A Modular, Algebra-Sequenced Paramorphic Constraint-Based Type Checker for Rosetta

Mark H. Snyder

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. Nancy Kinnersley

Dr. Man Kong

Date Defended

2007/08/27

The Thesis Committee for Mark H. Snyder certifies
That this is the approved version of the following thesis:

A Modular, Algebra-Sequenced Paramorphic Constraint-Based Type Checker for Rosetta

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Abstract

The objective of this thesis is to demonstrate the feasibility of performing interpretation, specifically static type checking, in a particularly modular way. We use a term space of fixpoints of sums of functors so that, by writing individual algebras for each portion of the entire language, we can then combine those algebras into an algebra that functions over the entire target language. The overall computational style employed uses a sequenced paramorphism to reduce the terms to the value space of types. As a proof of concept, this thesis presents a nominal typechecker in Haskell for the language Rosetta. It relies heavily on InterpreterLib, a Haskell library for designing interpreters in exactly the style described.

Contents

Acknowledgements

I'd like to thank my entire research group–my adviser, Perry Alexander, for being a constant source of energy and guidance, as I learn more about computer systems design and the world of research in general; Garrin Kimmell for always offering to help me find out how to fix things that are broken, be they machines, Haskell code, or LaTeX; Nick Frisby, for his ability and willingness to explain Haskell's weirder error messages and all things monadic; Philip Weaver, for teaming up with me and suffering through LangUtils and the entire progression to InterpreterLib for that crucial basic understanding; and Jeni Streb, for keeping us in line during all the classes and lab hours, even though her work is a bit separate at the moment. Also, I'd like to thank my family–a large part of who I am today can only be explained by the love and care I've received every day of my life, and I realize more and more every day just how fortunate I am to have such nurturing family ties. Last but not least, my friends–life's a sad solo tango without friends to share it.

Chapter 1

Introduction

1.1 Statement of Problem

Implementing language interpretation one feature at a time is a tantalizing approach, but can lead to considerable amounts of updating existing code, delaying the addition of more functionality. The benefits of the smaller mental leaps can be grossly outweighed by the legwork to account for integrating the next feature. On the opposite side of the spectrum, implementing an entire complex language interpreter in one step may simply be too difficult if the language is constantly undergoing change. For a large enough system, choosing either extreme can be overwhelming. As the needs of the language are recognized, the syntax or term space of the language may be modified, extended, or retracted. Similarly, the value space, or result of calculation, may also undergo modifications. The ability to cope with change becomes increasingly more vital to the success and survival of the project.

One of the ultimate goals in facilitating interpretation is modularity. For any portion of the development process, the more we can generalize and separate com-

ponents, the better prepared we are for change. We seek modularity in the term space that an interpreter targets, as well as in the value space of the interpreter's results. Modularity is possible in the language's semantics by breaking the language down into smaller meaningful pieces, called semantic algebras. We also benefit from a separation of recursion from the definition of the term space and from the definition of evaluation.

This thesis shows an approach to interpretation utilizing modular monadic techniques to allow for an organic, evolving interpreter that is not as susceptible to many common issues inherent in a large interpreter based on an ever-changing target language. We discuss the evolution of solutions to the common issues, culminating in the present design. While many of these techniques have been understood for some time, the combination of all the techniques discussed, along with the code-generative aspects of InterpreterLib, leads to a systematic and more streamlined process. This thesis is a realization of that process.

1.1.1 Defining the Language

Type checking is a common static analysis that is frequently performed by interpreters. There would be no point in executing code or even compiling the code to be executed, if there was no guarantee that the language's semantics have been obeyed. In order to present the issues involved in writing a relatively large interpreter, a type checker for a subset of the Rosetta language is offered as a proof of concept. An informal description of the components and concepts which comprise the final interpreter is discussed here, to give motivation for the work.

First, I present the term space of the language, with respect to the representation and the likelihood of adding values to it. Issues with defining the term space

recursively as one monolithic structure will be addressed by re-defining the term space in terms of many functors. A functor is essentially a data structure where the subterms have been identified explicitly as subterms, and are parameterized so that a different carrier, or semantic space for the subterms, may be used at different times, such as types during type checking, values during evaluation, and terms themselves when defining terms. If we have functors for different semantic parts of our language and combine them, then we obtain the overall language by taking the fixed point of that combined functor. A sum type is a mechanism for combining functors, and is a disjoint union.

A sum takes two functors, and labels them as being either the 'left' or 'right' functor, and then keeps track of which one is where (via types), for future manipulations. A sum of two functors is itself a functor, so any number of functors may be combined in this nesting fashion, much like the standard `Cons` and `Nil` definition of a list allows us to generate lists of arbitrary length.

Since functors give a non-recursive definition of the term space, we still need to "tie the recursive knot" to gain the complete set of terms with something called a fixed point. The fixed point of a functor is essentially the values possible when the functor's subterms, or carriers, are limited to values of the functor itself. Obviously, there must be some values of functors which do not have subterms, or we could never represent a (finite) term. In creating the term space for Rosetta's internal representation, we take the fixed point of the sum of all the functors used to describe the language. In this fashion, orthogonal features of the language may be defined separately, but still brought together for the chance to interact.

Each functor has a corresponding semantic algebra to analyze the set of features it describes. An algebra is essentially the semantics of a functor's definition,

therefore we can talk about the functor of some algebra. For instance, a functor definition of If-Then-Else could have a corresponding algebra to state that the guard statement's value dictates which branch should be the result. Another example is a functor with a term to represent addition, where the algebra indicates how the two parameters should combine in the definition of addition. From a programmer's point of view, the algebras are the bit of code that define terms' meanings. In the current work, that means there is a type checking algebra for each functor in the representation of the language. Each algebra operates on just its functor, needing only to know the type of the carrier set, known as the value space. If the semantics of the overall language dictates that two algebras do need to know details of one another's functor, only that other functor must be dealt with within the other's algebra. Interactions only occur when they are necessary, and in a tightly controlled fashion.

Each algebra assumes that the subterms have already been evaluated and reduced to values. In order to accomplish this when it is quite likely that the subterms are of other functors, we need an approach to apply the correct algebras to the correct terms. We use the same structure for combining functors in order to write algebras over those functors. This assures us that we can pattern match the exact same labels to apply only the correct algebra to the correct functor. The net result is that a collection of algebras and their collection of functors can be combined to represent the algebra whose functor is the sum of all the functors. Adding or removing parts of a language can be as simple as adding or removing the algebra and functor from this summation.

At this point, we have the means to define a language with functors and to define the semantics of them with functions over those functors to complete the

algebras. We need some way to utilize these separate algebras in evaluating terms; specifically, we need a recursion scheme to handle applying the algebras to the correct terms. The two recursion schemes presented are the *catamorphism* and *paramorphism*. The catamorphism operates on a term by first recursing to the leaves of the term's abstract syntax tree, and then retreating back up the structure, determining the value solely by non-recursive portions of a term and the results of sub-terms' evaluations. The paramorphism is essentially the same, except that the original content of the sub-terms is also made available. This change is only necessary if the decisions of an algebra are based on both the original structure and result of various sub-terms.

Algebras can also be sequenced, such that when the recursion scheme is working its way back up the term's structure, the results of the first algebra are available to the second. A convenient example of the need for sequencing algebras is a pretty printer that must annotate types while printing terms. The type checking algebra can be run just before the printing algebra, which can then look at terms to print them, but also add in types to the output.

Monads are a path to regaining side effects in a pure language, such as Haskell; in implementing the Rosetta type checker, I use a monadic style, meaning that algebras' carriers will be monadic computations. The side effects achieved through monads play directly into the structure of algebras. We still define semantic algebras for the functors in a language and use the recursion schemes presented. Monads essentially define the representation of a computation and rules for when to perform that computation; this allows for localized side-effects, proving very useful for handling topics such as scoping or universal quantifiers.

1.1.2 Benefits

We can make significant changes to the capabilities of the type checker without effecting code upkeep. We add universal quantifiers, which allows us to create type variables. This necessitates the collection of constraints and running a unification algorithm on the results just prior to completing type checking. The only areas affected are those that must be: any constraints that must be generated are done in-place. Algebras that do not generate constraints are not affected. No extra parameter is threaded through, and no systematic change is made. Algebras which are entirely unaffected do not even need to change their type signatures to include an unused monad. Algebras are implemented by functions that pattern match over the separate terms of the associated functor, and each one can simply bind the sub-terms to yield the monadic computation's result and then combine those results into the analysis implied by the semantics of that term.

As will become apparent, the monadic style of computation in functional programming languages is an essential tool of the modularity of the overall design. Although learning monadic programming styles can be daunting at first, there is not too much to learn in order to get started, and once mastered, most would strongly resist giving it up.

One major benefit of using monads is the ability to add new effects to the computing capability by simply adding another monad into the mix. There exist controlled ways–via monad transformers–to introduce more monadic effects with minimal effort; it is very reasonable. Monads might well be loosely described as a way to enforce the computing order of expressions by binding them in a certain order; beyond that, they mostly add functionality by adding functions associated with a particular monad to interleave with each successive binding,

or by maintaining some kind of state implicitly. Haskell provides some built-in syntactic sugar for monads, called do-notation. do-notation is really just a way to list the bindings which will occur, giving a better visual appeal to a multi-nested set of bindings.

The term space changes as more and more features are supported. In the current work, the Rosetta language's concrete syntax is mostly in stasis, yet the internal representation constantly changes. After each milestone of support, more advanced features are tackled. This involves expanding the internal AST as necessary, expanding the type space as necessary, and so on. Yet with a modular style, the effects should be as local as possible. Only the functors and associated algebras will be affected, and then the algebra summation will pull it all back together. This allows us to iteratively write the language itself. There is never a single monolithic representation of all of Rosetta that must be reckoned with at any point in time; one feature at a time can be added and tested. Thus is comprised a particularly modular solution to interpreter development: modular monadic semantics, fixed points of sums of functors for the term space, separate algebras per functor for combination into a complete algebra, and sequencing of algebras as necessary or convenient.

1.2 Contributions of this Thesis

The main goal of this work is to discuss the modular techniques available to interpreter design, present solutions to the issues of development, and to provide a large-scale functioning example. As the example utilizes the techniques heavily, the majority of the discussion focuses on those techniques, and at that point the end result–the Rosetta Type Checker–is essentially a showcase. I discuss the

history of its development thus far, to illustrate the changes necessary in a real application, and the extent to which these techniques were able to handle those changes.

1.3 Organization

Section 2 gives an introduction to Haskell, with the view that the reader is already familiar with some functional programming. Section 3 gives an introduction to what Rosetta is, but only to the extent necessary to motivate the type checking example. In section 4, monads are introduced, as well as the usage in Haskell. Then, functors (section 5) and Algebras (section 5.2) are introduced and related; this sets the stage for the definition of a fixed-point and of a sum, to motivate the term space's shape as a fixed point of a sum of functors. Some enhancements are discussed (section 5.6), as well as common evaluation strategies employed in an algebraic programming environment, namely catamorphisms and paramorphisms (section 5.4). To utilize these in the current framework, I visit the notion of combining algebras (section 5.3), as well as sequencing algebras (section 5.5). Further, an algorithm for gathering constraints on the type of some term, and a unification algorithm to solve that system of constraints, is presented in section 7.1.2. Lastly, a snapshot of the typechecker in its current state is visited (section 7), to help see how all of this comes together, and then conclusions and future work are discussed in section 8.

1.4 Goals

This thesis presents a type checker (appendix A) for a subset of the language Rosetta [2], as an example of interpretation in a modular fashion. The example should motivate the flexibility of this style of programming of interpreters, both for a language in flux and for a large language in general. The language is represented as a fixed point of sums (section 5.1.3) of individual functors representing the different aspects of the language itself. Algebras (section 5.2) corresponding to each of these functors (section 5) are independently written, and then combined in such a fashion that each algebra will be called upon exactly when its own kind of terms are being evaluated; this pulls the algebras together into a single, combined algebra (section 5.3). Furthermore, the type checker is written in a monadic style, separating the different capabilities needed for different portions of the overall language. It also gives much of the power and expressiveness available in functional programming, while still preserving many of the properties guaranteed by a pure language such as Haskell (section 2).

Significant additions to the language should be able to be added sequentially, without significant overhaul of the interpreter's code at any given point in time. For instance, the addition of universal quantifiers requires the ability to generate type variables, and to gather constraints and unify them (section 7.1.2). Using a monadic computational style, we change no code except where universals are now concerned; the state is implicitly woven throughout. Without monads, we might have to explicitly add parameters in all code we eventually want to incorporate. This weighs heavily on the readability of code, as much of this piping merely passes values around, only to be used in the few instances we'd need to edit when using monads. Simpler changes such as adding terms, reorganizing terms, or removing

terms, also lead to changes in just one algebra; even better, we may only need to change the algebra-combination itself. As set as the concrete syntax of Rosetta is, the internal representation is in a constant, gentle flux as its needs are identified and solutions are found. The modular monadic algebraic approach for this type checker is well-suited to these very real issues.

Chapter 2

Haskell

Haskell [8] [13] [12] [7] is a pure, lazy, functional programming language. Features such as higher-order functions, type classes (similar to common imperative languages' interfaces), strong type inference, and a call-by-need evaluation strategy all are good reasons for much of today's language research to be carried out in Haskell, or else in languages with many of the same characteristics. The current work is all performed in Haskell–the tools used and the analyzers written all use Haskell in a modular monadic style.

2.1 Notions of Equality

In Haskell, the equals sign, `=`, is truly Leibniz equality. It implies that the two entities on either side are equivalent and can always be substituted for one another. Thus the concept of assignment is a moot point–we can state that a variable name is equivalent to a value, but we cannot later in time state that the variable is equivalent to a new value: Haskell will rightly complain that there are multiple definitions for the variable. This property of equality is the cause of

Haskell's stance on side-effects. Therefore, we will need some other mechanism to realize side-effects if we want to use them in programming. Monads will be introduced as a mechanism to regain side-effects in a contained fashion while still being able to rely on Haskell's purity elsewhere.

2.2 Purity and Side-Effects

Haskell is a pure language, implying that there are no side effects of any kind. Any time you call a function with the same parameters, you will always get the same results, no matter what. Of course, you can call a function twice with different parameters, and then get different results. Purity in the presence of monads will be discussed when monads are discussed in depth.

Examples of side effects are storing a new value to a variable at a specific point in time, creating new variables, and user input / computational output. Haskell's laziness means that expressions are not evaluated until needed, and thus any side-effects will be delayed until that point. Therefore, the simple idea of print statements throughout code to report on values while progressing through the code as well as values at those points is a tricky, if not impractical affair. An actual parameter might be an expression, replacing all occurrences. Yet, if no occurrence of it is ever explicitly used in the end result, it will never be evaluated, causing no side-effects such as printing. Not only must you be certain that your expression is needed, understanding the exact order in which the interpreter chooses to finally execute expressions can be tricky. Understanding laziness' implications is important.

