explicit staging annotations to a language, providing type-safe metaprogramming [7, 26, 36–38]. Rather than defining the stages automatically by, for example, a binding-time analysis, these annotations allow the user to choose specifically what gets evaluated at each stage. Additionally, languages can be reflective by exposing the internal AST to the programmer as a data structure in the language, allowing the user to fully manipulate the AST. This provides an additional benefit, generative programming, by allowing the user to define functions that manipulate and generate code.

The languages presented in this chapter implement staged programming and/or reflection using *quote* and *unquote* constructs. Staged reflection is made available via the following features, not all of which are available in each language:

- An abstract syntax tree for representing syntactic terms, declared in the language but implemented internally, including internally defined manipulation

functions. The AST can be fully manipulated by the user.

- Quasiquote notation, staging annotations that identify code as either meta code (an AST structure) or object code like any other code in the language. A *quote* (also known as *brackets*) causes its argument to be parsed but not evaluated, turning it into meta code. An *unquote*, *splice*, or *escape* may appear within a *quote* construct and causes its argument to be treated as object code.

- A *top-level* staging construct that bridges the gap between stages by converting meta code into object code in the language.

- One or more interpretation constructs to evaluate a syntactic meta term at compile time. Depending on the language, this may be the same as the top-level staging construct or the unquote construct, and its result may either be meta code or object code.

This thesis uses the phrases *meta code* or *AST structure* to refer to the abstract syntax representation of code that has been constructed using quotation notation or AST constructors, and *object code* to refer to ordinary code in the language or code that has been constructed via an unquote or top-level staging construct.

With the exception of Rosetta, the unquote and top-level staging constructs cause compile-time evaluation of their arguments. In Rosetta, an *unquote*, which is also the top-level staging construct, simply identifies its argument as object code so that it is not expanded to an AST structure.

With the exception of Scheme, each of these languages resolves all reflection at compile-time and enforces static scoping and static typing. Rosetta actually allows undefined identifiers inside quoted expressions, a kind of dynamic scoping, as discussed in section 5.4.

It should be noted that even in a statically typed reflective language, it is possible to construct the abstract syntax for a type-unsafe term. However, this abstract syntax can only be interpreted at compile time, not run time, and thus any type errors are still found statically. Also, it is possible to generate the abstract syntax for a variable that is not in scope, as discussed in section 5.4.

Remembering the syntax and semantics of each of these reflective languages can be confusing. The following table lists the terminology or function used for each of the aforementioned constructs for each reflective language discussed in this chapter:

Language	Quote	Unquote	Top-level Staging	Interpretation
Scheme	quasiquote	unquote	eval	eval
Template Haskell	brackets	splice	splice	lift, splice
MetaML	brackets	escape	run	lift, run
Rosetta	quote	unquote	unquote	value, typeof

Template Haskell and MetaML include a `lift` function that evaluates an expression at compile-time and converts the result to an AST structure.

Note that Scheme is not really a multi-stage language, as all reflection is performed at runtime. Also, MetaML is only staged, not reflective, because it does not include AST construction and manipulation functions.

The syntax for the quote and unquote constructs of each language are as follows:

- **Scheme**
  - Quasiquote is `‘(...)` or `(quasiquote ...)`
  - An unquote is `,(...)` or `(unquote ...)`
  - Eval is `(eval ...)`
- **Template Haskell**
  - Brackets are `[| ... |]`
  - Splice is `$(...)`
- **MetaML**
  - Brackets are `< ... >`
  - Escape is `~...`
  - Run is `run ...`
- **Rosetta**
  - Quote is `<< ... >>`
  - Unquote is ‘ ... ‘
  - Value and typeof are `value(...)` and `typeof(...)`.

MetaML is not discussed in great detail, though it deserves mentioning because it is multi-staged, performs type checking all at once instead of in stages, and does not include any reflective capabilities to manipulate abstract syntax.

There is a distinction between *multi-level* and *multi-stage* programming in the literature [36] that I do not make in this paper, but should still mention briefly. Both BTA-driven partial evaluation and the extensions to the BAFI language in section 6.1 are examples of multi-level programming. They allow evaluation to be turned on and off, but include no top-level staging construct, such as Scheme's `eval`, Haskell's top-level splice, or Rosetta's top-level unquote. A multi-level language is multi-stage if it includes such a construct.

5.1 Scheme

The discussion of reflective meta-programming begins with Scheme because it is the simplest reflective language. Everything in Scheme is either a list or an atom. Code in Scheme is represented as a list in which the first element is a function and all other elements are the arguments to that function. Thus, Scheme evaluates as follows:

```
(+ 1 2 3)        =>   6
```

Scheme supports run-time reflection. Evaluation can be turned off using a quasiquote (‘), effectively turning concrete syntax into abstract syntax, a list representing the original code. In Scheme there just happens to be no difference between concrete and abstract syntax. Evaluation can be turned back on using an unquote (`,`). Finally, an `eval` function takes in a list that represents abstract syntax and interprets it as code, executing it to yield a value. So, Scheme evaluates the following as shown:

```
‘(+ 1 2 3)       =>   (+ 1 2 3)
‘(+ 1 ,(+ 2 3))  =>   (+ 1 5)
(eval ‘(+ 2 3))  =>   5
```

Scheme code may operate on quasiquoted code like any other list. For example, the following function takes in a list and replaces the first element with `*`.

```
(define (f x) (cons ‘* (cdr x)))
```

The function `cdr` is the same as `tail` in Haskell, and `cons` is the same as `(:)`. One can then pass a quasiquoted list that represents the syntax for `(+ 2 3)`, and the function will yield the list `(* 2 3)`.

```
(f ‘(+ 2 3))          =>   (* 2 3)
(eval (f ‘(+ 2 3)))   =>   6
```

The `eval` function actually takes in a second argument, the environment in which to perform the evaluation. Possible environments include the current environment, or the environment in which any particular identifier was defined. This allows for some interesting dynamic scoping.

Quasiquote, unquote, and eval are sufficient for implementing reflection in Scheme because abstract syntax is represented as lists, a data structure that already exists in the language. Thus, functions such as `car` (`head` in Haskell), `cdr`, and `cons` can be used to generate, manipulate, and rewrite abstract syntax.

Reflection in Scheme is a macro system. The symbols ` and , are just shorthand for the macro functions `quasiquote` and `unquote` that do nothing more than turn evaluation on and off.

Scheme allows for nested quasiquotes, in which case evaluation is not turned back on until all quasiquotes are escaped. For example, the following term can be evaluated repeatedly as follows:

```
(quasiquote (quasiquote (+ 1 2 ,(* 3 4 ,(+ 2 3)))))
=> (quasiquote (+ 1 2 (unquote (* 3 4 5))))
=> (+ 1 2 60)
=> 63
```

In the staged reflective languages, this is known as *multi-stage* reflection. In a 2-stage reflective system, the abstract syntax representation of the language is a first-class value in the language and evaluating an AST structure always yields object code. In a 3-stage system, the representation of the representation is a first-class value, so evaluating an AST structure may yield another AST structure. Multi-stage systems allow an arbitrary number of stages and nestings of abstract syntax representations.

5.2 Template Haskell

Template Haskell [32] is an extension that adds static 2-staged reflection to Haskell. Haskell's abstract syntax is represented in Template Haskell as a Haskell algebraic datatype, directly available to the programmer. Brackets `[| ... |]` direct Haskell to treat its arguments as syntax in the reflective system rather than code to be executed. A splice `$(...)` executes its argument at compile-time and inserts the result back into the program. The argument to a splice must be meta code. Splices can appear at the top level (i.e. not within a bracket), in which case they are analogous to Scheme's `eval` function. Template Haskell also includes a convenient function, `lift`, that evaluates object code and lifts the result into the

syntactic meta world. So, `lift (1 + 2)` is the same as `[| 3 |]`. This is in contrast to a splice, which evaluates an AST construct to a semantic value in the object world of Haskell. Note that a *lift* function is not necessary in Scheme because the syntaxes for representing code and AST structures are the same.

Expressions are not the only abstract syntax that can be generated by brackets. Template Haskell also supports quoting patterns, declarations and types via `[p| ... |]`, `[d| ... |]` and `[t| ... |]`, respectively. The extra character tells Haskell how to *parse* the contents of the brackets.

Quasiquote notation provides a convenient shorthand for generating syntactic terms by relying on the parser and reflective system to convert concrete syntax into the algebraic datatype representation of Haskell code in Template Haskell. For complicated code generation and manipulation algorithms, the quasiquote notation is not enough, so Template Haskell's algebraic datatype representation of Haskell code is made available to the user, along with functions for generating abstract syntax.

```
sel :: Int -> Int -> ExpQ
sel i n = [| \x -> $(if i > n then error "Invalid␣tuple␣index"
                      else (caseE [| x |] [alt])) |]
    where alt = match pat (normalB rhs) []
          pat = tupP [varP (mkName ("a" ++ show i)) | i <- [1..n] ]
          rhs = varE (mkName ("a" ++ (show i)))
```

This function generates a lambda abstraction whose body is a case expression with a single pattern. It matches the argument of the lambda against `pat`, a tuple pattern containing `n` distinct variables patterns, `a1` through `an`. The right-hand side of the match, `rhs`, is `ai`. For example, `(sel 3 5)` generates abstract syntax for

```
\x_0 -> case x_0 of
          (a1, a2, a3, a4, a5) -> a3
```

Note that if `i` is more than `n`, it generates a *compile-time* exception, and that without the check for this condition, `sel 5 3` would generate

```
\x_0 -> case x_0 of
          (a1, a2, a3) -> a5
```

which can behave very strangely. If `a5` is in scope at the time of splicing `sel 5 3`, then its binding will be used. Otherwise, the typechecker will complain that `a5` is not in scope.

In the above example, Template Haskell generates a new name, `x_0`, rather than

using the original name, `x`. Reflective metaprogramming requires fresh name generation to avoid name capture problems, as discussed in section 5.4. Generating fresh variable names is a stateful computation, and thus requires a monad to capture this side effect. Template Haskell provides a *Quotation Monad*, `Q`, that encapsulates fresh name generation, as well as program reification and error reporting. Splicing escapes the Quotation monad by executing the stateful effects to yield a value in Haskell.

Some of the functions used above in the definition of `sel` are typed as follows:

```
varE :: Name -> Q Exp
caseE :: Q Exp -> [Q Match] -> Q Exp
tupP :: [Q Pat] -> Q Pat
```

Each of the above functions lifts a corresponding constructor into the quotation monad. Some of these constructors, from the algebraic datatypes that represent abstract syntax in Template Haskell, are

```
VarE :: Name -> Exp
CaseE :: Exp -> [Match] -> Exp
TupP :: [Pat] -> Pat
```

The quotation monad also captures side effects from program reification. The user may query an identifier at compile-time. The `reify` function, for example, says whether an identifier is a class, constructor, variable, etc., and gives other useful information for that identifier.

```
reify :: Name -> Q Info

data Info = ClassI Dec
          | ClassOpI Name Type Name Fixity
          | TyConI Dec
          | PrimTyConI Name Int Bool
          | DataConI Name Type Name Fixity
          | VarI Name Type (Maybe Dec) Fixity
          | TyVarI Name Type
```

Note that the reify function may fail if its argument is a name that is not in scope. Failure and recovery are also handled by the quotation monad.

5.3 Rosetta

Rosetta supports staged reflection through abstract syntax structures, template expressions, and interpretation functions [1]. Template expressions, also known

as quoted expressions, are constructed with quotes (`<< ... >>`) and escaped with unquotes (`‘...‘`). The abstract syntax for template expressions is declared in a Rosetta library along with constructors, recognizers, and observers for each AST structure, many of which are implemented internally. Just like in Template Haskell, AST structures can be fully manipulated using the supported reflective constructs.

In Template Haskell, abstract syntax is represented as algebraic datatypes that can be pattern matched. Rosetta also represents its reflective abstract syntax using algebraic datatypes; instead of pattern matching, it uses recognizers and observers. Recognizes are predicate functions that allow the user to determine what type of syntax a structure is, and observers are field selectors to extract specific arguments from a constructed datatype. For example, the algebraic datatype for the abstract syntax of an if expression is defined as

```
if_expression :: subtype ( primary ) is
data
  make_if_expression (
    if_condition :: expression;
    true_alternative :: expression;
    false_alternative :: expression ) ::
  is_if_expression
end data;
```

where `make_if_expression` is a constructor taking in three arguments, `is_if_expression` is the recognizer, and `if_condition`, `true_alternative` and `false_alternative` are the observers of each of the constructor's three arguments.

Quotes cause their argument to expand to an AST structure. Quotes are tagged with an identifier that indicates how to parse their argument. For example, `<< expression e >>` causes `e` to be parsed as an expression, whereas `<< type t >>` causes `t` to be parsed as a type. This serves the same purpose as the tag in Template Haskell's bracketed expressions. An unquote escapes a quote and is most useful for executing constructors, recognizers, and observers at compile-time.

Rosetta is not a programming language and is not meant to be executable. Reflection is meant to provide generative and generic programming to Rosetta, not compile-time optimization. Therefore, unquotes do not turn evaluation on, as in the other languages, except for the reflective functions (constructors, recognizers, and observers). Rosetta does, however, provide a `value` function that performs partial evaluation at compile time. It takes in an AST structure and evaluates it as much as possible. The result is an AST structure in a normal form, since it cannot be evaluated further. Variables that are not in scope at the time of performing partial evaluation are left unevaluated, allowing them to take on a

dynamic scope when they are later unquoted. The following examples illustrates partial evaluation in Rosetta.

```
x :: integer is 1;
y :: expression is value(<< expression x + 2 + a >>);
z :: let x :: integer be 1000; a :: integer be 5 in ‘y‘ end let;
```

When `<< x + 2 + a>>` is constructed, `x` is in scope and `a` is not, therefore `value(<< x+ 2 + a >>)` reduces to `<< 3 + a >>`. This is then unquoted in a context in which `a` is in scope and bound to `5`, thus `z` will be bound to `3 + 5`, or `8`. The binding of `x` to `1000` is ignored because `x` was already statically bound to `1`. Note that the resulting binding for `z` is the same if `value` is applied to `y` instead of in the right-hand side of `y`'s declaration, as in the following example:

```
x :: integer is 1;
y :: expression is << expression x + 2 + a >>;
z :: let x :: integer be 1000; a :: integer be 5 in ‘value(y)‘ end let;
```

Although `<< x + 2 + a >>` is not evaluated when `y` is bound to it, `x` is still statically bound to `1`. Scoping in Rosetta is discussed again in section 5.4.

Rosetta also provides a `typeof` function that reduces a template expression to a template type. Thus, `<< expression 1 + 2 >>` reduces to `<< type integer >>`. Recall that types are values in Rosetta, so they can be represented in the abstract syntax just like any other term.

The `value` preserves types; that is, the following holds for any term `x`:

```
typeof(value(x)) == typeof(x)
```

The `typeof` function can certainly fail at compile-time for the same reasons that any static typechecker can fail. Variables may be out of scope, as allowed by the `value` function, or the expression may be badly typed, such as the expression `<< expression 1 + 'a' >>`. Typing template expressions is discussed in section 5.5.

The following example uses reflection in Rosetta to define a function that assembles a complete facet declaration from a facet interface and a facet body. Facet interfaces provide the signature, exports, and declarations, while the body provides declarations and terms. If a body and an interface have the same name (label), then they can be combined into a complete facet. Because declarations appear in both the interface and the body, the set of declarations from each are appended to create the declarations for the complete facet. This `combine_elements`

```
combine_elements(fi :: facet_interface_declaration;
                 fb :: facet_body_declaration)
              :: complete_facet_declaration is
   if facet_label(fi) = facet_label(fb)
      then make_complete_facet_declaration(
         facet_label(fi), facet_signature(fi),
         exports(fi), declarations(fi) + declarations(fb),
         terms(fb))
       else bottom
    end if;
```

```
combine_elements(fi :: facet_interface_declaration;
                 fb :: facet_body_declaration)
              :: complete_facet_declaration is
   if facet_label(fi) = facet_label(fb)
      then << complete_facet_declaration
         facet ‘facet_label(fi)‘ ‘facet_signature(fi)‘ is
            ‘exports(fi)‘
            ‘declarations(fi) + declarations(fb)‘
         begin
            ‘terms(fb)‘
         end facet ‘facet_label(fi)‘ >>
       else bottom
   end if;
```

Figure 5.1. Reflective function for constructing a complete facet declaration from an interface and body

function can be defined using only template functions or by using a mix of quotes, unquotes, and template functions. Both definitions are provided in figure 5.1.

Rosetta also includes two useful functions, `string` and `parse`, that convert between string and algebraic datatype representations of an AST structure. So, `string(<< expression 1 + 2 >>)` is `"1␣+␣2"` and `parse("1␣+␣"2)` is the same as `<< expression 1 + 2 >>`. The parse function can certainly fail at compile-time if it encounters a syntax error.

5.4 Scoping and Name Generation

Reflection causes issues with scoping and name closures. Because quoting a term effectively turns evaluation off, variables in that term might not be bound when the term is created. Thus, the value of a variable in an AST construct may depend on the context in which that abstract syntax is unquoted, spliced, or run. Furthermore, splicing/running/unquoting can cause names to overlap. Examples in this section are illustrated using Template Haskell's terminology and quasiquote notation, but the issues raised by these examples apply to MetaML and Rosetta as well.

In MetaML and Template Haskell, the typechecker maintains static scoping by asserting that `x` is in scope when `[| x |]` is constructed, and that when this term is spliced, it will be evaluated in the same scope in which it was declared. However, the user can cause dynamic scoping by explicitly building abstract syntax using `varE (mkName "x")`. In this case, the value of `x` is not known until the term is spliced. Thus, the context in which this term is spliced determines how it evaluates. To avoid this, the user should generate fresh names `newName` instead of `mkName`.

Rosetta does not require identifiers to be in scope when they are quoted. If a variable inside a quote is in scope, it is replaced with its value. Otherwise, it is left alone. For example, consider the following declarations:

```
x :: integer is 1+2;
z :: expression is << expression x + 3 + a >>;
```

In this case, `x` is in scope, but `a` is not, when then template expression for `z` is constructed. Thus, `x` is statically scoped, and `a` is dynamically scoped. Note that quoting an undefined identifier, such as `a` in the above example, is not allowed in Template Haskell and MetaML. We can unquote `z` in a local context using a let expression:

```
let x be 10; a be 20 in ‘z‘ end let;
```

which elaborates to

```
let x be 10; a be 20 in 1 + 2 + 3 + a
```

where `x` has been replaced by its static binding, `1 + 2`. Note that there is an error if `a` is not in scope when `z` is unquoted. This will be caught by a second typechecking pass after elaboration.