2.3 Type Signatures

Haskell type signatures are indicated with a double-colon, followed by the type itself, as in the following examples:

```
True :: Bool
not :: Bool → Bool

tertiaryAdd :: Int → Int → Int → Int
tertiaryAdd x y z = x + y + z
```

First, we see that a basic type is just given, in place. Secondly, we see a function, named `not`, that takes in one parameter, that must be a `Bool`, and returns another `Bool`. The arrow (→) indicates that the left side is a parameter, and the right side is the resulting type after applying that parameter. Thirdly, we see a function, `tertiaryAdd` with multiple arrows, meaning that we can apply multiple values as parameters. This function could be called by applying any number of its parameters, as in the following examples with their types annotated (though usually they would not be). At the interpreter prompt:

```
terminal> tertiaryAdd 1 2 3 :: Int
    6
terminal> let addNine = tertiaryAdd 4 5 :: Int → Int
terminal> addNine 6
    15
```

At this point, parenthesization must be addressed. When parsing a type signature with your eyes, you must mentally insert parentheses between arrows to be right-adjusted. For instance, look at the following type signature and its assumed parenthesization:

```
Int → Int → Int → Int
Int → (Int → (Int → Int))
```

This is currying in action. Every function is simply a function of one parameter, that in turn can yield a function or a value. We see that applying more parameters keeps reducing the type:

```
tertiaryAdd :: Int → (Int → (Int → Int))
tertiaryAdd 4 :: Int → (Int → Int)
tertiaryAdd 4 5 6 :: Int
```

Conversely, when applying expressions to functions, we must read with left-most parentheses: `func a b c` should be read as `(((func a) b) c)`. All the parentheses in the following block are optional, as they mirror the order of binding in Haskell. Thus, with typing, if `func :: (A →(B →(C →D)))`, then:

```
(func a) :: (B →(C → D))
((func a) b) :: (C → D)
(((func a) b) c) :: D
```

2.4 Functions as Values

Being a functional language, Haskell fully supports functions as first-class entities: they can be used as parameters, they can be results, they can be anywhere a traditional value is allowed. For instance, the `map` function is a higher-order function that receives a function of type `a → b`, a list of `a`'s, and returns a list of `b`'s. It applies the given function to each element of the list of `a`'s, and constructs the list of all the results as the list of `b`'s. Note that the `map` function uses a function as a parameter, and furthermore, with currying, we could apply just that function, and result in another function:

```
map :: (a → b) → ([a] → [b])
map f [] = []
map f (a:as) = (f a) : as
```

```
inc :: Int → Int
inc x = x + 1

incList :: [Int] → [Int]
incList xs = map inc xs

terminal> :t map inc --:t asks for the type of the expression
     map inc :: [Int] → [Int]
```

2.5 Type Classes

One of the most powerful features of Haskell is the notion of a type class. A type class is a mechanism for asserting that an arbitrary selection of types can be used in certain ways. This is the same idea behind instances in languages such as Java. Indeed, a type class is essentially a collection of functions that must be implemented for any type which is declared to be an instance of that particular type class. As an example, consider the type class `Num`. `Num` facilitates some basic numerical operations, and we will consider a watered-down version for discussion's sake.

```
class Num a where
    plus :: a → a → a
    multiply :: a → a → a
    negate :: a → a
```

This asserts that any type can belong to the type class `Num`, as long as instances of plus, multiply, and negate exist for that class. For instance,

```
instance Num Int where
    plus a b = a + b
    multiply a b = a * b
    negate a = (--a)
```

This provides the actual definitions for the three functions, and now any time a function's type signature includes a type constraint such as `Num a`, `Int` may be used in place of `a`. Consider another example:

```
class Ordered a where
    lessThan, greaterThan, equals :: a → a → a

instance Ordered Int where
    lessThan a b = a < b
    greaterThan a b = a > b
    equals a b = (a == b)
```

Any type can belong to the type class `Ordered`, as long as there is an instance provided for `lessThan`, `greaterThan`, and `equals`. Integers are certainly ordered, as the instance above shows.

2.5.1 Type variables

Type variables are simply variables that represent types. They show up in a variety of languages, from templates in C++, generics in Java, and most every functional language. Haskell completely distinguishes between its types and values, as opposed to languages where the types are values themselves, such as Rosetta or any language based on pure type systems [21]. We can use type variables in Haskell to parameterize a type. Here, we parameterize a `List` structure over its contents, capturing the essence of a list without limiting it to be a list of integers, for instance:

```
--instead of many separate lists types:
data IntList = ILCons Int IntList | ILNil
data BoolList = BLCons Bool BoolList | BLNil

--write one list definition:
data List a = Cons a (List a) | Nil
```

We also see the syntax for creating data structures. We've created a new type, `List`, expecting a type to replace all occurrences of `a`, generating a type such as `List Int` or `List Bool`, which specializes the `List` code for that type. We see that a type variable is simply a way to parameterize a type. Like a function call, we must pay attention to how many parameters are needed to instantiate the function call, or in this case, fully instantiate the type. Yet we should also view the type variable as being a part of the type. `List a` is a valid type as long as there is a type for `a` in context, just as `x+y` is a valid expression as long as there are values for `x` and `y` in context.

Type classes are used to place constraints on type variables in a type signature. When we don't want to say that a type has to be something concrete, such as `Int` or `Char`, yet we don't want free reign such as given by universally quantified types, we can include a Horn clause in the type signature:

```
sort :: (Ordered a) ⇒ [a] → [a]
sort (x:xs) = add lessThan x (sort xs)
sort [] = []

add::(Ordered a) ⇒ (a → a → Bool) → a → [a] → [a]
add f x [] = [x]
add f x (y:ys) = if f x y then x : y : ys else y : (add f x ys)
```

`(Ordered a)` requires that whatever is applied to this sort function, it must be a list of things of a type with an instance for `Ordered`. Since we created an instance of `Ordered` for `Int`s, we can call the sort function on a list of `Int`s:

```
terminal > sort [1,4,2,5,3]
    [1,2,3,4,5]
```

Likewise, if we have an instance of `Ordered` for characters, then we could sort a list of characters with the *same* sorting function.

```
instance Ordered Char where
   lessThan a b = a<b
   greaterThan a b = a>b
   equals a b = (a==b)

terminal> sort ['a','c','b']
     ['a','b','c']
```

We have been off-loading all the real computation to Haskell itself–we are not truly defining what equality and inequalities are for characters, we're simply relying on Haskell's definition. Such is the nature of the example. Type classes involving abstract data types are of course also able to off-load work quite often, where we simply choose what part of the data to compare.

Type classes have a decent parallel to traditional imperative languages: Java's interfaces, and C++'s inheritance of classes containing only abstract methods (and no instance or class variables) generate a similar effect. One important difference to realize, however, is where changes are added. With type classes, we may create an instance of an existing data type for an existing type class, and not change either one. In Java (and similarly in C++), to add an interface's methods to a class, we must modify the class directly. Often, this poses no problem, but at times, we want or need to leave the original source code alone, for instance due to proprietary limitations.

What if we want to only assert equality for certain types, and ordering for others? In this way, Haskell's type classes can be used to create a hierarchy of assured functionality. The above example is actually implemented in Haskell more like the following:

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

```
    (/=) :: a → a → Bool
    x /=y = not (x==y)

class (Eq a) ⇒ Ord a where
    (<),(>) :: a → a → Bool
```

Notice that we can have a set of constraints in designing type classes–not just for functions. This example above says, in effect, "in order to be an instance of type class `Ord`, a type `a` must also belong to type class `Eq`". Thus, any instance of `Ord` may use the member functions of `Eq` confidently. If those `Eq` functions were not instantiated for some type, then it is not well-defined. It either should have been made an instance of `Eq` (to be `Ord`, it will need to have a concept of equality), or else it does not belong in `Ord` (when no concept of equality is legitimate, then ordering also makes no sense).

Another feature of type classes is default definitions. Notice that `(==)` and `(/=)` are both defined *inside* the type class, where we'd normally see only function signatures. As a result, we may create an instance of `Eq` by defining just one, and the other is already created. We can always overwrite these default definitions; any functions with available defaults may either be left out of an instance (and the default used), or included to replace the default. Incidentally, some type classes can be automatically derived (no instance clause is necessary to correctly generate implementations for the functions). `Eq`, `Ord`, `Show`, and `Typeable` are some common type classes which are derivable. Look at this data declaration:

```
data Tm = Num Int
        | Add Tm Tm deriving (Eq, Show)
```

`Eq` of two `Tm`'s is simply a check for identical structure, as well as equivalent `Int`'s where appropriate. `(Add (Num 4) (Num 5))` does not equal `(Add (Num 2) (Num 3))` despite having the same shape. However, we see that the expressions:

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

are identical, and are equivalent. Similarly, `show` converts the structures into printable form (a string) with a healthy sprinkling of parentheses. Not all type classes can be derivable, and derivable classes can always have manually created instances [12].

Chapter 3

Rosetta

Rosetta is a Systems Specification Language being standardized by IEEE. It allows for the specifier to design separate modules, called *facets*, and attribute various model constraints and property constraints to those facets. Rosetta's key contribution is heterogeneity–specifications can be written using different semantics for different components of the specification. A prominent goal is to be able to reason about a specified system's behavior and how its components interact. Rosetta also offers capabilities for abstract modeling, more traditional structural composition, and defining how different domains *interact*. Facets may be defined in separate domains, such as state-based and continuous time, and the interactions will be characterized by the domains from which the facets originated. There is a wealth of information [2] beyond the brief description presented here. As this thesis focuses on the feasibility of certain modular monadic techniques in writing interpreters, the discussion of Rosetta is mostly interested in describing the language sufficiently to reason about its type checking analysis.

The scope of the entire Rosetta language is larger than necessary for the present work. Here I focus on a subset of the Rosetta language, omitting all dependent

typing, and restricting ourselves to type checking a single facet at a time. We will handle all basic types in Rosetta terms–there are twelve numerical types (Bit, Natural, PosInt, NegInt, Integer, Rational, Real, PosReal, NegReal, Imaginary, Complex, Number), as well as Boolean, Char, String, and Element. We also support functions, universal quantifiers, and the various parameters, declarations, and terms of Rosetta Components, Facets, and other design units.

Let's take a look at a Rosetta facet. We will then discuss its basic components, a few details, and the basic requirements for a correct facet.

```
facet Simple (x,y :: input integer;
                z :: output boolean) :: state_based is
     a :: integer;
begin
     t1: a = x - y;
     t2: z = a > 0;
end facet Simple;
```

Figure 3.1.

Figure 3.1 declares a facet, named `Simple`, that has three parameters: `x` and `y` are integers, and `z` is a boolean. Note also that `x` and `y` are inputs, while `z` is an output. Facets are models of some system from a particular domain's point of view. They may have as many inputs and outputs as desired in the interface. After the parameters section, we see the domain `state_based`. This indicates that the facet will be defined under the basic rules of a state-based system, and the syntax reserved for state-based concepts is now available. Specifically, the `state_based` domain and all of its labels are available. Next, we may declare what local entities we want. Here, we have only created `a`, which is an integer. After `begin`, we see the specification portion, defining the logic of this facet. We have two labeled terms. `t1` asserts that `a` is equivalent to `x-y`, and `t2` asserts that `z` is

equivalent to `a>0`.

A facet may be parameterized over any number of inputs, outputs, and design parameters; once fully instantiated, its type is its domain. But for it to be correctly typed, its parameters' types must exist, all declarations must be of valid types and if instantiated, must be correctly typed expressions. Furthermore, all terms must be booleans or facets. Boolean terms state properties that must hold–if the properties don't hold, the component is invalid in some way. Facets may also include packages of other Rosetta items, but we are restricting ourselves to a single facet for the current type analysis. Other Rosetta items may have further assertions, implications, and requirements.

Rosetta allows functions to be used as first-class values; for instance, a facet might input an integer `a` and a function `f :: integer→boolean` as parameters, and use `f a` in a term. This allows for higher levels of abstraction, as more and more general functions can be written in such a fashion and by combining such functions. This gives Rosetta added richness of expression.

Full-fledged Rosetta has a dependent type system; our current work will omit all portions of the language relying on dependent types. While we still can handle universal quantifiers, this will simply mean that all our types will include only type information, and will not include any non-type expressions needing evaluation. There is still a significant set of specifications possible with this restriction on the language, so this is not debilitating to our current discussion.

3.1 Internal Representation

Once a Rosetta file is parsed, it is represented with an internal abstract syntax tree (AST). Essentially, it is a collection of functors (section 5) which are combined

as a fixed point of the sum of them all; furthermore, in order to perform type checking, an algebra for each functor with the carrier type of types themselves is defined, and those algebras are themselves combined into one complete algebra. All of these techniques will be discussed in detail in the following chapters; what this means in a pragmatic sense is that all type checking is performed over this abstract syntax, and thus we will be more focused on this internal representation than the concrete Rosetta syntax itself. The discussions of algebras, functors, and fixed points below are based on the assumption of using such a representation, and thus while it may not look much like Rosetta itself, it is actually quite acutely similar to the structure of Rosetta in this internal representation. The Rosetta AST is called the NRast, or non-recursive ast.

Chapter 4

Monads

Haskell is a pure functional language, and this gains us some good reasoning about our code–at the expense of side effects. The category-theoretical concept of monads [18] [26] [23] was found to be an excellent means of regaining those computational side effects without actually sacrificing the purity of Haskell, nor the functional style with its higher-order capabilities [11] [25] [24] [1] . Different monads exist, and provide different side effects. Monads for maintaining state, for providing context to a calculation, for recording results, or even for the possibility of an error value are examples of common monads.

4.1 Definition

A monad is a functor along with two natural transformations, `>>=` ("bind") and `return`. Further transformations called *non-proper morphisms* may be defined, and are the part where a monad becomes useful and unique from other monads. In Haskell, monads are a part of the language itself, via a type class:

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

Note that `>>` is simply a variant of `>>=` that ignores the result of the previous bind. Its usefulness will be revealed during the discussion of the State monad (4.4.2).

In order to make a functor an instance of the `Monad` class, we need instances of `return` and `>>=` for that functor. Consider this definition of a Maybe monad, that has either a wrapped up value in it, or no value at all.

```
data Maybe a = Just a | Nothing deriving (Show, Eq)

instance Functor Maybe where
    fmap f (Just a) = f a
    fmap f (Nothing) = Nothing

instance Monad Maybe where
    return a = Just a

    Nothing  >>= _ = Nothing
    (Just a) >>= f = f a

fromJust :: Maybe a → a
fromJust (Just a) = a
fromJust (Nothing) = error "tried␣fromJust␣on␣Nothing."
```

`Maybe` is both a functor and a monad. We could have some sequence of operations we'd like to perform on a term, each of which could fail, The code would have been messy originally, but now we have a systematic approach to indicate that a computation could result in either a value or a failure, as well as a mechanism for handling each. Note that `>>=`'s definition states that if an error is already present, we simply return `Nothing`, yet if there is some value available, we unwrap it (with

Haskell's pattern matching) and apply the function, which itself may yield a value or an error. A series of lookups into some structure would be ideal–we first try to look up a variable in some context, then we search within it for some named portion, search further for a sub-named portion, and then we return the portion specified. The first three steps may have failed, and if so, we might as well stop then.

```
lookupEnv :: String → Env → Maybe Tm
lookupTm :: String → Tm → Maybe Tm

find str1 str2 str3  env =
    lookupEnv str1 env ≫= lookupTm str2 ≫= lookupTm str3
```

Note that if `lookupEnv str1 env` evaluates to `Nothing`, the definition of `>>=` applied to `Nothing` will collapse the rest of the computation:

```
     Nothing ≫=    g    ≫=    h
⟹              Nothing ≫=    h
⟹                           Nothing
```

So, if at any state we have an error, that error will propagate onward, and the following computations will be omitted.