In the operational semantics for a lambda calculus, variables introduced by lambda abstractions must be renamed to avoid name conflicts. In denotational semantics, this problem is avoided by denoting to a name-capturing function in the language in which the interpreter is written. Reification of a function back to abstract syntax, however, requires variable renaming in both operational and denotational semantics. The ability to capture closure in the denotation of a lambda abstraction is lost when it is lifted back into the syntactic domain of the language being interpreted.

The aforementioned reflective languages must rename identifiers with freshly generated names to avoid overwriting the meaning of certain variables. Whether for operational semantics or denotational semantics, reification of a function back to abstract syntax must involve variable renaming. For example, consider a function that generates lambda abstractions of the form `[| x→ x + ...|]` and folds them with an initial value to generate a function that takes in any number of integers and adds them together:

```
genAdd :: Int -> ExpQ -> ExpQ
genAdd 0 y = y
genAdd n y = [| \x -> $(genAdd (n-1) [| $y + x |]) |]
```

The parameter, `x`, is different for every lambda abstraction, and must be renamed with a fresh name for each one. Thus, `genAdd 5 [| 0 |]` generates

```
\x_0 -> \x_1 -> \x_2 -> \x_3 -> (((0 + x_0) + x_1) + x_2) + x_3
```

5.5 Types for Reflection

Template expressions only need to be syntactically correct to pass a typechecker in Template Haskell and Rosetta. They are not fully typechecked until interpreted or spliced back into a program. Thus, `[| 1 + 'a' |]` and `<< expression 1 + 'a' >>` are valid, but splicing or unquoting these expressions is not. Since no splicing or unquoting may occur at run-time, statically reflective languages can still guarantee type safety.

The type of a template expression in Rosetta is whatever tag appears as the first argument to the quotes, so the type of `<< expression 1 + 'a' >>` is `expression`, or for an AST structure constructed with a "make" function, the type is indicated by the function name. For example, `make_complete_facet_declaration` generates syntax of type `complete_facet_declaration`, which is a subtype of `facet_declaration`, which is a subtype of `declaration`.

MetaML types quoted expressions more strictly than Template Haskell and Rosetta. For example, the type of `< 1 + 2 >` is `< int >`, indicating not only that it is syntactic code, but also that it represents an integer. As a multi-stage language, MetaML allows nested brackets, so the type of `< < 1 + 2 > >` is `< < int > >`.

The type represented by a syntactic structure in Template Haskell or Rosetta is essentially unknown until it is spliced. Thus, splicing a syntactic structure and then using it in an expression could produce a type error. For example, the type error in the following expression is not found until the splice is evaluated:

```
[1,2,3] ++ $(1 + 2)
```

Thus, type errors due to spliced expressions cannot be found until after the splice is evaluated. This leads to a 2-stage typechecker, with compile-time evaluation being performed between the two stages. MetaML avoids this issue by requiring that all syntactic structures represent only type-safe expressions and performing typechecking in a single "once and for all" stage before any evaluation.

Chapter 6

Implementing Multi-Stage and Reflective Languages

This chapter discusses how to implement staged reflection in an interpreter or compiler. It first extends the example interpreter from chapters 3 and 4 with Scheme-like metaprogramming using InterpreterLib. Then, the implementation of reflection for Rosetta is discussed in section 6.2.

The InterpreterLib example is not truly reflective because it does not include an exposed AST and constructs to manipulate that AST. It also does include an interpretation construct, such as `eval` in Scheme, or `value` in Rosetta. However, it shows how to define staging constructs for a language, and sets up the basis for the implementation of reflection in Rosetta.

The entire Rosetta toolchain, called *Raskell*, is implemented in InterpreterLib. The Raskell toolchain is scheduled for release in October and will be available on the SLDG website [35]. The abstract syntax tree for Rosetta is defined as an ordinary algebraic datatype in Haskell and the parser is written using Parsec. The Raskell AST is provided in the appendix (section A.3). From this recursive AST (the *Rast*), AlgC generates a non-recursive AST (the *NRast*), including functor signatures, miscellaneous boilerplate code, and a *convert* function that converts from the Rast to NRast. All forms of analysis, both static and dynamic, operate on some form of the non-recursive AST. Many analyses rewrite the NRast to a specialized version that is more appropriate for the particular kind of analysis. Rewriting the AST is easily accomplished by defining a generic algebra in InterpreterLib. Figure 6.1 provides a graphical representation of the toolchain, which performs the following steps:

1. Parse Rast and convert to NRast
2. First-stage typechecking, modified to be aware of quote and unquote constructs.
3. Elaboration, which resolves all reflection statically.
4. Second-stage typechecking, to find any errors that may have been introduced during elaboration. The algebra itself is the same one used for first-stage typechecking.
5. A dynamic analysis, such as simulation or synthesis.

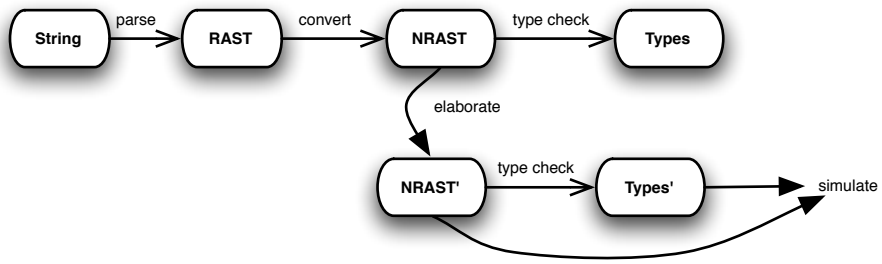

Figure 6.1. Raskell Toolchain

Implementing reflection in Rosetta is considerably different than in the other languages discussed in this paper. Semantics for multi-stage and reflective languages are discussed in the literature [7, 9, 25, 36–38].

My implementation draws from these semantics, but differs heavily due significant differences between Rosetta and other reflective systems. These differences are as follows:

- Rosetta is not an executable language, and the denotations of many constructs, such as temporal constructs, are not known at elaboration time. Only the *interpretable subset* of Rosetta may be evaluated at compile-time.
- Rather than being integrated into a typechecker and evaluator, reflection is implemented as an elaboration phase separate from all other analyses. Other than the generic updates to the typechecker, no other existing algebras are used or modified.

- Quote and unquote control conversion between Raskell's NRast and the AST structures defined in Rosetta. The only form of evaluation that they control is that of the *reflective language* (see below).
- Instead, the `value` and `typeof` functions are separate interpretations constructs.
- Undefined identifiers inside template expressions are dynamically scoped; they do not have to be in scope until the template expression is unquoted. The value function performs partial evaluation, evaluating sub-expressions that are fully defined and leaving undefined identifiers alone.
- Rosetta has a fairly complex type system, including dependant types and types as terms in the language.
- Rosetta's reflective system is implemented in InterpreterLib.

The *interpretable subset* of Rosetta includes all constructs, functions, and operators that could be evaluated statically. There are two subsets of this language: i) The *expression language*; and ii) The *reflective language*. The expression language includes arithmetic, Boolean, and list operators, if expressions, and various other functions that operate on and yield expressions. The reflective language includes all functions that can yield AST structures, including the constructors, recognizers, and observers of the AST structures, if expressions, and the `value` and `typeof` functions. Note that if expressions, as well as several other functions and operators, are in both sub-languages because they can yield either Rosetta values or AST structures.

The expression language is represented in both the Raskell AST and the Rosetta AST structures. The `value` function operates on the subset of the reflective language that represents the expression language.

6.1 InterpreterLib Example

InterpreterLib's generics and algebra combinators make it easy to add simple metaprogramming constructs to a language. In this section, the interpreter from chapter 4 is extended with quote and unquote syntax and semantics. Extensions to the typechecker and evaluator are minimal and generic, requiring no direct modification of the original algebras. Turning on and off the typechecker and evaluator is controlled by a switch algebra.

The algebras for this language were extended to support floats in section 4.4. The implementation in this section is an extension of the `tyAlg'` and `evalAlg'` algebras defined in that section. The `Dec` and `Inc` operators can also be supported by defining an `elabQ` function that is identical to `elab` but ascribed a different type:

```
elabQ :: Fix (Ops :$: Quotes :$: LangS) -> Fix (Quotes :$: LangS)
elabQ = cata elabAlg
```

First, new syntax must be defined in AlgC for the quote and unquote constructs:

```
signature Quotes = Quote *
                 | UnQuote *
```

The semantics for these constructs are the same as quasiquote and unquote in Scheme. The *quote level*, an integer indicating how deeply nested quotes are at any given context, is maintained in the Reader-equivalent Switch monad. A `Quote` causes its argument to be evaluated with an incremented quote level, whereas an `UnQuote` causes its argument to be evaluated with a decremented quote level. At every node in the AST, the switch algebra selects whether or not to apply the interpretation algebra (`tyAlg'` or `evalAlg'`). A quote level of 0 causes the algebra to be applied. Otherwise, typechecking or evaluation is turned off, as shown in figures 6.3 and 6.4.