4.2 Syntactic Sugar for Monads

Writing code strictly in terms of `>>=` and `return` makes for some perhaps inelegant code, as we see below. Haskell has some built-in syntactic sugar, the `do`-notation, that can make code using monads more readable. The two following sections of code are identical in meaning:

```
(return a) ≫= (λb →
  f b      ≫= (λc →
  g c      ≫= (λd →
```

```
  d   )))

do
    b ← (return a)
    c ← f b
    d ← g c
    return d
```

The effect is to bind the result of `return a` to the name `b`, then apply `f` to `b` and bind the result of that to the name `c`, apply `g` to `c` and call it `d`, and lastly to return `d`. The unique functions included with particular monads fit nicely into this notation–they simply define one more line of the do-notation, sometimes binding a value to a new name, sometimes performing tasks which do not rely on the previous definition's result. It looks strikingly like an imperative-style program. While there is definitely no imperative action being performed, the mere act of binding the different monadic computations in the order given does impose enough restrictions that we can have some temporary side effects. The values bound to names may affect how the inner computations behave. Once we leave the monad, this simulation of side effects is done.

4.3 The Monad Laws

There is a set of monad laws [15] which all monads need to obey:

1. `(return x) ≫= f == f x`
2. `m ≫= return == m`
3. `(m ≫= f) ≫= g == m ≫= (λx → f x ≫= g)`

Rules one and two ensure that `return` is a left- and right- identity for `bind`, and the third provides associativity (up to the introduced lambda abstraction) for `bind`. Monads are used to *represent* computations without the notion of automatically running them as needed. While evaluation is the difference between `2+4`

and `6`, a monadic computation gives us the difference between `2+4` and the task of adding `2` and `4`. One is an expression that represents a value, while the other is a representation of the computation that could be performed or passed around at will. We later on define the meaning of "running" a monad, which then performs the represented computation and yields the resulting value.

There is no mechanism in Haskell for mandating that these laws must hold for a monad. It is the programmer's responsibility to verify them whenever a new monad is created. They are certainly safe to assume for all of the provided instances of `Monad` in Haskell's standard library.

While side effects may be possible in a sense within a monadic computation, it is a contained environment. A monad run on the same values must still give the same result every time, just as a function must return the same result if given the same inputs. Thus, the purity of Haskell is not compromised by monads, but we gain the ability to perform tasks such as error checking, or some stateful computation, as we will see in examples below.

4.4 Some Common Monads in Haskell

Some of the most basic monads provided in Haskell are the Identity, State, Reader, and Writer monads. A brief example usage of each is described, and the basic operations particular to them is given.

4.4.1 Identity

The simplest monad is Identity, which merely does nothing to its contents. Identity has only one constructor, and `>>=` and `return` each have only one corresponding pattern match; Identity also has no non-proper morphisms, meaning

there is no added functionality associated with Identity. What's the point? Just as it is useful to have a Nil for lists or a base case for some induction, it is sometimes nice to include Identity, so that we can have a simple base case for any disussion of monads in general. If we want to talk about monadic computations, but not get caught up in the special features of any one monad, Identity can give us a concrete example that can we can use in code. It can also serve as the bottom of a stack or monad transformers as we see in the next section, 4.6.

```
data Identity a = Identity a

instance Monad Identity where
    return a = Identity a
    (Identity a) >>= f = f a

runIdentity (Identity x) = x

-------

terminal> runIdentity (Identity 4)
     4
```

4.4.2 State

The purpose of the State monad is to allow the storage of some value, the state, that can be either viewed or updated at any point during execution. The State monad provides non-proper morphisms `get` and `put`, which return the current state and take a value to replace the current state, respectively. Consider a simplified definition of `State` and its instance of `Monad`:

```
data State s a = State (s → (s,a))

runState :: State s a → (s → (s,a))
runState (State f) s = f s
```

```
instance Monad (State s) where
    return a = State $ λx → (x,a)

    (State a) >>= f = let (s', a') = f a in
          State $ λx → (x, a')

class MonadState s m | m → s where
   get :: m s
   put :: s → m ()

instance MonadState s (State s) where
   get   = State $ λx → (x,x)
   put s = State $ λ_ → (s,())

data State s a = State (s → (s,a))
```

By itself, `return` results in a monadic computation that has already been supplied a value for a, and is simply waiting for a state to pipe through into the returned pair. Similarly, `>>=` uses pattern matching to ensure that the state which is given to its first parameter, e.g. `State a` above, gets passed through to its second parameter `f`, and finally passing the value generated by applying `f` to the value from the first. With only `>>=` and `return`, State is doing nothing except piping through state and then performing `f x`.

Another type class, `MonadState`, defines the non-proper morphisms that make `State` unique and useful. The type class requires functions `get` and `put`, which obtain the value in the current state and place a new value into the state, respectively. Since `put` doesn't pass a meaningful value on and merely modifies the state that passes through it, it has the resulting type `m ()`. `()` is 'unit' in Haskell, a representation of a 'don't-care' value. This is the perfect use of `>>` (refer to `>>`'s definition above):

```
put 5 >> get >>= (λx → return (x+1))
```

In 4.4.2, we see that `put 5 :: m ()` has placed `5` into the state, and `get :: m b` does nothing with the unit value resulting from `put 5`, which is exactly the situation where `>>` is appropriate. Since we are only interested in the side effects of `put 5`, we can use `>>` instead of `>>=`.

A common use for state in interpreters is to aid generation of unique names for type variables. We simply store a number in the state, and whenever we need a new type variable, we `get` the number, increment it and `put` it back, and then use that number as a suffix to the variable name.

```
generateSymbol = do
     n ← get
     put (n+1)
     return (‘‘v’’ ++ (show n))
```

The order that monadic computations are performed will of course have an effect on what value is available in state. Consider a tree-like structure of monadic computations, where each node will obtain a number from state, increment it, and then allow its sub-trees to run; this would be an in-order traversal for numbering. If instead, each node bound its sub-trees and then obtained a number, this would be a post-order traversal. It is important to note that after a monadic State computation has been run, both the resulting value and the end state are available.

4.4.3 Reader

The Reader monad allows us to look at, but not modify, a value provided as part of the environment. We can also use the Reader monad to create the value to be used in the environment of further monadic computations, thus allowing us to append to or replace the environments for subterms. Because we have not actually changed the current environment, merely added on to the environment

that the subterm sees, all modifications to the environment while in a subterm cannot be seen from the term above it. For instance, a lambda abstraction can add its variable to the current context to be used for the body of the lambda, but the world outside the lambda has no notion of that variable. Since this pattern of adding to and removing from a list of values so exactly mimics the concept of context in type checking (or, for that matter, evaluation), it is quite common to see Reader used to maintain the context during type checking operations. It may contain something as simple as a list of variable/value pairs, to which we may simply concatenate new parts of context, or it may be a more complex structure with associated functions that we can use to assimilate more information, such as a symbol table or some tree structure. We can store anything we want in the Reader, but List-oriented structures tend to be the most common. Also, the notion of context is so deeply linked to the Reader monad, that we often call the associated value the context or the environment, even if we are not explicitly using it for a context or environment.

Reader provides non-proper morphisms to get the current environment (`ask`) as well as give an environment to another computation (`local`), indicating that that computation should be run with the supplied environment. The function `ask` is a monadic computation; it can be bound to a name, holding the current context. `local` takes a function of type `ctxt→ctxt`, a monadic computation, and yields a monadic computation. A common usage is as below. Assume the context is a list of pairs of names and types, and that TypeMapping is the constructor for arrow types:

```
typeof (Lambda name type t2) = do
     env ← ask
     t2' ← local (const ((name,type):env)) t2
```

```
return (TypeMapping type t2')
```

The effect of this function `eval` is to get the current environment and call it `env`; then, adding `(name,type)` to the current environment, run the body using that new environment, returning the type from the domain to the range. `const` is a Haskell function defined as `const a b = a`.

4.4.4 Writer

Writer is a monad that allows us to collect pieces of information at any time throughout the computation. We can `tell` the Writer monad about more values, that will be recorded in a regular fashion. It actually requires a monoid , which has a starting value and a way of combining more values with it. Some simple examples would be the value zero and addition; the value one and multiplication; the value `True` and the operation `&&`. One common usage of Writer is to use the empty list `[]` and the concatenation operation `++`. The only non-proper morphism of Writer necessary for our discussion is `tell`. `tell` takes another value and uses the writer's operation to combine that value with the pool of values already told.

The Writer monad's usefulness in the current work arises with type checking of universal quantifiers. Type checking universal quantifiers requires the creation of type variables, which in turn requires the collection of constraints while traversing the structure, and solving that set of constraints. For constraints collection, whenever another constraint is found in some sub-term, it must propagate to be included in the entire set of constraints. The unification determines (a) the type of the term and (b) if it used its sub-terms in a type-safe manner. This is an ideal use of the Writer monad. Whenever we come across another constraint, we simply `tell` the writer monad of the new constraint, propagating it upwards from

sub-term to super-term. When the monadic computation is completely run, we will get not only the result, but also the total information that was told. For example, telling pairs of necessarily equivalent types:

```
typeof (If b t1 t2) do
     b' ← b
     type1 ← t1
     type2 ← t2
     tell [(b',boolean),(type1,type2)]
     return type1
```

The code above binds the results of computing `b`, `t1`, and `t2`, and then tells the Writer that `b'` must be of type `boolean`, and also that `type1` and `type2` must be equivalent. It then arbitrarily chooses between `type1` and `type2`, and returns that as the resulting type of the entire If-expression.

4.5 Running Monads

Although the Monad class definition does not require it, almost all monads come with a '`run`' function, often having a signature `T a` → `a` for a monad `T`. The concept of running a monadic computation is to 'unpackage' the contents using any special rules implied. All of the side-effects obtained by using the unique functions of a monad will be confined to the area inside the monadic computations. Running a monadic computation with the same inputs will always get the same results, despite there being a local scope in which side-effects are observable, such as state being passed around. We demonstraite using a run function with a common usage of the Reader monad, storing a list of `(String,Val)` pairs and writing a `lookupV` function as below:

```
lookupV :: (Eq t) ⇒ t → [(t,a)] a
lookupV x [] = error "missing!"
```

```
lookupV x ((z1,z2):zs) = if z1==x then z2 else (lookupV x zs)

term :: Reader [(String, a)] a
term = Reader (λ x → lookupV "s" x)

--example implementation of a runX function
runReader (Reader r) x = r x

terminal> runReader term [("r",4),("s",5),("t",6)]
    5
terminal> runReader term [("r",False),("s",True)]
    False
terminal> runReader term []
    * * * Exception: missing!
```

4.6 Monad Transformers

We often want to use more than one monad at a time–perhaps utilizing Reader for context, State for fresh variable names, and Writer for constraints. Thus, we need to create a single monad that contains all the wanted monadic features at once. By themselves, monads don't interact and would have no idea what to do with a monadic computation from a different monad. For instance, the Reader monad has no means of handling State monadic values. We must combine the monads in some fashion. And since monads are defined by instances of type classes, we only need to create something that has all those instances defined. For each monad, we can create a new monad which behaves essentially like the original monad, except there is an extra parameter for an inner monad. For example, there is the `ReaderT` monad, creating values of type `ReaderT r m a`, where `r` is again the environment, `m` is the inner monad, and `a` is again the value generated by running the monadic computation. Just as `Reader r` is an instance of `Monad` (leaving off just the inner value space), `ReaderT r m` is an instance of `Monad`. Thus, we can

place a monad transformer into a monad transformer, stacking them as deeply as necessary [10]. One task remains: we need to make available to the transformer any of the non-proper morphisms from the interior monads, such as `get` and `put` for `State`, `ask` and `local` for `Reader`, and so on. This is called *lifting* a function; the signature for `lift` is seen in the class definition of a monad transformer:

```
class MonadTrans t where
     lift :: (Monad m) ⇒ m a → t m a
```

All of the monads that are pre-defined in Haskell have implemented instances of `MonadTrans` such as `ReaderT r`. Thus, when combining the provided monads, lifting is fully supported. If we created our own monad to use with the provided monads, we would need to define a way to lift those unique features of our monad into any monad transformer that uses our own monad as part of its inner monad, or vice versa. In practice, the monads that Haskell supports are sufficient for most programming tasks, and the task of lifting between monads can often be bypassed entirely, using the libraries' definitions.

Running monads when transformers are involved simply involves running the different monad transformers in the order in which you are nesting them. Just as either an error or a list of values is different than a list of either errors or values, running monads in the "wrong" order will possibly change the results (or be impossible).

```
runIdentity (runReaderT (runStateT x (0,"")) env )
```

In the current work, using monadic computations has made for some relatively simple constraints on type signatures. If a `phi` function needs to utilize some particular monad's side effects, then that function needs a constraint to state that the monad in use (assumedly part of the `phi` function's signature) contains the

features of the needed monad. For instance,

```
phi_var :: (MonadReader Context m) ⇒ Lambda (m c) → (m c)
phi_var (Var x) = do
    env ← ask
    lookupEnv x env
```

Because `phi_var` needs to use the context, we added the constraint `MonadReader Context m`. Any function using multiple monads simply adds a constraint for each one. When algebras (specifically, their `phi` functions) are combined, the overall type is essentially constrained by the union of all those constraints, which is precisely the intention.

Chapter 5

Functors and Algebras

The term space and value space of a language are quite likely to change often during a language's earlier phases of design. This can be particularly inhibiting to an interpreter–any time we either extend an existing language or modify a language, we have to reconsider all code involving the terms or values. Therefore, we must explore ways of representing the term-space in a way that is open to change with little to no modification. Similar issues arise in defining semantics for those terms. We should maintain the code that gives terms semantic meaning in a modular way, so that as the term space grows or changes, we can quickly and clearly add or modify exactly the related portions of code, and nothing else.

Functors are structure-preserving mappings. For our purposes, we may use a definition that states that a functor gives us a structure and a way to map some function through it. The concept of mapping is separate from a particular structure; we can map an increment function through a list of numbers or through a tree of numbers. We could map an absolute value function through a list of numbers, as well as through a tree of numbers or even through some user-defined data type containing numbers, such as some simple term language. In essence, a

functor is both the structure and the knowledge of how to push a function through the sub-terms throughout that structure.

5.1 Functors in Haskell

In Haskell, a Functor type class exists that requires the single function `fmap`, of the form:

```
class Functor f where
    fmap :: (a → b) → f a → f b
```

The type used for f is the functor. That type is a functor because fmap can be called on it, implying that a function can be mapped throughout its structure. We could create any number of different structures, and make each one an instance of functor by defining the appropriate mapping function `fmap`:

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

instance Functor List where
     fmap f (Nil) = Nil
     fmap f (Cons a as) = Cons (f a) (fmap f as)

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

instance Functor Tree where
     fmap f (Leaf a) = f a
     fmap f (Node lb a rb) =
          Node (fmap f lb) (f a) (fmap f rb)

data Tm a = Num a
          | Add (Tm a) (Tm a)
          | Mul (Tm a) (Tm a)

instance Functor Tm where
     fmap f (Num i) = Num (f i)
     fmap f (Add a b) = Add (fmap f a) (fmap f b)
```

It is important to notice that the instance of Functor is `List`, and not `List a`, for example. We would call `List`, `Tree`, and `Tm` functors, because they define a structure along with an instance of `fmap` for pushing through computations in that structure. They do not, however, define what is in the structure. We could use Haskell's default number implementation for `Tm a`, or we could design our own system. Perhaps we want to represent numbers as `Zero` and `Successor` of a number or `Predecessor` of a number. Perhaps we want to have a number system which represents infinity and negative infinity specially, and so we'd use separate constructors for those, and define the meanings of things like multiplying something by infinity. We can then later change our choice of representation, and not have to rewrite code involved in `Tm` itself.

Perhaps even more telling is `List`. We can define many powerful functions over `List`, regardless of what is in the list. Being a functor, we can keep all the details of how to manipulate lists entirely separate from how to manipulate the contents of lists. Notice that `fmap` takes in a function of type `a` $\rightarrow$ `b`, and then results in a function from the functor over `a` to the functor over `b`, or `f a` $\rightarrow$ `f b`. The essence of `fmap` is to intelligently distribute the `a` $\rightarrow$ `b` function throughout the functor's structure, and the type signature beautifully maintains that. It does not matter what `a` and `b` are, as long as their types are respected by the function to be mapped.

5.1.1 Carriers

A functor's contents is often known as the *carrier* of the functor, as the functor describes the shape and the carrier is the type of values found within that shape. If we have `List Int`, we have the functor `List` with carrier `Int`. `List Char` is

again the functor `List`, now with carrier `Char`. Assuming `a /= b`, `fmap` has the capacity to change the carrier of a functor. Just as a function of type `a → b` may be seen as a transformation from values of type `a` to values of type `b`, `fmap` may be seen as a transformation from functors with carrier `a` to functors with carrier `b`, once the underlying transformation has been supplied to the structure-preserving `fmap`. Carriers will be pivotal to the discussion of functors and algebras combined—algebras will expect a certain functor with a certain carrier, and the ability to combine algebras will greatly rely on the ability to share carriers.

When we define many separate functors, they each only know that "something" will be placed in their carriers. Suppose we defined Tm as:

```
data Tm = Num Int | Add Tm Tm | Mul Tm Tm deriving (Show, Eq)
```

These data types are closed in the sense that we know that a `Tm` will only consist of `Num` or `Add` or `Mul` terms. The fact that `Add` contains two *Tms*, instead of two unknown things, is what limits them from being utilized in some larger context easily.

We would like to see a more striated term space as shown below, separating orthogonal concepts as much as necessary. Common groups of terms that often show up in separate functors include basic mathematics, boolean logic with if-else expressions, lambda calculus, and so on. Here, we show the separation of mathematical operations and numbers from boolean terms and if-expressions as a simple example:

```
data Math_F x = Num Int | Add x x | Sub x x

data Logic_F x = Tru | Fls | If x x x

instance Functor Math_F where
    fmap f (Num x) = Num x
```

```
    fmap f (Add x y) = Add (f x) (f y)
    fmap f (Sub x y) = Sub (f x) (f y)

instance Functor Logic_F where
    fmap f (Tru) = Tru
    fmap f (Fls) = Fls
    fmap f (If a b c) = If (f a) (f b) (f c)
```

We need some means to combine `Math_F`, `Logic_F`, and assumedly many more elements into the overall language. Also, we need to somehow enforce the recursion between these elements. The idea is that we can replace any of the `x`'s with any other term of a functor from the group we have in mind, and precisely only those terms. The idea is first to collect the individual functors into one complete functor (achieved with a construct called a sum), and then to 'tie the recursive knot' and ensure that this collection of parts may recurse as deeply as necessary, via a fixed point. We introduce both the sum and fixed point in the following sections, and then introduce algebras to get a better motivation for the value space. This organizational approach contains some significant hurdles to identify and overcome at first, but the work of Duponcheel [4] and Hutton [9] are excellent sources for gaining familiarity.

5.1.2 Sums of Functors

We need a means of combining multiple functors into one functor; the concept of a sum is a concise, simple solution for this task. A sum is a construct which contains either of two types of values at any given point in time, but not both. For instance, numbers and booleans:

```
data Sum x y  = Left x | Right y

type BoolsNums x = Sum (Math_F x) (Logic_F x)
```

```
instance (Functor f, Functor g) ⇒ Functor (Sum f g) where
    fmap f (Left  x) = Left  (fmap f x)
    fmap f (Right x) = Right (fmap f x)
```

The sum of functors is itself another functor. The instance above provides the appropriate `fmap`. Whenever we create a structure to combine functors, such as `Sum`, the general instance of `Functor` looks slightly different. Notice that we seemingly recursively call `fmap` on the right-hand sides; these are actually calls to *other* versions of the overloaded `fmap` function. `Sum`'s task is simply to dispatch the correct `fmap`, and this recursive calling is not the same as the recursion we've been intentionally avoiding in the `Functor` instances for `Math_F` and `Logic_F`. In general, we apply the function to carriers, and re-call fmap on any functors in a term. For instance, we defined the constructor `Cons x (List x)`, and defined `fmap` over `Cons` as `fmap f (Cons x xs) = Cons (f x) (fmap f xs)`.

5.1.3 Fixed Points

One ingredient is missing thus far– our terms are not recursive. We use the well-understood concept of a *fixed point* to allow functors like `BoolsNums` to be recursive. Keep in mind that if we wanted to actually use the functor `BoolsNums`, we would have to supply its carrier type. We've tacitly assumed that `BoolsNums` are themselves in `BoolsNums`, but to create a term, its type eventually needs to resolve to something concrete, such as:

```
(Left (Add 4 5))::BoolsNums Int

(Left (Add (Left (Num 3)) (Left (Num 5)))) :: BoolsNums (BoolsNums Int)
```

At the bottom there is something non-recursive, and with our current capabilities, terms such as `Left (Add 3 (Left (Add 4 5)))` have no correct type. `3`'s place-

ment indicates a type of `BoolsNums Int`, and `4`'s placement indicates a type of `BoolsNums (BoolsNums Int)`. Unless every term is the same depth, we fail to create a legal term. This is far too brittle and causes even more issues at evaluation time. Instead, we need a type able to represent that `BoolsNums` may be included inside itself, and actually be its own carrier when we create a term. Consider the fix-point definition:

```
data Fix f = In (f (Fix f))

out (In x) = x
```

This says that the fixed point of some functor `f` is simply the functor with its own fixed point as the carrier. The `In` constructor is simply there to allow us to pattern match and to allow Haskell to be lazy and not require digging deeply to the "bottom" of our recursion depth. `out` is simply a function which strips off the `In` label. We can now create the fixed point of our `BoolsNums` functor:

```
type Lang = Fix BoolsNums
```

Of course, we could omit this, and simply use `Fix BoolsNums` whenever appropriate. `type` merely creates a type synonym, thus either one is acceptable.

At this time, we have the final version of term space that we'll need. We can reuse any functors in a new sum, and we can tie the recursive knot with a fixed point. We have a modular term space that can be appended easily by only editing our functor sum, and we can reuse the individual functors concurrently in different languages at once. Changes to any single functor are clear, and appropriately have no impact on the other functors. We've separated the task of defining the terms of a feature from the task of defining what features a language has.

5.2 Algebras

An algebra encompasses both a set of terms and operation(s) over those terms and defines the semantics of the set of terms. In general, we use a functor to define the set of terms, and a function to define the meaning of those terms. The function is often called a `phi` function, and has the type `phi::(Functor f)=>f a→b`. Together, a functor and a phi function can comprise an algebra [3] [4].

5.2.1 Examples of Algebras

We explore a few basic algebras, learn their general structure, and then tie algebras back to functors to motivate their usage. We continue with our `Math_F` and `Logic_F` example as above. Consider the phi function below:

```
--recall, data Math_F x = Num Int | Add x x | Sub x x.

phi_MathF (Num n) = n
phi_MathF (Add a b) = a + b
phi_MathF (Sub a b) = a - b
```

The functor `Math_F` defines the chosen mathematical structures, and the `phi_MathF` function gives the reasoning over those things. The same could be done for boolean logic:

```
--recall, data Logic_F x = Tru | Fls | If x x x

phi_LogicF (Tru) = True
phi_LogicF (Fls) = False
phi_LogicF (If b t f) = If b then t else f
```

In both instances, the terms that are placeholders for values simply result in the value (e.g. `Num 5 ==>5`, `Tru ==>True`, etc.), while the terms that contain more complex meaning have to compute their results based on sub-terms' values. We

assume that all carriers, or subterms, have already been reduced to their values. Thus, we can offload the work of addition, subtraction, and if-expressions to the host language if it suits us. Consider the shape of the phi functions:

```
phi_MathF  :: Math_F  x → x
phi_LogicF :: Logic_F x → x
```

This exposes the general form of a phi function. The phi function's parameter is some functor with a particular carrier, and the function reduces that structure down to some value in the value space. For a functor f with carrier c:

```
phi :: f c → c
```

We can therefore write one phi function for each algebra, and we then have a complete and modular picture of the meaning of our terms. What we still lack is a means to combine these phi functions for individual functors into one phi function for the sum of those functors, and a means of providing the recursion scheme our algebras require.

In the following sections, we discuss a means of combining multiple phi functions into one combined phi function, expressly for utilizing our sums of functors, and discuss the type of recursion schemes necessary to utilize these algebras. The key to those recursion schemes is guaranteeing our assumption that the sub-terms have already been evaluated, and correctly ordering the calls to the appropriate phi functions. Keep in mind that even a single algebra and its functor would still need some recursion scheme. We can apply the same recursion scheme to a sum of functors and a sum of phi functions precisely because a sum of functors is itself a functor, and as we shall see, a sum of phi functions behaves just like a phi function over all the combined functors.

5.3 Combining Algebras

Just as we broke the term space into many functors, we would like to write separate algebras for each of those functors, and combine them in some fashion. This gives us modularity in our semantics to exactly mirror the modularity in our term space, allowing us to pick and choose what functors define the term space we want, and then simply combine the respective phi functions. It turns out that the solution for combining phi functions is quite similar to our solution for combining functors–we use a sum structure.

Each phi function has a related functor, and we assume the functors are combined in a sum as described above; therefore, we should rely on that ordering and construct a way to dictate how to apply the correct function for the correct type of value. `funcSum`, when partially applied with functions `f` and `g`, simply pattern matches on the `Left` or `Right` label and applies the correct function. As we nest more functors together in a larger `Sum`, `funcSum` can traverse right through and navigate those same labels to identify the appropriate phi function.

```
data LM_Val = VNum Int | VLog Bool

funcSum f g (Left x) = f x
funcSum f g (Right x) = g x

phi_MathF :: Math_F  LM_Val → LM_Val
phi_MathF (Num x) = VNum x
phi_MathF (Add (VNum x) (VNum y)) = VNum (x + y)
phi_MathF (Mul (VNum x) (VNum y)) = VNum (x * y)

phi_LogicF :: Logic_F LM_Val → LM_Val
phi_LogicF (Tru) = VLog True
phi_LogicF (Fls) = VLog False
phi_LogicF (If (VLog a) b c) = if a then b else c

phi_LMF :: BoolsNums → LM_Val
```

```
phi_LMF = funcSum phi_MathF phi_LogicF
```

We should stop for a brief moment and note that we are dictating what function to apply manually, instead of using a type class to overload some Algebra class with a `phi` function. In Haskell, multiple uses of the same functor for different algebras becomes irksome. This method allows for easier use of algebraically behaving constructs by not requiring the overloaded operator resolution to do the work. In short, a solution involving type classes makes programming with algebras more brittle, and as such we bar that path of discussion.

When we give `phi_LMF` a term of type `BoolsNums LM_Val`, that is a sum of functors, then `funcSum` peels off the `Left`/`Right` labels and the appropriate phi function is called. Note that ordering of the functors and ordering of the phi functions should be the same, and that we strictly right-nest them. This works no matter how many `phi` functions and functors we have. Consider the evaluation steps in the case of three functors and three `phi` functions, named `phi1`, `phi2`, and `phi3`:

```
phi = funcSum phi1 (funcSum phi2 phi3))

phi (R (R (Some term))) ⟹
(funcSum phi1 (funcSum phi2 phi3))  (R(R(Some term))) ⟹
(funcSum phi2 phi3) (R(Some term)) ⟹
phi3 (Some term)
```

As this step-through evaluation shows, the right-nested `funcSum`s directly correspond to the right-nested ordering of the functors in the sum for terms; thus we can use the same algebra's `phi` function and functor in different main `phi`'s, and the means we use to construct those summed phi's is the exact means we use to tell what `phi` to apply.

5.4 Recursion Strategies

Catamorphisms and paramorphisms are two mechanisms for simplifying data structures. They take a regular structure, such as a list, any abstract data type, or in the current work a Rosetta AST, and regularly reduce it to a single value. Both catamorphisms and paramorphisms are ways to apply an algebra to a complex term in stages. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire [16] gives an excellent introduction to the concepts of catamorphisms and paramorphisms, as well as anamorphisms and hylomorphisms.

5.4.1 Catamorphisms

A catamorphism [16] is a means of collapsing a structure into a single value. It requires a function that dictates how evaluation of a term is performed, given the values of its subterms. This is precisely the `phi` functions we've discussed. It also requires there to be some terminal case, such as a functor with no subterms, or a monoid's identity element. The general approach is that the structure is explored to the "leaves" of the structure, and then the values of subterms are calculated and propagated back up the structure.

In the current work, the function to evaluate a term of some algebra once its subterms have been evaluated is the `phi` function from the same algebra. Since we have fixed points of sums for our terms, the catamorphism (`cata`) must first strip off the `In` constructor, then propagate `cata phi` to all subterms via `fmap`. `fmap` is structure-preserving, so at this point, we have that same functor's structure with evaluated subterms. `phi` may be applied, determining the value of the overall term. Either this result is passed up further, or the evaluation is done.