The HoFix monad must be redefined to support the switch algebra. The definition of this new monad is as follows. It is parsed by MonadBuilder to generate the appropriate Haskell definitions, including the `runEvalMetaMonad` that expects an initial quote level, environment, and global bindings.

```
monad EvalMetaMonad v = Switch Int
                      . Reader [(String, v (EvalMetaMonad v))]
                      . State [(String, v (EvalMetaMonad v))]
```

Extensions to the syntactic space, value space, and type space are provided in figure 6.2. The `Quotes` functor is added to the syntactic space, and the value space is extended to include syntax (a "dynamic value") because reflective languages expose the abstract syntax to the programmer as values in the language. Finally, `Syntax` is added to the type space. Any quoted expression is of type `Syntax`.

Note that only the syntactic space is actually extended, whereas the value space and type space are redefined. Also, the value space is defined as an algebraic datatype instead of a composite functor. We show how to extend a functor value space and define a more modular metaprogramming extension to a language in another paper [40]. In this paper I use a less modular algebraic datatype value

space and implementation for simplicity. `SubType` instances of the algebraic value space are trivial to define and are not provided.

```
type MetaLangS = Quotes :$: LangS
type MetaLang = Fix MetaLangS

data MetaValue m = N Int | D Float | B Bool | U | DV MetaLang
                 | F (MetaValue m -> m (MetaValue m))

data Ty = TyInt | TyFloat | TyBool | Ty :-> Ty | TyUnit | Syntax
```

Figure 6.2. Metaprogramming extensions to syntax, semantics, and types

The extended typechecker is defined in figure 6.3. It generically updates the original typechecker, adding support for the metaprogramming constructs in less than 15 lines of code. The solution consists of two algebras, `tyQuote` and `switchTyAlg`. The former defines typechecking for the new `Quotes` functor, by simply incrementing the quote level for `Quote` and decrementing it for `UnQuote`. The latter selects between the original `tyAlg'` and a "disabled" typechecker based on the quote level.

```
tyQuote = switchAlg (mkAlg . phi) where
  phi n (Quote t) = localSwitch succ t
  phi 0 (UnQuote t) = fail "Top-level␣unquote"
  phi n (UnQuote t) = localSwitch pred t

switchTyAlg = switchAlg (mkAlg . alg) where
  alg 0 t = apply tyAlg' t
  alg n t = do
    x <- sequence t
    when (x==x) (return ())
    return Syntax

tyAlgQ = tyQuote @+@ switchTyAlg

runTyQ :: Fix (Quotes :$: LangS) -> Ty
runTyQ t = evalState (runReaderT (runSwitchT
                                  (cata tyAlgQ t) 0) []) []
```

Figure 6.3. Extended Typechecker for Metaprogramming

The `sequence` function is not the one defined for lists. It is a more general function that operates on any *traversable* data structure. It is used to sequence all monadic computations in a functor. Note that the `sequence` function for lists is just a specific case of this function, when `t` is `[]`. The type of `sequence` is

```
sequence :: (Monad m, Traversable t) => t (m a) -> m (t a)
```

The `Traversable` class is very similar to the `Functor` class. It includes a structure-preserving function, `traverse`, that maps a function onto a data structure. All InterpreterLib functors are traversable, because instances of `Traversable` are automatically generated by AlgC. These instances define `traverse f` as directly applying `f` to each recursive node, rather than pushing `f` deeply into each subterm, the same strategy used for defining instances of `Functor` in InterpreterLib.

Although typechecking is disabled when the quote level is more than 0, the `alg n t` cause cannot simply be defined as `return Syntax`. This is because the quote level might be reduced back to zero by an unquote in one of the subterms of `t`, at which point typechecking must be re-enabled. So, all monadic effects from the subterms of `t` should be executed in sequence, and as long as the sequence succeeds (i.e. there are no type errors in the subterms of `t`), then we can `return Syntax`.

Because the result of typechecking a quote might not ever be used, Haskell's laziness can cause some errors to never occur. We must explicitly force `x` to be evaluated so that any errors occur. The `seq` function is supposed to accomplish this, by taking in two arguments, strictly executing the first and ignoring its result, and yielding the second. However, laziness allows the `seq` function itself to never be evaluated because its result is never requested. The only known solution is to explicitly compare `x` to itself.

The `tyQuote` and `switchTyAlg` algebras are combined to create a typechecker for `MetaLang`, and the `runTyQ` function is defined using the run functions for the State, Reader, and Switch monads.

Extending the evaluator is a little more complicated. The evaluator itself is defined in figure 6.4, but it makes use of a reification framework that must be explained first.

Every unquote appears within a quote, and the result of evaluating an unquote's argument must be reified back into the syntactic space to be combined with the rest of the unevaluated syntax in its enclosing quote. As an example, consider the following term

```
Quote (Add (Num 1) (UnQuote (Add (Num 2) (Num 3))))
```

which should evaluate to

```
Add (Num 1) (Num 5)
```

The result of evaluating the argument to the unquote, `Add (Num 2) (Num 3)` is an integer, `5`, in the value space. This is converted to syntax, `Num 5`, and spliced back into the rest of the AST within the quote. Defining reification from values to syntax is easy to do for most elements in the value space, and is implemented as a `Reify` class for modularity:

```
class Reify val syn where
  reify :: val -> syn

instance Reify (MetaValue m) MetaLang where
  reify (N x) = mkNum x
  reify (D x) = mkFloat x
  reify (B True) = mkTru
  reify (B False) = mkFls
  reify U = mkUnit
  reify (F _) = error "reify␣not␣defined␣for␣functions...␣yet"
  reify (DV x) = x
```

Functions are somewhat tricky to reify, though this is a solved problem [2, 31]. For simplicity, they are not supported here.

Evaluation should be turned off in the context of a non-zero quote level. To "evaluate" a term in a non-zero quote level, all subterms should be monadically bound (one of them might include an unquote that turns evaluation back on) to yield a value for each subterm. Each of these values should then be reified into syntax. Finally, each subterm of the term being evaluated should be replaced with its reified value. The result will be a syntactic term with the same constructor, injected into the value space, but with each syntactic subterm potentially reduced. This "id" algebra is implemented very generically as the following `reifyAlg`:

```
reifyAlg :: (Monad m, SubType MetaLang v, Reify v MetaLang)
            => Algebra MetaLangS (m v)
reifyAlg = mkAlg phi where
    phi t = do v <- unwrapMonad (traverse (WrapMonad . liftM reify) t)
               returnInj (In v :: MetaLang)
```

The `reifyAlg` algebra traverses all subterms, binding them to yield a value and reifying the result. Then, the resulting term is injected into the value space as dynamic syntax. The constructor `WrapMonad` constructs a traversable algebraic datatype from a monadic computation, and the function `unwrapMonad` converts the data structure back to a monadic computation. They are defined in the Haskell module `Data.Applicative`. The `reifyAlg` algebra essentially performs

`traverse (liftM reify) t`; these two functions just make it possible.

The extended evaluator is defined in figure 6.4. The algebra `algQuote` is analogous to `tyQuote` above; it modifies the quote level based on syntax. A top-level quote or unquote construct is removed as its argument is evaluated, whereas nested quotes and unquotes remain. This is because evaluation stays off until all quotes are escaped with unquotes.

A switch algebra, `evalAlgQ` selects between two algebras. When the quote level is 0, `algQuote` is applied to quotes and unquotes, and `evalAlg'` is to everything else. When the quote level is more than 0, `reifyAlg` is applied, but updated so that quotes and unquotes are handled by `algQuote`.

Like the evaluator in section 4.4, this one depends on type information to choose how to evaluate arithmetic expressions. Thus, a sequence algebra is used to sequence the meta-aware typechecker and evaluator algebras.

No interpretation function, analogous to `eval` in Scheme, has been defined. Staging is accomplished by simply applying `reify` and then `runEvalQ` again. For example,

```
runEvalQ (Quote (Add (Num 10) (UnQuote (If Tru (Num 1) (Num 2)))))
```

evaluates to

```
Add (Num 10) (Num 1)
```

which, if `runEvalQ . reify` is applied to it, evaluates to

```
11
```

Note that typechecking must be performed between stages, because the result of evaluating a quote could yield an invalidly typed term.

6.2 Raskell Implementation

An elaborator for Rosetta is implemented as part of the Raskell toolchain. Elaboration is performed statically, right after typechecking and before any dynamic analysis such as simulation or synthesis. At the end of elaboration, typechecking is performed once more.

The base typechecker for Rosetta is developed separately from the elaborator and is unaware of any reflective constructs, such as quote, unquote, value, typeof,

```
algQuote :: (SubType MetaLang v, MonadSwitch Int m)
           => Int -> Algebra Quotes (m v)
algQuote n = mkAlg (phi n) where
  phi 0 (Quote t) = localSwitch succ t
  phi 0 (UnQuote t) = error "Top-level␣unquote"
  phi 1 (UnQuote t) = localSwitch pred t
  phi n (Quote t) = do
    (t' :: MetaLang) <- checkTypeM (localSwitch succ t)
    returnInj (mkQuote t')
  phi n (UnQuote t) = do
    (t' :: MetaLang) <- checkTypeM (localSwitch pred t)
    returnInj (mkUnQuote t')

evalAlgQ ty = switchAlg alg where
  alg 0 = algQuote 0 @+@ evalAlg' ty
  alg n = updAlg (const (algQuote n)) reifyAlg

seqEvalAlgQ = tyAlgQ `seqMAlg` (\ty _ -> evalAlgQ ty)

runEvalQ :: MetaLang -> MetaValue (EvalMetaMonad MetaValue)
runEvalQ t = fst t6 where
  t1 = cata seqEvalAlgQ t
  t2 = runSeqT t1
  t3 = runSwitchT t2 0
  t4 = runReaderT t3 []
  t5 = evalState t4 []
  t6 = runEvalMetaMonad t5 0 [] []
```

Figure 6.4. Extended Evaluator for Metaprogramming

parse, and string, though it does understand the algebraic datatypes defined in the reflection package. The reflective modules in Raskell include two major components: i) A generic update to the base typechecker that adds support for reflection, and ii) The elaborator itself.

The updated Rosetta typechecker is very similar to the one implemented in section 6.1. The following additions are made to the base typechecker:

- Quote and unquote turn on and off typechecking just like the typechecker in the previous section.
- The type of a top-level unquote is `top`, a supertype of all other types, allowing it to be used anywhere. The actual type is not known until after elaboration, after which typechecking is performed a second time.
- The type of a quoted expression is whatever tag appears in the quote, so the type of `<< expression 1 + 2 >>` is `expression`.
- The type of `value(t)` is the type of `t`, and will also be a type from the reflective system, such as `expression`, rather than a basic type like `integer`.
- The type of `typeof(t)` is `type` as long as `t` is a syntactic term.

The types for AST structures, such as `expression` and `declaration`, are already declared in the reflection package and available to the base typechecker. The updated typechecker only needs to add typechecking rules for quoted expressions, which is very easy to do.

6.2.1 Elaborator

The elaborator is defined independently of all other analyses, such as typechecking, simulation, and synthesis.

At first, the elaborator does not evaluate any terms, and simply collects bindings and performs variable substitution. This ensures that any identifiers that appear in a top-level unquote will be evaluated within the static scope in which they were declared. This "Elab" mode is a simple term-to-term transformation that does not involve a separate value space. In the future, a value space and reification algebra similar to the one in section 6.1 may be used. This could avoid problems, such as capturing names, by denoting Rosetta syntax to Haskell values.

Constructs that convert from AST structures to Rosetta code, particularly a top-level unquote, cause the elaborator to switch to an evaluation mode. In this

"Eval" mode, the elaborator evaluates constructors, recognizers, and observers for the Rosetta AST structures, as well as the expression language in Rosetta. Template expressions are expanded from Raskell abstract syntax to Rosetta AST structures by switching to an "Expand" mode. The Expand mode switches back to Eval mode when it encounters an unquote.

- **Elab Normal** - Nothing is performed except declaration bindings, variable substitutions, and constructs that convert from syntactic structures to Rosetta code, notably top-level unquotes, by switching modes. The right-hand side of declarations are left as unevaluated syntax, and variables are bound to that syntax.
- **Elab Quoted** - Nothing is evaluated. Declarations bind variables to a flag that indicates that the variable is in scope, but has no value and should not be substituted during elaboration. Variable substitutions are performed only for those which were declared outside the quotes.
- **Eval** - The reflective sub-language is evaluated. Quotes are evaluated by switching to the Expand algebra.
- **Expand** - Raskell abstract syntax is converted to constructed Rosetta algebraic data structures in the value space. An unquote switches back to the Eval algebra.

The rules in figure 6.5 define how the elaborator switches between modes based on quote and unquote expressions. Each rule lists the current mode, the constructor that is encountered, and the action that is performed on the argument of that constructor. Note that in both the Normal and Quoted Elab modes, the RoQuote, RoUnQuote, or TuQ constructor is always reapplied to the result, preserving the structure of the AST.

The value space (defined in figure 6.6) is used during evaluation of a top-level unquote. It currently includes six constructors, the most important of which are `Con`, for represented Rosetta's constructed algebraic datatypes, and `Sym`, for injecting syntax into the value space.

The `Sym` constructor is used during both Elab and Eval modes. In the environment, a variable is bound to syntax using the `Sym` constructor to indicate that the variable does not yet have a value, but is currently symbolic for that syntax. In Elab mode, all variables are bound to unevaluated syntax. In Eval mode, when a variable is

Mode	Constructor	Action
Elab Normal	RoQuote	Switch to Quoted mode
Elab Normal	TuQ	Execute in Eval mode, then splice
Elab Quoted	RoQuote	Error
Elab Quoted	RoUnQuote	Switch to Normal mode
Eval	RoQuote	Switch to Expand mode
Eval	RoUnQuote	Error
Expand	RoQuote	Error
Expand	RoUnQuote	Switch to Eval mode

Figure 6.5. Elaboration rules for switching modes

```
data Value m = Lit RoLiteral
             | Func ([Value m] -> m (Value m))
             | Con RoName [Value m]
             | List [Value m]
             | Label String
             | Sym Lang'
```

Figure 6.6. Value Space for Raskell's Elaborator

looked up in the environment and found to be symbolic syntax, it is immediately evaluated (see figure 6.8). Also, any unbound identifier, `x`, is effectively bound to `Sym (mkRoVar x)`. Many of Rosetta's built-in functions and operators are simply ignored by the elaborator, either because they have not been implemented yet or have no meaning during elaboration. They are ignored by simply not binding them in the initial environment.

Most substitutions performed by the elaborator should not show up in the final result. The final result of elaborating a Rosetta AST should be the same as it was before, except that every top-level unquote is replaced with its spliced result. Also, the result of splicing a top-level unquote could be invalidly typed. Thus, this result must not be used until it can be typechecked. Both of these issues are addressed by introducing the following functor to the AST:

```
signature TuQ = TuQ Int *
```

As the very first step to elaboration, the AST is converted to an intermediate form, an AST that includes this TuQ functor:

```
type Lang' = Fix (TuQ :$: LangS)
```

The introduction of this functor to the AST serves two purposes: i) it acts as a flag

to the elaborator, so that no term that includes the result of a top-level unquote is ever evaluated, no matter how that top-level unquote is passed around due to bindings and substitutions; and ii) It allows each top-level unquote to be tagged with a unique integer, so that at the end of elaboration each top-level unquote in the original AST can be replaced by its spliced result.

So, an elaboration stage consists of the following steps:

1. Replace each top-level unquote with a TuQ, tagged with a unique integer using the State monad. Defined in figure 6.7.
2. Perform elaboration, starting with the *Elab Normal* mode. This "ruins" the original AST by performing variable substitutions that should not show up in the final result. Only the results of each top-level unquote matter.
3. Traverse the resulting AST, gathering the spliced results of all TuQ expressions. The Writer monad is used to build an associative list that associates each TuQ tag with its spliced result. Defined in figure 6.9.
4. Reconstruct the AST. Start with the AST produced in step 1, and replace each TuQ with its associated term from the list from step 3. Afterwards, no TuQ constructs remain. Defined in figure 6.10.
5. Re-typecheck the result.

Steps 1 through 4 are brought together using the `elabOnce` function defined in figure 6.11.

Step 2 of the elaborator consists of the following functions:

- **elabAlg** - The algebra used in both "Elab" modes. It manipulates and depends on the current mode, which is managed by the Switch monad.
- **evalAlg** - The algebra used in "Eval" mode. It is not aware of the mode and does not handle quotes and unquotes.
- **expandAlg** - The algebra used in "Expand" mode. It is not aware of the mode and does not handle quotes and unquotes.
- **evalAlgQ** - An algebra that selects between `evalAlg` and `expandAlg` based on the mode that is managed by the Switch monad, and manipulates the mode when it encounters quote and unquote expressions.

```
genTag = do
  tag <- get
  put (tag + 1)
  return tag

addTuQ :: Lang -> Lang'
addTuQ = flip evalState 0 . flip runSwitchT 0 . cata alg where

  alg = updAlg algExpr (transAlg emptyAlgList)

  algExpr old = switchAlg (mkAlg . phi old)

  phi _ 0 (RoUnQuote t) = liftM2 mkTuQ genTag t
  phi _ n (RoUnQuote t) = liftM mkRoUnQuote (localSwitch pred t)
  phi _ n (RoQuoteExpr tag t) =
       liftM (mkRoQuoteExpr tag) (localSwitch succ t)
  phi _ n (RoQuoteDecl tag t) =
       liftM (mkRoQuoteDecl tag) (localSwitch succ t)
  phi old _ t = apply old t
```

Figure 6.7. Step 1 - Replace each top-level unquote with uniquely tagged TuQ

- **splice** - The function that converts from a Rosetta value back to Raskell abstract syntax. If the value space is rewritten as a functor in the future, then this function can be implemented as an algebra.