```
cata :: (Functor f) ⇒ (f a → a) → Fix f → a
```

```
cata phi = phi ∘ (fmap (cata phi)) ∘ out
```

The type variable `a` will actually be of the form `(m val)`, where `m` is a monadic structure, and `val` is the result of the analysis. For type checking, that can be some special data type (e.g. `data Ty =...`), but for Rosetta, types are values of the language itself, so we will have `phi` functions of the form `(m Lang)` →`(m Lang)`, where `Lang` is of course the fixed point of the sum of all functors involved. Since `phi` is generating a monadic computation to be run later, we can manipulate the environment of `Reader`, the value of `State`, and other monadic attributes. This gives us the important ability to take information from one part of the term and make it available for subterms *below* that point. It might have seemed that a catamorphism would stop information from flowing down the AST. This would be true if not for the delay introduced by returning monadic computations instead of returning computed values.

A classic example of a catamorphism is the fold, also known as crush. Folding implies reducing some structure to a single value (perhaps a sum, minimum, or maximum, or a boolean describing whether or not some predicate always held). Many list operations are folds, and therefore are catamorphisms. In fact, lists have `foldl` and foldr defined for them already:

```
foldr :: (a→b→b) → b → [a] → [b]
foldr f base [] = base
foldr f base (x:xs) = f x (foldr f base xs)

foldl :: (a → b → a) → a → [b] → a
foldl f base [] = base
foldl f base (x:xs) = foldl f (f base x) xs
```

With `base` being the initial intermediate value, each fold operates from its respective end of the list, and combines the intermediate value and end of the list into

a new intermediate value, traversing across the list until the list is empty. Below is a small example of a functor, an algebra, and the code utilizing a catamorphism to perform the evaluation. We use only one algebra to simplify the discussion, but a sum of phi functions and a sum of functors are equally applicable with the catamorphism as defined above.

```
data F x = Add x x | Num Int
    | Tru | Fls | If x x x deriving (Show, Eq)

instance Functor F where
    fmap f (Add x y) = Add (f x) (f y)
    fmap f (Num x) = Num x
    fmap f (Tru) = Tru
    fmap f (Fls) = Fls
    fmap f (If a b c) = If (f a) (f b) (f c)

data Val = Vn Int | Vb Bool deriving (Show, Eq)

phi (Add (Vn x) (Vn y)) = Vn $ x + y
phi (Num x) = Vn x
phi (Tru) = Vb True
phi (Fls) = Vb False
phi (If (Vb b) t f) = if b then t else f

eval:: Fix F → Val
eval = cata phi

t1 = In $ Num 3
t2 = In $ Num 5
t3 = In $ Add t1 t2
t4 = In $ Tru
t5 = In $ If t4 t3 t2
```

The definition of the functor `F`, the value space `Val` and the phi function `phi` are all exactly as before. The catamorphism operates by pushing a function down into the term, or AST, recursively into each successive layer, until the non-recursive portion of the structure (where there are no subterms) is reached. This reduces all the leaves, as it were, to single carrier values. Now, the terms just above

the leaves are ready for evaluation. The catamorphism's recursive calls finish in tail-recursive fashion, and thus the entire term is finally evaluated, with only a non-recursive function and the catamorphism.

In the current work, the catamorphism handles type-checking itself; we want to reduce a term into a single type, but that type depends on the types of the subterms. Thus a typechecker may be an algebra which operates over terms containing types as the carrier, instead of subterms. This algebra's phi function, combined with cata, allows us to type check any supported Rosetta term.

Monoids, briefly mentioned earlier (section 4.4.4), are structures that always have an implied catamorphism. We pause briefly to explore monoids to better understand the catamorphism and its usage. A monoid is an operation, along with an identity value, that fully describes how to combine multiple values together. Consider zero and addition; if we start with zero and add any number to it, we get another number. Add that number to the next, and we get another number. No matter how many numbers we sequentially add, we still end up with just one number–the sum. `True` and `&&` form a monoid, as do `False` and `||`, `1` and `*`, and `[]` and `(++)`.

In all of these examples, the function has a type of the form `f :: a →b → b`, and the identity element is of type `b`; quite often `a = b`. A monoid has a related phi function by supplying the identity element for the second parameter at all "leaf" terms, and subterms as the second parameter at all successive terms. While our functors and algebras require a bit more packaging with sums and fixed points, we still are just folding a structure into a single value. A catamorphism is sometimes called a general fold, as we can take some structure and fold the structure in repeatedly until we get one single value. Monoids are nice, simple structures that

display this folding-in behavior.

5.4.2 Paramorphisms

Paramorphisms are strikingly similar to catamorphisms. The difference lies in that a paramorphism keeps the structure of sub-terms available *as well as* their values. Thus, any part of an associated `phi` function may now use the structure *and* the results of subterms. If the carrier of a `phi` function were a pair of term and value, we could actually mimic a paramorphism strictly using the catamorphism.

The general type of `phi` function will now have another parameter for the unevaluated term itself:

```
phi :: Fix MySum → F a → a
phi self (F a) =...
```

It is a property of the Rosetta AST that things such as parameters are also a part of the language. As such, if we were to look at a facet with parameters in a strictly catamorphic point of view, these parameters would also be reduced to nothing more than a type while typechecking. For this reason, type checking has been migrated to a paramorphic style. Most of the typechecking has no need for that self-inspection, and can simply ignore the self-parameter which the paramorphism provides.

5.5 Sequencing Algebras

One analysis may depend on the results of another. Thus we find it desirable to have the intermediate results of one analysis available to another algebra at each node of a term. We might perform type checking and provide those results at each sub-term to another algebra that relies on the type of terms to decide how

to interpret terms. In the current work, one example situation is in providing the results of type checking to an algebra that arbitrates between overloaded operators such as `+`.

This introduces the concept of algebra sequencing. We would like to have the results of one algebra available to the `phi` functions of another algebra for every single sub-term. The first algebra should ideally need no knowledge that it will be used in such a fashion. Of course, the following algebra will have means of accessing the results of the first algebra–the approach here is to provide the results as another parameter. The type of algebras used in a sequenced nature should now be (for a Functor F):

```
firstAlg :: ⇒ F (m v) → (m v)
nextAlg  :: ⇒ v → F (m x) → (m x)
```

One excellent learning example is to sequence an identity algebra with a catamorphic algebra. The result of the first algebra, the original term, is presented as information available to the second algebra. We have a paramorphism defined in terms of two sequenced catamorphisms–this is precisely how InterpreterLib defines the paramorphism.

There is a slight increase in complexity when we combine separate components of an algebra that is expecting some other algebra's results to be sequenced to it. We must thread a parameter through the components so that we are still creating an algebra with the correct type signature as we'd always expect. Assuming we've written individual algebras for `nextAlg` above named `nextA1`, `nextA2`, and so on, we could continue in the same fashion as described later in section 6 :

```
alg v = (mkAlg (nextA1 v)) @+@ (mkAlg (nextA2 v)) @+@...
```

A greater gain is the ability to take an algebra that was not written with

sequencing in mind, and then use it as the first part of a sequence of algebras, or to use the same algebra in multiple sequences. A real example is the need to have type-aware algebras in the generation of assembly instructions or the description of hardware. Do we duplicate some part of the typechecking process to find the values we need? No–that would mean we have multiple places that the semantics of typechecking has been defined, leading to difficult code to maintain. With the ability to sequence a typechecking algebra before the code that decides how many bits wide a MUX needs to be, or what version of an overloaded operator to dispatch, we can use the exact code written for typechecking. If they are in co-development, then the changes to the typechecking algebra would be immediately available to the following algebra. Note that sequencing one algebra with an algebra expecting to be sequenced results in another normal looking algebra; at this point, algebras may be chained together as far as desired or necessary.

To understand this, consider the recursive scheme of a catamorphism. The first action is to push the evaluation function down through subterms and let the subterms percolate back up, evaluating results based on subterms' results. In sequencing two algebras, we should think of both algebras as performing the first step (`fmap` pushing a function through the functor's structure). At this point, the first algebra lets the subterms' resulting values be passed up a level and then evaluated. This result is passed to the *second* algebra in determining its value at that level. The first algebra must always keep ahead of the second algebra, but does not need to completely finish before the second algebra can be begun. This relates to the concept of a fusion law [16].

5.5.1 Fusion Laws

Fusion laws determine when two transformations of a structure can be combined, or fused, into one transformation. They allow us to combine two passes through a data structure into one. While not strictly necessary for computation, the fusion laws give solid mathematical reasoning to an optimization of evaluation. Performing fewer transformations on a structure leads to less traversal, and hence less work in general for a computation. Consider the two following functions:

```
f x = 5 * x
g xs = foldr (+) 0 xs
```

We consider mapping `f` to a list (which results in a new list), and then applying `g` to the resulting list (which results in a single number). We can see the evaluation progress:

```
g (map f (Cons 1 (Cons 2(Cons 3 Nil)))) ⟹
g (Cons (5*1) (Cons (5*2) (Cons (5*3) Nil))) ⟹
g (Cons 5 (Cons 10 (Cons 15 Nil))) ⟹
foldr (+) 0 (Cons 5 (Cons 10 (Cons 15 Nil))) ⟹
(1*5)+((2*5)+((3*5)+0)) ⟹ 30
```

But couldn't we also have thought of it as inserting `f` into the definition of `g`, and then applying this new version of `g`?

```
f ( g (Cons 1 (Cons 2(Cons 3 Nil))) ) ⟹
f ( foldr (+) 0 (Cons 1(Cons 2(Cons 3 Nil)))) ⟹
foldr (λx y → (f x)+y) 0 (Cons 1(Cons 2(Cons 3 Nil))) ⟹
((5*1)+((5*2)+((5*3)+0))) = 30
```

Using Meijer's syntax [16], for a function `f` and a catamorphism $(\!|b\ ,\ \oplus\ |\!)$ to be capable of fusing into $(\!|c\ ,\ \otimes\ |\!)$, the following must hold:

$$f \circ (\!|b\ ,\ \oplus\ |\!) = (\!|c\ ,\ \otimes\ |\!)$$

$$\Leftarrow$$

$$(\text{f b} = \text{c}) \wedge ((\text{f } (\text{a} \oplus \text{as})) == (\text{a} \otimes (\text{f as})))$$

Essentially, a fusion law dictates when it is acceptable to combine two separate passes of a structure, fusing the two traversals into one. Notice the base case must be transferred, and then the recursive cases are either given first to `f` or else the entire result is passed to `f`, and the same result should always occur.

With respect to the current work, we have not verified whether a fusion law for sequencing algebras exists. The main question is whether the monadic computations of the first algebra have already been run, or merely constructed. Do different monads behave differently under these circumstances? Hopefully the monadic behaviors of algebra sequencing can be more solidly understood. For now, static analyses are certainly safe. It does not matter whether the monadic computations of the first algebra have been run ahead of time or are run in lock-step, or indeed even re-run, as there is no problem with this evaluation being eager or lazy. In short, while this is an open question, its result does not affect the validity of our results.

5.6 Injection and Projection Techniques

Defining the term space as a fixed point of a sum of functors makes for very useful but sometimes ungainly terms. We briefly discuss an approach for simplifying the usage of such a term space, to contend that it is a feasible approach. We create functions to correctly package terms into the fixed point of the sum of functors, as well as functions to correctly un-package them from that same structure.

For example, we might see something like this:

```
tm1 = In(Right(Add(In(Right(Num 4)))(In(Right(Num 5)))))
```

What is worse is if we were to significantly modify the structure of our sum, as we remove or add functors to our sum, code containing a pattern match may rely on the number of nested `Right`s there are. What we need is a function to handle the addition and removal of the correct labels. The usual names for these functions are `inject` and `project`, or `inj` and `prj`. We also often want a function to help in the creation of terms; here, they are pre-pended with make. There will tend to be one make function for each constructor of each functor. Initially, if we had a fixed point of a particular sum, we could create these functions as such:

```
data F1 x = F1 Int
data F2 x = F2 x
data F3 x = F3 x x
type MySum =Sum F1 (Sum F2 F3)

-- functor instances omitted here...

makeF1 :: Int → Lang
makeF1 x = In (L (F1 x))

injF1 :: F1 (Fix MySum) → MySum (Fix MySum)
injF1 x = L x

prjF1 :: MySum (Fix MySum) → (F1 (Fix MySum))
prjF1 (L x) = x


makeF2 :: Fix MySum → MySum (Fix MySum)
makeF2 x = R (L (F2 x))

injF2 :: F2 (Fix MySum) → MySum (Fix MySum)
injF2 x = R ( L x )

prjF2 :: MySum (Fix MySum) → F2 (Fix MySum)
prjF2 (R(L x)) = x
```

For a simple enough system, this is not a terrible solution. Now we can use

the `prjX` functions to unwrap a value without laboring over the extra constructors in particular. If we modify our sum, we would have to modify the definition of these projection functions as well, but the code using them has the abstraction of a function call to save it from needing to be re-written. Even better, if we tried to project the wrong type of value in it, we would get a non-exhaustive pattern error, so Haskell itself wouldn't let us try to `prj3` a `F2(Fix MySum)` value. While every single `prjX` function has the same-type single parameter, they return differently typed values; hence, our supplying only one pattern-matched version means we are only writing the code for the times when it should be used.

5.6.1 Using a Type Class for Injection and Projection

Using individual functions for projection makes for a simple, usable solution, but there is room for even more generality. Using a type class, we can create one function that gets overloaded, relying on Haskell's type system to infer what instance of projection or injection is appropriate. Even though there will be a different type involved while removing every single constructor, there will always be a definition for `inj` or `prj`.

```
class SubType f sum where
     inj :: f a → sum a
     prj :: sum a → Maybe (f a)

instance SubType f (Sum f g) where
     inj x = L
     prj (L x) = Just x
     prj (R x) = Nothing

instance (SubType f g) ⇒ SubType f (Sum e g) where
     inj x = R $ inj x
     prj (L x) = Nothing
     prj (R x) = prj x
```

If we had a language `Lang` as a fixed point of a sum, we can strip off the fixed point constructor `In`, and then project to the individual value of type `f Lang`. As long as we ascribe it a type, we can pattern-match using the `Just` constructor and remove the `In` and all `R`'s and `L`'s without re-writing code to do so. Similarly, to build a value we inject, ascribe a type, and then use it as we want.