The elaborator's `evalAlgQ` algebra is much like the `tyQuote` and `algQuote` algebras from section 6.1. It adds one feature that is worth mentioning. The evaluation algebra expects a variable to already be evaluated when it looks it up in the environment. However, a variable that is used within a top-level unquote may have been declared outside the unquote, in which case the elaboration algebra would have bound it to unevaluated syntax. `evalAlgQ` updates the evaluation algebra with the following algebra, so that if a variable is bound to a syntactic term, it is evaluated upon retrieving it from the environment.

```
phiExpr old (RoVar x) = do
  trace ("Lookup␣" ++ show x) (return ())
  env <- ask
  case lookup x (env ++ initEnv) of
    Just (Sym t) -> para evalAlgQ t
    Just t -> return t
    _ -> error ("Undefined␣identifier:␣" ++ show x)
phiExpr old t = old t
```

Figure 6.8. Algebra to handle unevaluated syntax during Eval mode

```
gatherTuQ :: Lang' -> [(Int, Lang')]
gatherTuQ = execWriter . cata alg where

  alg = transAlg (mkAlg phi @:@ emptyAlgList)

  phi (TuQ tag t) = do
     t' <- t
     tell [(tag, t')]
     return (mkTuQ tag t')
```

Figure 6.9. Step 3 - Gather TuQ expressions into an associative list

```
reconstruct :: [(Int, Lang')] -> Lang' -> Lang
reconstruct env = flip runReader env . cata alg where

  alg = transAlg (mkAlg phi @:@ emptyAlgList)

  phi (TuQ tag _) = do
    env <- ask
    case lookup tag env of
      Just t -> return (removeTuQ t)
      _ -> error ("Reconstruction␣error:␣No␣TuQ␣with␣" ++ show tag)

removeTuQ :: Lang' -> Lang
removeTuQ = cata (transPureAlg (mkAlg phi @:@ emptyAlgList))
    where phi (TuQ _ _) = error "TuQ␣in␣removeTuQ"
```

Figure 6.10. Step 4 - Reconstruct original AST, replacing each top-level unquote with its spliced result

```
elabOnce :: Lang -> Lang
elabOnce t = reconstruct tuqs t' where
  t' = addTuQ t
  elaborated = fst (runEvalMetaMonad (para elabAlg t') Elab [])
  tuqs = gatherTuQ elaborated
```

Figure 6.11. An elaboration stage - steps 1 through 4

6.2.2 Example

Examples of reflection that are currently supported by the elaborator are available in the appendix (section A.5). This section discusses a specific example and how it is expanded, evaluated and spliced during elaboration. Consider the following declarations of reflective terms:

```
if0::expression is << expression if true then 10 else 15 end if >>;

mangle_if::expression is
           if is_if_expression(if0)
           then make_if_expression(if_condition(if0),
                                   false_alternative(if0),
                                   true_alternative(if0))
           else bottom
           end if;
```

Let us consider two terms, `‘mangle_if‘` and `‘value(mangle_if)‘`, and see how they are elaborated. The elaborator replaces `if0` with the right-hand side of its declaration everywhere in the declaration of `mangle_if`.

The elaborator then enters the top-level unquote and switches to Eval mode. It then switches to Expand mode to handle the expansion of `if0` using `expandAlg`. All occurrences of `if0` expand to

```
make_if_expression(true, 10, 15)
```

which is represented in the value space as

```
Con "make_if_expression"
    [ Lit (RoBool True), Lit (RoInt 10), Lit (RoInt 15) ]
```

The recognizer `is_if_expression` is evaluated by the `evalAlg` algebra. It evaluates to true if its argument is a `Con` with the tag `"make_if_expression"`, thus it returns true when applied to `if0`. The if expression is evaluated by the `evalAlg` algebra, so the `make_if_expression` structure is returned. In the evaluation al-

gebra, `if_condition`, `false_alternative` and `true_alternative` are simply defined as returning the first, third, and second arguments, respectively, of `args` in `Con "make_if_expression"args`. Thus, the `make_if_expression` structure evaluates to

```
Con "make_if_expression"
    [ Lit (RoBool True), Lit (RoInt 15), Lit (RoInt 10) ]
```

In the case of `‘mangle_if‘`, the `evalAlg` algebra is done, and this result is then spliced back into the elaborator to yield the following Haskell representation in the Raskell AST:

```
RoIf (RoLit (RoBool True)) (RoLit (RoInt 15)) (RoLit (RoInt 10))
```

which pretty-prints to the following concrete syntax:

```
if true then 15 else 10 end if;
```

In the case of `‘value(mangle_if)‘`, the result is first evaluated using the `value` function, yielding `Lit (RoInt 15)`, which splices to `mkRoLit (RoInt 15)` and pretty-prints as `15`.

Chapter 7

Conclusion

Modular Monadic Semantics and InterpreterLib provide a framework for defining highly modular and extensible analyses. Language syntax and semantics are defined as compositions of smaller languages that can be combined together in a mix-and-match fashion. Algebra combinators and generics facilitate reuse of algebras in InterpreterLib. They allow an algebra to be used in many different analyses, and even be updated to exhibit alternative behavior, without modifying the original algebra.

As shown in section 6.1, extending a language to include staging constructs is quite simple in InterpreterLib. One must update the typechecking and evaluation algebras to support quote and unquote constructs, and this can be done generically without modifying the original algebras.

To make a language fully reflective, a data structure to represent the language's own abstract syntax must be defined in the language itself. Then, expansion and slicing algebras must be defined to convert between the interpreter's internal AST representation of the language and these AST structures defined in the language. Quote and unquote constructs indicate when to expand and splice. The thesis shows how InterpreterLib can be used to add semantics for quote and unquote constructs. A practical example, the Rosetta elaborator, shows that InterpreterLib can be used to add fully reflective features to a language.

7.1 Limitations and Future Work

The elaborator is still in its infancy. The expansion, splicing, and value functions only support a small portion of the Rosetta AST. The framework exists to sup-

port the rest of the AST, however numerous differences between the Raskell AST (defined in Haskell) and the AST structures (defined in Rosetta) complicate this task. For example, a single datatype in the Raskell AST, `RoParameter`, represents all types of parameters, from function parameters to facet parameters. In the Rosetta algebraic datatype, parameters are separated into a generic `parameter` and a specific `facet_parameter`. This means that the way a `RoParameter` expands is based on the context in which it is used, requiring an environment monad to pass that context to the algebra that handles `RoParameter` expansion. Furthermore, there are elements in each AST that simply do not appear in the other AST, making it impossible to translate certain constructors. Either one or both of the ASTs should be changed to reconcile the differences.

Rather than represent all Rosetta AST structures as tagged constructors (using `Con`) in the value space, the "value space" could be defined as a Haskell representation of the Rosetta AST, probably even a fixed point functor, possibly even the same AST defined in Raskell. This would allow the `value` and `splice` functions to be written as algebras, and would make translation between the different AST representations easier.

The `typeof` function is not implemented at all. Because it operates on Rosetta AST structures, not the Raskell AST, the existing typechecking algebras cannot be reused to define the `typeof` function without converting the AST structure to the Raskell AST. The `parse` and `string` functions are also not implemented, though they can be defined easily by reusing the existing parser and pretty printer.

Top-level unquotes are not the only construct that converts from Rosetta AST structures to normal values in Rosetta. The `string` function converts an AST structure to a string. Every AST recognizer returns a Boolean, and numerous other AST functions, such as `label_to_string`, convert from an AST structure to a Rosetta value. Most of these AST functions are defined in a Rosetta package and do not need to be implemented internally by the elaborator. This means that any other analysis after elaboration can still evaluate recognizers by simply parsing and interpreting the reflection package. For example, consider the following expression:

```
if is_if_expression(make_if_expression(...)) then 1 else 2 end if;
```

The elaborator can safely ignore this and allow a dynamic analysis to handle it based on the definitions in the reflection package. However, this will not work if the argument to the recognizer is not a constructor, but instead a template expression:

```
if is_if_expression(<< expression ... >>) then 1 else 2 end if;
```

In this case, the elaborator must statically expand the template expression to

a Rosetta AST structure and allow a later analysis to deal with the recognizer. Fixing this bug in the elaborator should not be at all difficult.

The elaborator does not yet support recursion. Value recursion via `mdo` works for evaluating the expression language, but somewhere in the elaborator a term is being evaluated too strictly, causing an infinite loop.

Lambda abstractions are not yet supported by the elaborator. This means the user cannot define functions that operate on abstract syntax. This is an extremely important feature that should be implemented as soon as possible. However, the variable substitution method used during Elab mode is naive and should be replaced before supporting lambda abstractions, as it introduces problems related to scoping and name capture. A better solution would be to associate an environment with each variable, so that when a variable inside a quote is evaluated, it is done so within the context in which it was declared. Attempts to implement this solution in the past have been unsuccessful.

The `value` function should take in an optional second parameter, an environment in which to evaluate its argument. The elaborator could implement a `get-environment` function that translates an identifier into the environment in which it was declared, to be used by the `value` function.

Appendix A

Miscellaneous Code

A.1 Typechecker for BAFI Language

This section provides the typechecker for the toy language, written in Interpreter-Lib, before the extensions in chapter 4 are applied.

```
tyArith :: (Monad m) => Arith (m Ty) -> m Ty
tyArith (Num _) = return TyInt
tyArith (Add x y) = liftM2 arithHelper x y
tyArith (Sub x y) = liftM2 arithHelper x y
tyArith (Mult x y) = liftM2 arithHelper x y

arithHelper TyInt TyInt = TyInt
arithHelper _ _ = error "Bad␣arguments␣to␣arithmetic␣operator"

tyBool :: (Monad m) => Boolean (m Ty) -> m Ty
tyBool Tru = return TyBool
tyBool Fls = return TyBool
tyBool (IsZero x) = do
  tyx <- x
  if tyx == TyInt then return TyBool
     else error ("Bad␣argument␣to␣IsZero:␣" ++ show tyx)
tyBool (If c t1 t2) = do
  tyc <- c
  ty1 <- t1
  ty2 <- t2
  if tyc == TyBool && ty1 == ty2
     then return ty1
     else error "Bad␣arguments␣to␣If"
```

```
tyFunc :: (MonadReader [(String, Ty)] m) => Func Ty (m Ty) -> m Ty
tyFunc (Var x) = do
  env <- ask
  case lookup x env of
    Just ty -> return ty
    _ -> error ("Undefined␣identifier:␣" ++ x)
tyFunc (Lambda x ty body) = do
  env <- ask
  tybody <- local (const ((x, ty):env)) body
  return (ty :-> tybody)
tyFunc (App f x) = do
  tyf <- f
  tyx <- x
  case tyf of
    (ty1 :-> ty2) -> if ty1 == tyx then return ty2
                     else error "Argument␣to␣app␣has␣unexpected␣type"
    _ -> error "Invalid␣type␣for␣LHS␣of␣app,␣expecting␣function"

tyImp :: (MonadState [(String, Ty)] m) => Imp Ty (m Ty) -> m Ty
tyImp (Set x ty y) = do store <- get
                        put ((x, ty):store)
                        tyy <- y
                        when (tyy /= ty) (error "Unexpected␣type␣in␣Set")
                        return TyUnit
tyImp (Get x) = do store <- get
                   case lookup x store of
                     Just ty -> return ty
                     _ -> error ("Undefined␣identifier:␣" ++ x)
tyImp (Seq x y) = do tyx <- x
                     if tyx == TyUnit then y
                        else error "Invalid␣type␣for␣LHS␣of␣Seq"
```

```
tyAlg = mkAlg tyArith @+@ mkAlg tyBool @+@
        mkAlg tyFunc @+@ mkAlg tyImp @+@ funitAlg

runTy :: Lang -> Ty
runTy t = evalState (runReaderT (cata tyAlg t) []) []
```

A.2 Evaluator for BAFI Language

This section provides the evaluator for the toy language, written in InterpreterLib, before the extensions in chapter 4 are applied.

```
phiArith :: (Monad m, SubType Int v)
         => Arith (m v) -> (m v)
phiArith (Num x) = returnInj x
phiArith (Add x y) = liftM2Sub ((+) :: Int -> Int -> Int) x y
phiArith (Sub x y) = liftM2Sub ((-) :: Int -> Int -> Int) x y
phiArith (Mult x y) = liftM2Sub ((*) :: Int -> Int -> Int) x y

phiBool :: (Monad m, SubType Bool v, SubType Int v)
        => Boolean (m v) -> (m v)
phiBool Tru = returnInj True
phiBool Fls = returnInj False
phiBool (IsZero x) = do
  (xval :: Int) <- checkTypeM x
  returnInj (if xval == 0 then True else False)
phiBool (If c t f) = do
  condition <- checkTypeM c
  if condition then t else f

phiFunc :: (MonadReader [(String, v)] m, SubType (Exp m v) v)
        => Func ty (m v) -> m v
phiFunc (Var s) = do
  env <- ask
  case lookup s env of
    Just x -> return x
    Nothing -> fail ("undefined␣identifier:␣" ++ s)
phiFunc (Lambda s _ body) = do
  env <- ask
  returnInj (Exp (\v -> local (const ((s, v):env)) body))
phiFunc (App xm ym) = do
  (Exp f) <- checkTypeM xm
  y <- ym
  f y

phiImp :: (MonadState [(String, v)] m, SubType () v)
       => Imp ty (m v) -> m v
phiImp (Get x) = do
  store <- get
  case lookup x store of
    Just v -> return v
```

```
    Nothing -> fail ("Undefined␣variable:␣" ++ x)
phiImp (Set x _ y) = do
  store <- get
  yval <- y
  put ((x, yval):store)
  returnInj ()
phiImp (Seq x y) = x >> y
```

```
evalAlg = mkAlg phiArith @+@ mkAlg phiBool @+@
          mkAlg phiFunc @+@ mkAlg phiImp @+@ funitAlg

runEval :: Lang -> Value EvalMonad
runEval t = fst (runEvalMonad (cata evalAlg t) [] [])
```

A.3 Raskell Abstract Syntax Tree

This section provides the definitions of the Rosetta AST as an algebraic datatype in Haskell in the Raskell toolchain. For brevity, a few parts of this AST are left out. Only the parser uses this AST directly. AlgC generates a non-recursive fixed-point AST from these definitions, along with a function to convert from the recursive AST to non-recursive AST. All other analyses in the Raskell toolchain operate on some non-recursive AST, either the one generated by AlgC or a specialized version of that AST.

```
data RoExpr = RoVar RoQName
            | RoLinkedVar RoQName Int
            | RoLit RoLiteral
            | RoLambda [RoParameter] [RoParameter] RoExpr RoExpr
            | RoApp RoExpr [RoExpr] AppType
            | RoLet [RoDecl] RoExpr
            | RoIf RoExpr RoExpr RoExpr
            | RoCase RoExpr [(RoExpr, RoExpr)]
            | RoSet [RoExpr]
            | RoSequence [RoExpr]
            | RoQuant Quant [RoParameter] RoExpr
            | RoParen RoExpr
            | RoTypeName RoType
            | RoUnInterpreted
            | RoUnderscore
            | RoQuoteExpr String RoExpr
            | RoQuoteDecl String RoDecl
            | RoUnQuote RoExpr
```

```
data RoType = Type
            | RoBaseType String
            | RoSubtype RoExpr
            | RoSetType RoExpr
            | RoSeqType RoExpr
            | RoMultisetType RoExpr
            | RoArrayType RoExpr
            | RoDepProduct [RoParameter] RoExpr
            | RoTyVar RoName
            | RoDomainType String [RoType]
            | RoDesUnits   [RoType]
            | RoTyAll String RoExpr
            | RoListOfTypes [RoType]
            | RoTyParamMode String

data RoImportSpec = RoImport [RoExpr]

data RoDecl = -- name type declval
              RoItem RoName RoExpr RoDeclValue |
              -- decl where_clause
              RoGuardedDecl RoDecl RoExpr

data RoDeclValue =
    -- domain qvars params declarations terms  imports exports
    RoFacet RoExpr [RoParameter] [RoParameter] [RoDecl] [RoTerm]
            RoImportSpec RoExportSpec |

    -- domain qvars params decls imports exports
    RoFacetInterface RoExpr [RoParameter] [RoParameter] [RoDecl]
                     RoImportSpec RoExportSpec |

    -- declarations terms imports
    RoFacetBody [RoDecl] [RoTerm] RoImportSpec |

    -- domain parameters declarations imports exports
    RoPackage RoExpr [RoParameter] [RoDecl]
              RoImportSpec RoExportSpec |

    -- domain parameters delcarations improts exports
    RoPackageInterface RoExpr [RoParameter] [RoDecl]
                       RoImportSpec RoExportSpec |

    -- declarations imports
```

```
    RoPackageBody [RoDecl] RoImportSpec |

    RoDomain RoExpr [RoParameter] [RoDecl] [RoTerm]
             RoImportSpec RoExportSpec |

    RoData RoDataType |
    RoEnum [String] |
    RoValue RoExpr

data RoTerm = RoLabTerm RoName RoExpr
            | RoUnlabTerm RoExpr

--                              name   type   mode
data RoParameter = RoParameter RoName RoExpr RoMode

--                             parameters  constructors
data RoDataType = RoDataType [RoTypeVar] [RoConstructor]

--                                  name recognizer observers
data RoConstructor = RoConstructor RoName RoName [RoObserver]

data RoObserver = RoObserver RoName RoExpr

data RoTypeVar = RoTypeVar RoName RoExpr

data RoMode = RoParamMode RoExpr | NoMode

data RoDesignUnits = RoDesignUnits RoName [RoDecl]

data RoLiteral = RoChar Char
               | RoBool Bool
               | RoString String
               ...
               | RoInt Integer

data RoQName = QName [RoName] RoName
type RoName = String

data AppType = RoPrefix | RoInfix | RoPostfix | RoNormal

data RoExportSpec = ExportAll | ExportDefault | ExportOnly [String]

data Quant = Forall | Exists | Sel
```

A.4 Rosetta Algebraic Datatypes for Reflective AST

This section provides a small subset of the Rosetta algebraic datatypes that define the reflective AST. These declarations are provided in the Rosetta libraries `rosetta.lang.reflect.lexical` and `rosetta.lang.reflect.abstract_syntax`.

```
make_label ( s :: label_string ) :: label is constant;

label_to_string ( l :: label ) :: label_string is constant;

expression :: subtype ( nonterminal ) is
  precedence_0_expression;

precedence_0_expression :: subtype ( expression ) is
  precedence_1_expression + precedence_0_operation;

  ...

precedence_13_expression :: subtype ( precedence_12_expression ) is
  primary + precedence_13_operation;

precedence_13_operation :: subtype ( precedence_13_expression
                                     * infix_binary_operation ) is
  sel ( operation :: infix_binary_operation |
        left_operand ( operation ) in primary
        and operator ( operation ) = precedence_13_operator
        and right_operand ( operation ) in primary );

primary :: subtype ( precedence_12_expression ) is
  parenthesized_expression + literal + name
  + tick_operation + anonymous_function + anonymous_facet
  + set_formation + multiset_formation + sequence_formation
  + type_formation + let_expression + if_expression
  + case_expression + quantified_expression;

parenthesized_expression :: subtype ( primary ) is
  data
    make_parenthesized_expression (
      contained_expression :: expression ) ::
    is_parenthesized_expression
  end data;

if_expression :: subtype ( primary ) is
  data
```

```
    make_if_expression (
      if_condition :: expression;
      true_alternative :: expression;
      false_alternative :: expression ) ::
    is_if_expression
  end data;

infix_binary_operation :: subtype ( expression ) is
  data
    make_infix_binary_operation (
      left_operand :: expression;
      operator :: infix_binary_operator_token;
      right_operand :: expression ) ::
    is_infix_binary_operation
  end data;

infix_binary_operator_token :: subtype ( operator_token ) is
  { less_than_equals_token,
    less_than_token,
    or_token,
    and_token,
    plus_token,

    ...

  };
```

A.5 Reflection Examples

These are working examples of reflection in Rosetta. In each example, a list of declarations is provided, followed by what those declarations actually elaborate to. Declarations that do not change are not included in the second list.