```
terminal>  (inj (F1 5)) :: MySum (Fix MySum)
     In ( L (F1 5) )

terminal> let x = In ( inj (F1 5) ) :: Fix MySum
terminal> prj (out x) :: F1 (Fix MySum)
     Just (F1 5)

terminal> fromJust (prj (out x) :: F1 (Fix MySum))
     F1 someval
```

There are instances where the ascription is not needed. If the result is to be used somewhere, or for instance is immediately used in pattern matching, then the type will be inferred from the usage, giving Haskell the necessary type information.

Finally, we can inject terms into and project terms out of the sum with `inj` and `prj`, and we can get into and out of the fixed point via `In` and `out`, without reference to a particular series of application of constructors. We are free to change the sum or use a functor in multiple sums and are not required to manually construct projection functions for each particular usage–the instance is always available from the general instances. The choice of term space now places no extra burden on changing or repeated usage. The exact same approach is possible with the value space–a value space may be a fixed point of a sum of values, and projection and injection functions may similarly be defined. In summary, the effort of encoding the boxing and un-boxing of values into a conglomerate type is pushed almost entirely onto the host language, with a few simple instances

and overloaded functions. This makes for a significantly robust representation of terms, such that portions may be added or removed at only a high level, leaving the details to be automatically derived.

Chapter 6

InterpreterLib

6.1 Introduction

InterpreterLib [27] is a library of Haskell code to aid in the creation of interpreters in the modular monadic style. It is an implementation of the techniques discussed thus far. InterpreterLib facilitates the creation of functors and algebras over those functors. It also provides code generation support for implementing various techniques such as combining algebras, combining functors, and projecting or injecting terms and values. Many of these techniques rely on specific data structures for a sum or fixed point, and utilize type classes to provide instances of the provided functions.

The programmer writes the functor definitions and phi functions for those functors. Then the programmer uses pre-defined operators to combine those phi functions and to combine the term space of a group of functors. Then, the programmer uses pre-defined operators to sequence multiple algebras, if necessary, and define evaluation patterns as catamorphisms or paramorphisms. The legwork of creating type classes for the meaning of a functor, an algebra, and projectable

and injectable term spaces is complete; sums, fixed points, recursion schemes such as catamorphisms are defined to operate over these structures. Algebra combinators are defined to facilitate sequencing, switching between, and updating algebras. By running the code generation phase on a functor definition, the programmer gets to focus on the semantics of the different portions of the language instead of the implementation of approaches to defining the semantics. The modularity of the approach is exactly as has been described in previous sections. An algebra may be used in many other combinations of algebras without touching the original code.

That is not to suggest that the programmer is oblivious to all that is happening under the hood. A solid understanding of the meaning of a term space as a fixed point of a sum of functors, the recursion scheme of catamorphisms and paramorphisms, and indeed even the meaning of a functor and an algebra are all indispensable for the usage of InterpreterLib. The true benefit is the ability to have generated what boilerplate code is necessary for the modular monadic approach, and to isolate the definition of the semantics.

6.2 Example Usage

Following is a basic utilization of InterpreterLib. We will show three separate algebras for basic mathematics, basic logic, and one for a unit term. We will learn how to define the functors, generate the boilerplate code, write the phi functions, and combine them into a single algebra that encompasses all three functors' semantics as a whole. This not intended to be a complete tutorial on InterpreterLib; rather, it provides an example of the functionality available. There are more thorough descriptions and discussions of InterpreterLib [27].

6.2.1 Creating Functors

First, we will create a description of the functors, each as separate data structures. Unlike the descriptions of functors we've been working with, this contains Haskell data structures that define the terms in a *recursive* format. The functors themselves are generated in the next phase.

```
module Rast where

data MathF = Add MathF MathF
           | Sub MathF MathF
           | Num Int deriving (Show, Eq)

data LogF = Tru | Fls
          | If LogF LogF LogF deriving (Show, Eq)

data UnitF = Unit deriving (Show, Eq)
```

6.2.2 Generating Boilerplate

Next, we call upon the code generator, `algc`, to create the functors themselves, as well as any instances of type classes necessary. Depending on what an algebra will be used for, there may be a need for other instances. For this small example, we will simply ask for all possible instances. We first supply the file, then define what the name of our termspace should be (i.e., the name for the fixed-point of the sum of functors), a name for the file containing all the functors and instances, and finally what instances should be generated.

The file 'Lang.hs' contains the sum of functors and fixed point of that sum. We could have created this manually. If we want to create extra uses of the same functors, we may find ourselves doing just that. 'Make.hs' is the series of functions used to construct the terms with all the appropriate nestings of `Left`s and `Right`s

for the sum, as well as the fixed point label. Conveniently, the naming scheme of prepending 'mk' to each means that terms created in this fashion are nearly the same as terms created for the original recursive definition, but with a lot of 'mk's tagged on everywhere. A critical effect of this file is to assign types to the make functions, when constraints are involved; otherwise, Haskell would not know what type to use when a sum is found, and the type ascription would be mandatory down the road. Lastly, 'NRast.hs' contains the functors and all the required instances of type classes.

6.2.3 Writing the Individual Algebras

At this point, we are done with both the definition of the terms we need for our language and all the boilerplate code to utilize the framework of InterpreterLib. We now need to write the `phi` functions to define the semantics of our algebra. Since we have various value types we'd like to return, we need to create a type that encompasses them all. This could be done with a new data structure, as below, or with an Either type, or the value space could even be its own fixed point of a sum of functors, if it suits us. There is significant leeway in the value space, and as such we have not needed to discuss it in much detail.

```
module Evaluator where

import Regular.Algebra(@+@,mkAlg)

-- to let us run our hardly-monadic code
import Control.Monad.Identity

-- import all the functors we made
import All

data Val = VNum Int | VBool Bool | VUnit deriving (Show, Eq)
```

```
phiMath(Add x y) = do
     (VNum x') ← x
     (VNum y') ← y
     return $ VNum $ x' + y'
phiMath(Sub x y) = do
    (VNum x') ← x
    (VNum y') ← y
    return $ VNum $ x' - y'
phiMath(Num x) = return (VNum x)

phiLog (Tru) = return $ VBool True
phiLog (Fls) = return $ VBool False
phiLog (If b t f) = do
    (VBool b') ← b
    if b' then t else f

phiUnit(Unit) = return VUnit
```

6.2.4 Combining Algebras

Now we can combine the algebras into one algebra. We have not yet created any association between algebras, so there is no collective structure that can handle the semantics of mathematics, logic, and unit all at once. Just as we created the fixed point of the sum of functors to combine the term space, we will do so for the phi functions. It is essential that we respect the order that we added them to the sum. By convention, InterpreterLib adds them alphabetically when you ask for it to generate the sum for you. `@+@` is the operator for combining algebras. We also define evaluation as a catamorphism of the algebra.

```
alg = (mkAlg phiLog) @+@ (mkAlg phiMath)
       @+@ (mkAlg phiUnit) @+@ funitAlg

evalM = cata alg

eval x = runIdentity ∘ evalM

t1 = mkIf (mkTru) (mkNum 4)
```

```
(mkAdd (mkSub (mkNum 5) (mkNum 4)) (mkNum 1))
```

Notice that we added `funitAlg` to the end of our sum; InterpreterLib puts this end-of-the line placeholder functor at the end of our sums when we ask for it to generate the sum for us. All of our functors are used in similar fashion (none are the 'end of the list'), making the code for InterpreterLib more robust. `mkAlg` is a function that helps InterpreterLib understand that the `phi` function is part of an algebra in the mathematical sense, and thus can be used as such with the code in InterpreterLib. It was created by a `newtype Algebra`, to ensure that `apply` may be called on the result, and thus `cata`'s definition can now use this particular `phi` function when it sees the correspondingly correct type of functor. `evalM` creates the catamorphism with our combined algebra; its type is `evalM::(Monad m)=>Lang→m Val`. `eval` runs all the monads we use (none, in this case, so we only run Identity to get to the value) on the monadic result of `evalM`. `evalM` and `eval` are usually written together in some fashion–I've split them up to show the two distinct tasks of creating the catamorphism and running the monads to perform the monadic computations that result. All that remains is to use the evaluation, such as :

```
terminal> eval t1
     VNum 4
```

6.2.5 Further Additions

To add to this language, we would: (1)write the new pieces of the term-space into the 'Rast.hs' file; (2) re-generate the boilerplate with a call to `algc`; (3) write `phi` functions for the new pieces; (4) add `(mkAlg phiNewstuff)` into `alg`; and possibly (5) update `eval` if we used more monads. Notice we have not

edited the original `phi` functions, we gently edited `alg`, and we could never escape writing the new termspace parts or `phi` functions. We also did not have to write more boilerplate code. If we were more careful about the location of the original definitions of the termspace, we could have avoided regenerating the pre-existing boilerplate code. The results are excellent– modularity is preserved, and yet the tasks required for changing the language are straightforward and natural, requiring change practically only where actual change is truly desired.

Chapter 7

The Rosetta TypeChecker

The Rosetta type checker [22] is an interpreter in the modular monadic style: it consists of algebras for all individual functors for representing Rosetta terms. The resulting algebra evaluates terms to type values by using the recursion schemes discussed in section 5.4. We discuss further implemented capabilities as well. As presented, the Rosetta type checker handles basic expressions of the language, as well as the basic design units. This means that an individual facet or component may be type checked. The type checker does not allow the type checking of multiple modules; this is a subject of future work. Similarly, dependent typing is not implemented. The current implementation still represents a large enough piece of software to exhibit the techniques that this thesis must demonstrate.

7.1 Development History

The original type checker handled the most basic of terms using a unique value space (e.g., `data Ty =`...). More and more of the Rosetta language constructs were added, including functions and the design units themselves. During this period,

the built-in standard environment of the language was simply hard-coded into the initial environment. Overloading only exists in Rosetta for the base types such as in addition of the various numerical types; there is currently support for sub-typing in the type checker for those basic types. The result of the type checker will eventually be sequenced with another algebra to annotate the entire term with their types, and then arbitrate between versions of the overloaded operators. This is more a necessity for evaluation and thus has not yet been a focus in type checking. Also, with sequencing of algebras considered one way to break up a task into phased tasks, this is merely an instance where a new feature will be implemented orthogonally to the current work.

7.1.1 Using Paramorphisms

The Rosetta type checker utilizes a paramorphic recursion scheme. With the addition of facets, Rosetta parameters that exist in their own functor brought up another issue: what if we need to look into a structure to find names to add to the context? Since a parameter is represented by another functor, by the time a facet is looking at its parameters to look for names to add to the context, those parameters have already been evaluated into (monadic computations yielding) types, and the original information is gone.The type checker used a catamorphism initially, and in order to solve this problem, the type checker initially changed its carrier into a tuple containing both the resulting type as well as the re-constructed term. Since a catamorphism is a ground-level-up recursion scheme, we can utilize the exact same constructors that create terms to build a term with only its own constructor and the already complete sub-terms' constructed terms. In this way, we had an implementation of a paramorphism, which is more appropriate for the design of

the Rosetta language. It was functional, and certainly added more functionality to the language, but it was also a brittle approach. Every sub-term would bind to a tuple, and those terms had to be manually constructed at every single return statement.

This manual piping of values looks quite similar to the piping of values in non-monadic interpreters, and is a wonderful clue to the fact that it can be done in a better, cleaner style. Eventually, InterpreterLib could provide support for paramorphic recursion schemes by sequencing an identity algebra with the paramorphic algebra itself. Instead of finding the sub-terms' original structure by examining the tuples resulting from binding the sub-terms, we now have the entire term's original structure as a parameter to the phi function. We can now omit all code that simply passed through the structure, and use a carrier more appropriate to the task of type checking. Type checking does not return a pair of term and type, it returns a type. The capability to handle terms requiring a paramorphism should not require us to change the carrier, and the current approach achieves this. Utilizing the InterpreterLib implementation of a paramorphism gives an interpreter greater flexibility in a concise style of coding, without the pitfalls and fragility of manually providing such capabilities.

7.1.2 Constraints Collection and Unification

Constraint-based type checking [19] [6] [17] [20] is a method to allow the introduction of new type variables to define relationships between types without directly expressing what those types must be. The representation of a type by naming it allows the name to show up in multiple places, indicating that whatever type it is, it must be equitable in every place. Such a type is called a *universal*

type, and allows Rosetta to define a richer set of terms. For instance, we could define a list structure without defining what the contents of the list are, and thus create concatenation, head, tail, and many other functions over lists without requiring an implementation for lists of integers, and again for lists of booleans, and again for any other list needed. In order to represent universal quantifiers in type checking, we use type variables to abstract the actual type from a representation of what the type will be. In order to resolve type variables, we collect constraints on type variables and solve the set of constraints.

Unification (in the type checking realm) is the process of solving a set of constraints to determine a value for each type variable. By unifying a set of constraints, we can determine the exact type of a term, up to all constrained variables. This means that universals will still be unconstrained, but all basic types that are implied will be in place.

7.1.2.1 Constraint Sets

A constraint is either an equality requirement (type A is equivalent to type B) or a subtyping requirement (type A is a subtype of type B). A constraint set is simply a collection of these constraints. A constraint-based type checker will generate a constraint every time some assertion or contract must be obeyed. For instance, an if-expression generates two constraints: (i) the left and right branches must be of the same type; and (ii) the conditional must be a Boolean. Sometimes, no constraint needs to be generated. For example, we know that `True` is of type boolean, and nothing else needs to be checked. In general, constraints generated by an expression will be the union of the sub-expressions' constraints with any newly generated constraints. This process passes constraints up the

chain from subterms to the top-level terms, whether it is a single expression, an entire program, or perhaps a collection of Rosetta components.

7.1.2.2 Solving Constraints

Once the constraints of a term have been collected, we should have enough information to determine a type of the top-level term. If type variables still exist, the type variables represent universal quantifiers. We only want to solve the constraints at the top level term, as portions of the constraints may not be solvable on their own. We can solve a system of constraints by applying a few rules of reduction and substitution to remove all constraints: (i) we can remove all redundant constraints, such as "A is equivalent to A" and "B is a subtype of B"; and (ii) if a type variable is required to be equivalent to some type C, we can substitute C for that type variable throughout the entire set of constraints, often leading to more available rule applications. Consider the set of constraints:

```
C==Boolean, D==Boolean, C==D
```

We can consume the first requirement and replace all occurences of `C` with `Boolean` (via (ii)), to get:

```
Boolean==Boolean, D==Boolean, Boolean==D
```

and then remove the constraint `Boolean==Boolean` (via (i)). Similarly, we replace all occurences of `D` with `Boolean` (via (ii)), to get:

```
Boolean==Boolean, Boolean==Boolean
```

Thus we have redundant constraints that can be dropped (via (i)); with no constraints remaining, we have solved the set of constraints. As long as we also perform the replacements in the top-level term's indicated type, we will end up

with the actual type, solved over its constraints. Once no more constraints exist, the unification is complete. What remains is a listing of each type variable that may be included in the top-level term's type, and the corresponding type it represents. Type variables may still exist in the resulting type; this implies that they are universal quantifiers. If the constraint set could not be solved, the term is badly typed and we would not even consider the meaning of remaining type variables.

7.1.2.3 Integration with Existing Code

The first approach used to collect the constraints was to turn every algebra into one that returned a tuple containing the type of the term and the constraints generated. This necessitated manually pattern-matching the tuple, extracting the related sub-terms' constraints, combining them and returning them as part of the pair-result. While it was a relatively mechanical transition, it meant a lot of extra work and made the code brittle. At one point, the Rosetta type checker was returning a triple, consisting of the return type, the manually reconstructed original term, and the set of manually collected constraints.

Adding to this tuple for each significant change is a horrible way to manage code. These are very regular transformations that should be handled implicitly. The implementation of paramorphisms in InterpreterLib removed the necessity for manually reconstructing the original terms; what can be done about the constraints collection? The Writer monad is exactly what is needed. Constraints may only be added, never removed. Thus, we can continually add new constraints by telling the writer each time a new constraint should be generated. The Writer monad encapsulates all the necessary mechanisms for collecting the constraints,

and when we actually run the monads, we will get back the computed constraint-set, and can then perform unification. Unification won't be done until we've entirely type checked and constraint-generated the entire term anyways, so this information is available at the exact moment it is necessary. The unification algorithm (found in [19]) is run and type variables are then replaced throughout the result type. The exact list of type variables and solutions will be pivotal in resolving overloading in the future. This exact information is a by-product of the unification algorithm.

This was perhaps one of the largest changes to the entire body of code. Instead of having to specify all the rules that must be followed for a particular type to be valid for some particular term, we can simply relate what conditions must be valid. For instance, in an if-expression, we previously would have to ensure that the guard statement actually is a boolean, and we would have to figure out exactly what is the least super-type of both of the branches to determine the overall type before we can return what the type of the entire expression is. With a constraint-based approach, the added complexity of constraint-solving gives us the opportunity to simplify the typing process for many terms like if-expressions. Utilizing a constraint-based approach, we can simply add constraints that the guard must be a boolean, and create a new type variable used to state that both branches must be subtypes of the type variable. Also quite convenient is the fact that this type variable is exactly the type of the entire expression, and we have our result as well. This makes for a clean coding style, quite based on the type rule of a Rosetta if-expression:

$$\frac{\Gamma \vdash b : \mathit{boolean} \;\; \Gamma \vdash t : T \;\; \Gamma \vdash f : T}{\Gamma \vdash \textbf{if}\, b\, \textbf{then}\, t\, \textbf{else}\, f\, \textbf{end if} : T}$$

```
toRoExpr _ (RoIf b t f) = do
  b2 ← b
  t2 ← t
  f2 ← f
  s ← gensym
  let s' = mkRoTyVar $"if"++s
  tell [(b2, base "bool"), (t2,s'),(f2,s')]
  return $ s'
```

`RoIf` first binds all three sub-terms, and then generates a new type variable in the next two lines, named similar to `if0`, `if1`, or `if24`. `gensym` is a function utilizing the State monad that keeps a counter expressly for generating unique names. Next, `RoIf tells` the Writer monad all three constraints generated at this point–the guard must be a Boolean and both the true and false branches must be of the same type, represented by the type variable. Consider a more complex example:

```
toRoExpr _ (RoSet exprs) = do
  tys2 ← sequence exprs
  s ← gensym
  let s' = "set" ++ (show s)
  tell $ zip tys2 (repeat $ mkRoTyVar s' )
  return $ mkRoSetType $ mkRoTyVar s'
```

A Rosetta set is simply a homogeneous list. The generated constraints for such a set are that each element must be of the same type; therefore, we similarly create a new type variable and zip all the elements' types with that type variable, and tell the entire batch of constraints. To fully appreciate the cleanliness of utilizing the Writer monad, consider the previous alternative:

```
toRoExpr (RoIf b t f) = do
  (b2,bc) ← b
  (t2,tc) ← t
  (f2,fc) ← f
  s ← gensym
  let s' = mkRoTyVar $"if"++s
```

```
let constraints = bc++tc++fc++[(b2, base "bool"), (t2,s'),(f2,s')]
return $ (s',constraints)
```

The majority of the code for such a simple example is consumed with pattern-matching, combining, and re-tupling the constraints. The chance to identify the correct monad and rely on it to thread through the necessary information proves a vital means of maintaining concise code.

While there were some significant changes to migrate from the information necessary to decide a term's type without constraints to a system utilizing constraints, many terms of Rosetta were left alone. Specifically, any term that would not need to generate constraints of their own would not need any modification.

Overall, this is a more modular approach. Any particular algebra that does't need to generate any constraints doesn't need to know the other algebras are generating constraints via the Writer monad. They don't have to return pairs or tell any constraints, since the constraints portion would always be empty. The algebras that need to `tell` constraints are exactly the algebras that must know about the Writer monad, and so there is little chance of constraints-generation interfering with any other features' implementations. Indeed, if one algebra does not utilize the Writer monad, it cannot interact with it at all. This also gives more motivation for programming in a modular monadic style. Unification was originally intended for a recursive-descent style type checker, and the approach conceived for a more general monolithic style of type checking is still viable in a reasonable format, and we gain flexibility at the same time.

7.1.3 Modifications in Context

The context underwent its own migration, from a simple list of names paired with types, to one containing information on other modules, to the current system

where the representation is a symbol table. Changing the representation of context turns out to be a bit brittle, even with the current approach. As type checking of multiple modules that use each other becomes the next goal, the need for a more versatile context was pressing. The simple, flat list of context could not capture the full knowledge of discovered types without ignoring and violating possible duplicate names, unexposed parts of other design units, and indeed the scope of the language itself. For our purposes, a symbol table is a list of triplets of a name, a type, and another symbol table.

```
newtype SyTa = SyTa [(String, Lang, SyTa)]
```

This structure is able to more directly mirror the recursive nature of the scope of Rosetta, as opposed to some dynamic scope with ad-hoc rules to add and remove items to and from scope, as the context used to be:

```
type Env v = [(String,v)]
```

The change to a symbol table required the modification of all code that would add to the context, but was relatively isolated, as code that did not rely on the context was not required to pass through any new variables. Also, since the upgrade is from a state where only degenerate cases were possible before, the bulk of the transformation was a simple one, in which the third and new element of the new context entry would simply be an empty symbol table. In its current state, the context is much better in tune to the scope of the Rosetta language itself, and should aid in further reasoning of the Rosetta type system by more closely mapping reality, and not a series of code design decisions. One of the biggest lessons to learn from this is that an added layer of abstraction between using context and representation of context would be worth the effort–for instance,

creating a set of helper functions that would take the necessary information and interface with the context may have proved a successful solution. As it stands, the changes were not too severe, and changing the way some aspect of a language interfaces with the rest of the language is always going to inflict a wave of tweaks and changes.

7.1.4 Modifying the Value Space

The last major change has been to abandon the `Ty` value-space and commit to the true value-space for types of Rosetta terms themselves. Types are built into the language as terms, with their own functor in the non-recursive AST representation. Thus, logic to decide types by explicitly looking at sub-terms' types must now project the values from the fixed point of the sum into the type's functor in order to pattern match. This does not actually occur that often; as in the examples above, we do name the sub-terms' types and use them, but we do not actually pattern-match their exact structure. For instance, with the if-expression, we tell the constraint that `b` must be a `boolean`, but we do not loot at `b` and check if this is so. The sub-terms will also have to be projected if further inspection is necessary. This projecting was an annoyance, since it means the code could require a thorough overhaul despite its regularity of change, but it is somewhat avoidable. There is a function provided by InterpreterLib called `unsafePrjF` that will project from a fixed point of a sum of functors to the functor buried inside; it relies on the instances of the `SubType` class. Thus the paramorphism can provide the original term in its projected, exposed format, leading to little actual unpackaging of terms.

Delaying usage of the full Rosetta type functor to represent types was worth

the wait–early forays into the semantics of the language were not clouded by concerns with a relatively complex value-space structure. Furthermore, a lone session's work makes the current version legitimate for all future use, now that the Rosetta AST has largely completed its most significant metamorphosis. The motivation for this last change is to facilitate implementation of the reflective system in Rosetta. Rosetta code may contain descriptions of more Rosetta code, and the flow through the various stages of interpretation of such code relies on the ability to call upon parsing, type checking, and evaluation. Thus the type checker needs no modifications, and the logic for when to switch between type checking and anything else is added in an isolated fashion. Indeed, people are working on evaluation, type checking, and the reflection system respectively, and the evaluation code and type checking code have not needed any changes to their approach to be compatible with reflection. They merely need to function and return the types of values they ought to, such as elements of the Rosetta language instead of a special data structure to represent types in the type checker's case. As updates to the type checker or evaluator are checked into a repository, reflection can update to the most current implementations and continue to use them in a general enough way to allow its own progress, quite independent of the current issues in other analyses.

7.2 Implications of Alternative Approaches

We argue that writing the Rosetta type checker in the modular monadic style leads to more flexible code, but what alternatives are there, and how would they have fared under the same stresses of syntax element addition and removal, staged feature support, and so on? In this section, we compare the traditional recursive-

descent approach, writing a single type checking function. To be fair, we will utilize the recursive definition of terms for the internal representation, as this is the natural target for such an approach.

Initially, we must decide on a set of terms to support, and a value space for types. What is more, the term space must be agreed upon by anyone who wants to combine efforts–the parser and type checker would be worthless for evaluation if they did not operate on the same term space. This introduces a bit of artificial coordination between individual efforts. Our first approximation is in figure 7.1.

```
data Tm = TmTru | TmFls | TmBit Int | TmNat Int | TmInt Int

data Ty = TyBool | TyBit | TyNat | TyInt

typeof (TmTru) = TyBool
typeof (TmFls) = TyBool
typeof (TmBit x) = TyBit
typeof (TmNat x) = TyNat
typeof (TmInt x) = TyInt
```

Figure 7.1.

In the modular monadic style, the assumption is that there is one large AST in mind, defined as many separate functors. We can consider each functor as minimally some set of features we want to implement, and without changing the AST definition we merely include more of the functors in our sum. The AST definition would not change as a result of the current set of implemented features, it would only change if the AST were deemed insufficient or inappropriate by itself. In the current running example, we now want to add in the simply-typed lambda calculus. In order to maintain the proper bindings, we now need an environment. To save future effort later on, we realize that we should migrate to a monadic

version, utilizing only `Reader` at the moment. We will have to modify the old term definition, and the old type definition to result in the code of figure 7.2. To be fair, the type definition was modified at this point in the modular monadic approach in roughly the same fashion; the term space, however, was not modified by the decision to begin this phase, only by adding new constructs.

We add if-expressions to the language as well. This involves modifying the existing term space, as well as modifying the existing `typeof` code. We would need to add `TmIf Tm Tm Tm` to the definition of `Tm`, and add another pattern-match to the implementation of `typeof`:

```
typeof' (TmIf a b c) = do
    a' ← typeof' a
    b' ← typeof' b
    c' ← typeof' c
    if a' /= TyBool then error "must␣have␣a␣bool␣type␣guard."
        else if b' /= c' then error "must␣have␣matching␣branches"
        else return b'
```

Notice an emerging pattern–for each sub-term, we must call `typeof'` on each term and then bind the result to a variable name and then only use that name. With a catamorphic computation style, these sub-terms would all be monadic computations of the recursive call. This piping through of computations to yield their values is a simple task, but will always be necessary in this recursive-descent approach. If you find the need to change the name of the function, there will be quite a few re-namings to perform.

We need to consider the current flexibility to changes in our term-space. Are additions simple? What about removal or modification? Additions again imply modifying our term-space, a relatively simple task for now. Removing a term from the term-space may occur if the language under design ends up not needing

```
data Tm = TmTru | TmFls  | TmBit Int | TmNat Int | TmInt Int
    | TmLam String Ty Tm | TmVar String | TmApp Tm Tm

data Ty = TyBool | TyBit | TyNat | TyInt | TyMap Ty Ty
    deriving (Show, Eq)

type Ctxt = [(String, Ty)]

lkp x ((n,ty):ns) = if x==n then ty else lkp x ns
lkp x [] = error $ show x ++ "not␣found."
lookupEnv x = ask >>= λenv→ return $ lkp x env

typeof' :: (MonadReader Ctxt m) ⇒ Tm → m Ty
typeof' (TmTru)   = return TyBool
typeof' (TmFls)   = return TyBool
typeof' (TmBit x) = return TyBit
typeof' (TmNat x) = return TyNat
typeof' (TmInt x) = return TyInt
typeof' (TmVar x) = do
    ty ← lookupEnv x
    return ty
typeof' (TmLam x ty tm) = do
    env ← ask
    b   ← (local ∘ const ) ((x,ty):env) $ typeof' tm
    return $ TyMap ty b
typeof' (TmApp x y) = do
    env ← ask
    x' ← typeof' x
    y' ← typeof' y
    case x' of
      (TyMap a b) → if a==y' then return b else error $ "bad␣app␣types"
      _ → error $ "bad␣function␣arg␣to␣app"