The declarations in figure A.1 are some basic expressions that are used in all examples. Figure A.2 shows some examples of dynamic scoping. Note that after elaboration, the second term, `run_z`, will not pass the second stage of typechecking. Figure A.3 gives some basic arithmetic expressions, and shows that the elaborator maintains static scoping for variables that are in scope when they are quoted. Figure A.4 shows an if expression being constructed two ways, once using a template expression and once using an AST constructor, and then being unquoted. It also shows that a normal if expression (neither templated nor an AST structure) can be unquoted if it yields syntax. Figure A.5 shows complicated examples mixing arithmetic and if expressions with partial evaluation. The value function is applied at different levels, controlling exactly how much of each term to evaluate. In figure A.6, constructors, recognizers, and observers are used to reverse the two branches of an if expression. Note that `if0` and `if1` are identical and can be used interchangeably in all examples.

```
x::integer is 1+2;
y::expression is << expression x + 3 >>;
z::expression is << expression x + 3 + a >>;
if0::expression is << expression if true then 10 else 15 end if >>;
if1::integer is make_if_expression(true, 10, 15);
```

Figure A.1. Basic Expressions used in Reflection Examples

```
arith0::integer is let x::integer be 1000
                   in ‘<< expression x + 1 >>‘ end let;
arith1::integer is let x::integer be 1000 in ‘y‘ end let;
arith2::integer is let x::integer be 30 in (x + ‘y‘) end let;
arith3::integer is ‘<< expression ‘<< expression 1 + 2  >>‘ >>‘;
```

```
arith0::integer is let x::integer be 1000 in x + 1 end let;
arith1::integer is let x::integer be 1000 in 1 + 2 + 3 end let;
arith2::integer is let x::integer be 30 in (x + 1 + 2 + 3) end let;
arith3::integer is 1 + 2;
```

Figure A.3. Static Scoping and Arithmetic Expressions

```
splice_if0::integer is ‘if0‘;
run_if0::integer is ‘value(if0)‘;

splice_if1::integer is ‘if1‘;
run_if1::integer is ‘value(if1)‘;

if2::integer is ‘if true then << expression 10 >>
                 else << expression 15 >> end if‘;
```

```
splice_if0::integer is if true then 10 else 15 end if;
run_if0::integer is 10;

splice_if1::integer is if true then 10 else 15 end if;
run_if1::integer is 10;

if2::integer is 10;
```

Figure A.4. If expressions

```
splice_z::integer is ‘z‘;
run_z::integer is ‘value(z)‘;
run_za::integer is let a::integer be 4 in ‘value(z)‘ end let;
```

```
splice_z::integer is 1 + 2 + 3 + a;
run_z::integer is 6 + a;
run_za::integer is let a::integer be 4 in 6 + a end let;
```

Figure A.2. Dynamic Scoping

```
mix0y::integer is ‘<< expression ‘y‘ + ‘if0‘ >>‘;
mix1y::integer is ‘<< expression ‘y‘ + ‘value(if0)‘ >>‘;
mix2y::integer is ‘<< expression ‘value(y)‘ + ‘value(if0)‘ >>‘;
mix3y::integer is ‘value (<< expression ‘y‘ + ‘if0‘ >> )‘;

mix0z::integer is ‘<< expression ‘z‘ + ‘if0‘ >>‘;
mix1z::integer is ‘<< expression ‘z‘ + ‘value(if0)‘ >>‘;
mix2z::integer is ‘<< expression ‘value(z)‘ + ‘value(if0)‘ >>‘;
mix3z::integer is ‘value (<< expression ‘z‘ + ‘if0‘ >> )‘;
```

```
mix0y::integer is 1 + 2 + 3 + if true then 10 else 15 end if;
mix1y::integer is 1 + 2 + 3 + 10;
mix2y::integer is 6 + 10;
mix3y::integer is 16;

mix0z::integer is 1 + 2 + 3 + a + if true then 10 else 15 end if;
mix1z::integer is 1 + 2 + 3 + a + 10;
mix2z::integer is 6 + a + 10;
mix3z::integer is 6 + a + 10;
```

Figure A.5. Complicated Mixed Expressions

```
mangle_if::expression is
           if is_if_expression(if0)
           then make_if_expression(if_condition(if0),
                                   false_alternative(if0),
                                   true_alternative(if0))
           else bottom end if;

splice_if::integer is ‘ mangle_if ‘;

run_if::integer is ‘ value(mangle_if) ‘;
```

```
splice_if::integer is if true then 15 else 10 end if;

run_if::integer is 15;
```

Figure A.6. Reverse the Branches of an If Expression

References

[1] P. Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

[2] O. Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1996*, pages 242–257, 1996.

[3] L. Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters., 1995.

[4] L. Erkök and J. Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, September 2000.

[5] L. Erkök and J. Launchbury. A recursive do for Haskell. In *Haskell Workshop'02, Pittsburgh, Pennsylvania, USA*, pages 29–37. ACM Press, Oct. 2002.

[6] D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[7] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *International Conference on Functional Programming*, pages 74–85, 2001.

[8] W. L. Harrison and S. N. Kamin. Metacomputation-based compiler architecture. In *Mathematics of Program Construction*, pages 213–229, 2000.

[9] J. Hook and T. Sheard. A semantics of compile-time reflection, 1993.

[10] P. Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

[11] G. Hutton. Fold and unfold for program semantics. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34, pages 280–288. ACM Press, New York, 1998.

[12] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

[13] G. Kimmell, E. Komp, and P. Alexander. Building compilers by combining algebras. In *ECBS*, pages 331–338, 2005.

[14] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[15] J. E. Labra Gayo, M. C. Luengo Díez, J. M. Cueva Lovelle, and A. Cernuda del Río. LPS: A language prototyping system using modular monadic semantics. In M. van der Brand and D. Parigot, editors, *Proceedings 1st Workshop on Language Descriptions, Tools and Applications, LDTA'01, Genova, Italy, 7 Apr 2001*, volume 44(2). Elsevier, Amsterdam, 2001.

[16] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[17] R. Lammel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. *SIGPLAN Not.*, 39(9):244–255, 2004.

[18] R. Lammel, J. Visser, and J. Kort. Dealing with large bananas. In J. Jeuring, editor, *Workshop on Generic Programming*, 2000.

[19] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058, pages 219–234. Springer-Verlag, 1996.

[20] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[21] C. McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002.

[22] U. Meyer. Techniques for partial evalutation of imperative languages. In P. Hudak and N. Jones, editors, *Symposium on partiaal evaluation and symantics based program manipulation*, volume 26, pages 99–105. Yale University, New Haven, CT, sep 1991.

[23] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.

[24] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[25] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage, 2003.

[26] F. Nielson and H. R. Nielson. Two-level semantics and code generation. *Theor. Comput. Sci.*, 56(1):59–133, 1988.

[27] F. Nielson and R. H. Nielson. Automatic binding time analysis for a typed &lgr;-calculus. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–106, New York, NY, USA, 1988. ACM Press.

[28] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[29] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[30] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[31] T. Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 22–35. ACM, 1997.

[32] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[33] G. L. Steele. Building interpreters by composing monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[34] Systems Level Design Group. Interpreterlib. `http://www.ittc.ku.edu/Projects/SLDG/projects/project-InterpreterLib.htm`.

[35] Systems Level Design Group. Raskell Rosetta Utilities. `http://www.ittc.ku.edu/Projects/SLDG/projects/project-raskell.htm`.

[36] W. Taha. Multi-stage programming: Its theory and applications. Technical Report CSE-99-TH-002, 1999.

[37] W. Taha, Z.-E.-A. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type safety. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 918–929, London, UK, 1998. Springer-Verlag.

[38] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[39] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[40] P. Weaver, G. Kimmell, N. Frisby, and P. Alexander. Constructing Language Processors with Algebra Combinators. In *Proceedings of the ACM International Conference on Generative Programming and Component Engineering (GPCE'07)*, Salzburg, Austria, October 1-3 2007. To Appear.

[41] P. Weaver, G. Kimmell, N. Frisby, and P. Alexander. Modular and Generic Programming with InterpreterLib. In *Proceedings of the International Conference on Automated Software Engineering (ASE'07)*, Atlanta, Georgia, November 5-9 2007. To Appear.