typeof = flip runReader [] ∘ typeof'
```

Figure 7.2.

a term, or ends up providing something similar to deprecate a particular term, perhaps by elaboration. Removal is still somewhat local, but does involve editing the `typeof` function itself. The ability to avoid directly modifying the function is worth even more when we consider that others may be using a work-in-progress.

In supporting a new feature, the type checker may become unstable during the process. One solution for anyone who may rely on its stability is to simply not update to the current version until it is stable. But what if other minor bugs in relevant portions are found and fixed? Do we fix both the current type checker and the old version? This parallel implementation is obviously unfortunate and hopefully unnecessary. With the modular monadic approach, as long as the new feature is implemented with its own functor and algebra, we can simply omit the unstable or irrelevant features by removing the functor from the fixed point of the sum of functors and removing the phi function from the sum of algebras, respectively. This advantage exists no matter how large or small the target language is–the modular monadic solution will always be better prepared to present only portions of its current solution automatically. By presenting the different functors and algebras, different users can use the same type checker in different ways *simultaneously*. The recursive descent version presented thus far has a monolithic term space and a monolithic function for performing type checking, and cannot offer such an approach.

Another desired capability is to be able to re-write one phi function and use both versions in different applications. Perhaps we want to add sub-typing to the language, but the implementer of the evaluator does not currently want to handle sub-typing. If sub-types only play a semantic rôle in two of perhaps fourteen algebras, we can rewrite those two algebras to deny sub-typing semantics, and the

evaluator can continue development on its own track. Other uses of duplicate phi functions would be to implement two solutions to the same problem in different algebras, and run some test-bench on the results. The benefits incurred have been identified, and as such, InterpreterLib actually has a general means of achieving this–there is another algebra combinator, called `updateAlgebra`, which allows us to replace one phi function in an algebra simply by its type alone. Its usage is not required to discuss the Rosetta type checker, but its usefulness is yet another argument for using the modular monadic approach and the generic programming paradigms provided by InterpreterLib.

We also often want to re-use computation of one algebra with another. There is no support for sequencing type checking as a monolithic function into some other function. How can you get the results of each sub-term's type checking available at each sub-term for your new analysis? As information such as changes in context travels from terms to sub-terms, and as information such as constraints on the type travels upwards, there is no automatic mechanism for supplying this; type checking will have to be reconsidered in some fashion. With the modular monadic approach and the utilities of InterpreterLib, such a mechanism is present in `sequenceAlg`. Knowledge of the algebraic nature of the approach lends to more flexible usage.

Chapter 8

Conclusions, Future Work

8.1 Conclusions

The Rosetta Type Checker utilizes modular monadic semantics within the frame of InterpreterLib. It is a collection of separate algebras over the functors of the Rosetta language's internal non-recursive representation, and it uses a paramorphism to traverse the structure, maintaining context via the Reader monad, maintaining constraints on types via the Writer monad, and generating new type variables via the State monad. The type checker performs unification on the set of constraints generated to determine the type of the term. It handles basic elements and functions, individual design units and supports universal quantifiers. The nature of this approach enabled constant changes to the term space, the supported features, and even significant changes to the algorithmic approach in performing type checking, with manageable modifications. The result is a highly modular collection of algebras that can be re-used, sequenced, or updated in a variety of applications without actually requiring modifications to the type checker itself. The Rosetta type checker represents the benefits of the modular

monadic approach that InterpreterLib provides.

It takes time to get used to writing interpreters in a modular monadic style, but when writing code for a large enough system the benefits of modularity and flexibility favor this approach. The ability to change the underlying language with minimal interaction and minimal code-rewriting is an excellent goal. Furthermore, the ability to re-use code in many sequenced or qualified ways gives an unmistakable advantage. The ability to choose what functors to include, or how to sequence an algebra with an analysis without needing any modification to the original algebra gives the programmer a flexibility not available with a more traditional approach. Even the ability to change the algorithmic approach to one with minimal areas demanding consideration for rewriting makes the programming task tractable in a real-world environment. This style of programming is significantly adaptable; the current work was interpretation of a moving target to a moving target, and yet a thousand some lines of Haskell code was able to keep up. The theory behind the framework is sound and general enough to allow for significant maneuvering within that framework.

8.2 Future Work

Rosetta is defined as a dependently typed language, and the framework for a dependent language will likely induce some significant changes to the current state of the typechecker. Investigation into the properties of dependently typed languages in general is needed in order to ascertain what the Rosetta typechecker needs to fulfill the language's definition. It may be that only some subset of the possible dependent features will be necessary, or perhaps the dependent type system in Rosetta is unlike any before it. A dependently typed system implies

that some computation will be possible while calculating a type; it may be that an algebra can check for some known halting conditions that must always apply to the dependent aspects, and thus result in either a computable type or a type error. Obviously we cannot solve the general halting problem, but as McBride suggests [14], there are useful subsets of computations that can be proved to halt or not, based on structural induction. Failing such a possibility for Rosetta's dependent characteristics, the Rosetta type checker may need to require type ascriptions for dependent elements beyond structurally inductive ones, such that the task of inference is demoted to the task of verifying the ascribed type. If such an analysis proves possible, the Rosetta type checker can then simply run the analysis to ensure a term would halt, and then could utilize the Rosetta evaluator algebra. Assuming there is enough room for exploration here, the dependent type system of Rosetta will likely be the topic of my next thesis.

One limitation of the current typechecker is the inability to type check across modules. There is a very unrestrictive series of rules for where a package or component may reside. The inter-component dependencies must be sorted out prior to typechecking any module–use clauses must respect the export lists. In order to be able to add a package's exported parts to the context for another design unit, it must already have been sufficiently type checked to know what must be added. The root of the problem is that the task of creating the symbol table for an entire package containing items that refer to each other is limited by knowing the symbol table of all others to compute the symbol table of a particular design unit. In order to tie this particular recursive knot of symbol table dependencies, we can rely on Erkök's solution [5] to the repMin problem that Bird proposed in 1984. This has already been successfully implemented in

the Rosetta evaluator, and is a promising approach that may be applied in other instances as well. It essentially makes information available at each node that must first visit each node to calculate, and relies on the lazy evaluation nature of Haskell in its implementation via `mdo`. The technique might also be applied to provide the set of type variable substitutions, gained by constraint generation and solving, to each node for immediate replacement. This means that we may sequence the type checker with another algebra and get the actual type of each term, and not potentially some type variable that would later on be solved and given a substitution. The work on packages is likely the first and foremost goal of future work, as it opens up the largest set of Rosetta which would then be ready for type checking, and also is a relatively vital portion of the language by itself.

Appendix A

Type Checker Code

Portions of the code for the Rosetta type checker are presented here, to give an idea of the actual code. Too many other files are dependent on each other which are also under development, and so it would be pointless in trying to gain a snapshot of the entire system at this point in time; as such, only the relevant type checking files are presented.

A.1 TypeChecker/Alg.hs

This is the file in which the semantics are defined per algebra, one `phi` function per functor.

```
{-# OPTIONS -fglasgow-exts #-}
{-# OPTIONS -fno-monomorphism-restriction #-}
{-# OPTIONS -fallow-incoherent-instances #-}
{-# OPTIONS -fcontext-stack=40 #-}

module Rosetta.TypeChecker.Alg where

import Rosetta.Roast.NRast
import Rosetta.Roast.Common
```

```
import Rosetta.Roast.Lang

import Rosetta.TypeChecker.Common

import Regular.Functor
import Regular.Algebra
import Regular.Instances

import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Writer
import Control.Monad.Identity
import Maybe
import Monad

-- all the algebras
algRoConstructor a = mkAlg $ toRoConstructor a
algRoDataType    a = mkAlg $ toRoDataType    a
algRoDecl        a = mkAlg $ toRoDecl        a
algRoDeclValue   a = mkAlg $ toRoDeclValue   a
algRoDesignUnits a = mkAlg $ toRoDesignUnits a
algRoExpr        a = mkAlg $ toRoExpr        a
algRoImportSpec  a = mkAlg $ toRoImportSpec  a
algRoMode        a = mkAlg $ toRoMode        a
algRoObserver    a = mkAlg $ toRoObserver    a
algRoParameter   a = mkAlg $ toRoParameter   a
algRoQName       a = mkAlg $ toRoQName       a
algRoTerm        a = mkAlg $ toRoTerm        a
algRoType        a = mkAlg $ toRoType        a
algRoTypeVar     a = mkAlg $ toRoTypeVar     a


------------------------------------------------------------

toRoConstructor :: (MonadWriter Sigma m, MonadReader SyTa m,
    MonadState (Int,String) m)
    ⇒ Lang → RoConstructor (m TyLang) → m TyLang
toRoConstructor _ (RoConstructor n1 n2 observers) =
    fail ("RoConstructorNR␣would␣return␣TyFoo ○ ")

------------------------------------------------------------------

toRoDataType :: (MonadWriter Sigma m, MonadReader SyTa m,
    MonadState (Int,String) m)
```

```
    ⇒ Lang → RoDataType (m TyLang) → m TyLang
toRoDataType _ (RoDataType tyvars constructors) =
    fail ("RoDataType␣would␣return␣TyFoo ○ ")

-------------------------------------------------------------

toRoDecl :: (MonadWriter Sigma m, MonadReader SyTa m,
    MonadState (Int,String) m)
     ⇒  Lang → RoDecl (m TyLang) → m TyLang

toRoDecl _ (RoItem name ty declval) = do
    syta ← ask
    let (my_name, my_type,my_syta) =
      getSytaEntry ("item1␣" ++ name) name syta
    let syta' = conjoin2Sytas
        (addDecsFrom ("item2␣" ++ name) name syta) my_syta
    let syta2 = conjoin2Sytas
        (addDecsFrom ("item3␣" ++ name) my_name syta') my_syta
    (dv2) ← withEnv syta2 declval
    (ty2) ← withEnv syta2 ty
    case ((prjF ○ out ) dv2) of
        (Just (RoDepProduct dpd dpr)) → do
        tell $ [(dpr, ty2)]
        return ty2
        (Just a) → do
        tell $ [(pack a,ty2)]
        return ty2
        (Nothing) → error $ "couldn't␣see␣dv2,␣=␣:␣" ++ (show dv2)

toRoDecl self (RoAnnotatedDecl decl annots) = do
    syta ← ask
    (d2) ← decl
    return d2

-------------------------------------------------------------

toRoDeclValue :: (MonadWriter Sigma m, MonadReader SyTa m,
    MonadState (Int,String) m)
    ⇒  Lang → RoDeclValue (m TyLang) → m TyLang
toRoDeclValue self (RoFacet domain qvars params
       decls terms imports exports) = do
    syta ← ask
    dom2 ← domain
    q2 ← sequence qvars
```

```
    p2 ← sequence params
    d2 ← sequence decls
    --get their names
    let (pnames, qnames, dnames) =
      case unpack self of
        (RoFacet dom' q' p' d' t' i' e') →
          (getPnames p', getPnames q', getDnames d')

     --name all symbol-tablable things
     let allpars = q2 ++ p2 ++ d2
     let allnames = qnames ++ pnames ++ dnames
     let msg = "facet␣w/" ++ (concat (map (λa→a++",") dnames))
     let allsymtabs = (map (λa→ case (getSytaEntry msg a syta) of
          (a,b,c)→c) allnames)
     --add their stuff to the environment
     let env2 = conjoin2Sytas (extendSyTaList
          ((concat [qnames,pnames,dnames]))
          allpars
          (take (length allpars) (repeat emptySyta))
          syta
        ) syta
     d2 ← withEnv env2 $ sequence decls
     t2 ← withEnv env2 (sequence terms)
     i2 ← imports
     return $ dom2

toRoDeclValue self (RoFacetInterface domain qvars params
        decls imports exports) = do
    syta::(SyTa) ← ask
    dom2 ← domain
    let (qnames, pnames, dnames) = case unpack self of
        (RoFacetInterface dom' q' p' d' i' e') →
          (getPnames q', getPnames p', getDnames d')
    p2 ← sequence params
    let envs' = (map (λa → a syta)
        (map (addDecsFrom "") dnames)) :: [SyTa]
    let decsWithEnvs = (zip envs' decls)
    d2 ← sequence $ map (λ(a,b) → withEnv a b) decsWithEnvs
    i2 ← imports
    return $ dom2

toRoDeclValue self (RoFacetBody decls terms imports) = do
    syta ← ask
    let dnames = case unpack self of
```

```
        (RoFacetBody d' t' i')→(getDnames d')
    let envs' = (map (λa → a syta)
        (map (addDecsFrom "") dnames)) :: [SyTa]
    let decsWithEnvs = (zip envs' decls)
    d2 ← sequence $ map (λ(a,b) → withEnv a b) decsWithEnvs
    let env2 = extendSyTaList (dnames) d2 (getemptySytas (length d2)) syta
    t2 ← withEnv env2 (sequence terms)
    i2 ← imports
    if (boolsOrFacets d2)
      then fail ("Rofacetbody␣would␣parareturn␣TyFoo ∘ ")
      else fail "FacetBody:␣all␣declarations␣must␣be␣booleans␣or␣facets."

toRoDeclValue self (RoComponent domain qvars params
        decls asses reqs imps imports exports) = do
    syta ← ask
    dom2 ← domain
    p2 ← sequence params
    let (qnames,pnames,dnames) = case unpack self of
      (RoComponent dom' q' p' d' as' re' im' i' e')→
        (getPnames q', getPnames p',getDnames d')
    q2 ← sequence qvars
    d2 ← sequence decls
    let numentries = length qnames + length pnames + length dnames
    let env2 = extendSyTaList (concat [qnames,pnames,dnames])
        (concat [q2,p2,d2]) (getemptySytas numentries) syta
    a2  ← withEnv env2 $ sequence asses
    r2  ← withEnv env2 $ sequence reqs
    im2 ← withEnv env2 $ sequence imps
    i2  ← withEnv env2 imports
    return $ dom2

toRoDeclValue self (RoComponentInterface domain qvars params
        decls imports exports) = do
    dom2 ← domain
    p2 ← sequence params
    q2 ← sequence qvars
    d2 ← sequence decls
    i2 ← imports
    let (qnames,pnames,dnames) = case unpack self of
      (RoComponent dom' q' p' d' as' re' im' i' e')→
        (getPnames q', getPnames p',getDnames d')
    return dom2

toRoDeclValue self (RoComponentBody decls asses reqs implics imports) = do
```

```
    d2 ← sequence decls
    let dnames = case unpack self of
       (RoComponentBody d' as' re' im' i')→ getDnames d'
    a2 ← sequence asses
    r2 ← sequence reqs
    im2← sequence implics
    i2 ← imports
    if (boolsOrFacets (d2 ++ a2 ++ r2 ++ im2))
      then fail ("RoComponentBody would parareturn TyFoo ∘ ")
      else fail "ComponentBody: all decl's must be booleans or facets."

toRoDeclValue self (RoPackage domain params decls imports exports) = do
    syta ← ask
    dom2 ← withEnv syta domain
    p2 ← withEnv syta $ sequence params
    let (pnames,dnames,decltms) = case (unpack self) of
      (RoPackage dom' p' d' i' e')→ (getPnames p',getDnames d',d')
    let envs' = (map (λa → a syta)
        (map (addDecsFrom ("package w/"++(dnames!!0))) dnames)) :: [SyTa]
    let decsWithEnvs = (zip envs' decls)
    let computationsForEach = map (λ(a,b)→withEnv a b) decsWithEnvs
    d2 ← sequence computationsForEach
    i2 ← imports
    return dom2

toRoDeclValue self (RoPackageInterface domain  params
        decls imports exports) = do
    dom2 ← domain
    p2 ← sequence params
    let (pnames,dnames) = case unpack self of
      (RoPackageInterface dom' p' d' i' e')→ (getPnames p',getDnames d')
    d2 ← sequence decls
    i2 ← imports
    return $ dom2

toRoDeclValue self (RoPackageBody decls imports) = do
    d2 ← sequence decls
    let dnames = case unpack self of
       (RoPackageBody d' i')→ getDnames d'
    i2 ← imports
    if (boolsOrFacets d2)
       then fail ("RoPackageBody would parareturn TyFoo ∘ ")
       else fail ("PackageBody: all declarations must be booleans "++
           "or facets.")
```

```
toRoDeclValue self (RoDomain domain params decls terms
        imports exports) = do
    syta ← ask
    dom2 ← domain
    p2 ← sequence params
    d2 ← sequence decls
    let (pnames,dnames) = case (unpack self) of
      (RoDomain dom' p' d' t' i' e')→
        (getPnames p',getDnames d')
    let allpars = p2 ++ d2
    let env2 = extendSyTaList (concat[pnames,dnames])
        allpars (getemptySytas (length allpars)) syta
    t2 ← withEnv env2 (sequence terms)
    i2 ← imports
    return dom2

toRoDeclValue self (RoTranslator params expr) = do
    p' ← sequence params
    e' ← expr
    fail "not␣sure␣how␣to␣handle␣RoTranslators..."

toRoDeclValue self (RoFunctor params expr) = do
    p' ← sequence params
    e' ← expr
    fail "not␣sure␣how␣to␣handle␣RoFunctors..."

toRoDeclValue self (RoCombinator params expr) = do
    p' ← sequence params
    e' ← expr
    fail "not␣sure␣how␣to␣handle␣RoCombinators..."

toRoDeclValue self (RoData ty) = fail ("RoData␣would␣return␣TyFoo ∘ ")

toRoDeclValue self (RoValue expr) = do
    e2 ← expr
    return e2

------------------------------------------------------------

toRoDesignUnits  :: (MonadReader (SyTa) m)
            ⇒  Lang → RoDesignUnits (m TyLang) → m TyLang

toRoDesignUnits self (RoDesignUnits name units) = do
```

```
    let us = case (unpack self) of
                  (RoDesignUnits name' units') → units'
    unitys ← sequence units
    return $ mkRoDesUnits unitys

-----------------------------------------------------------------

toRoExpr  :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoExpr (m TyLang) → m TyLang
toRoExpr _ (RoVar qname) = qname
toRoExpr _ (RoLit lit) = do
    case (lit) of
     t@(RoChar       c ) → return (base "char")
     t@(RoBool       b ) → return (base "bool")
     t@(RoString     s ) → return (base "string")
     t@(RoNumberInf    ) → return (base "number")
     t@(RoNumberNegInf ) → return (base "number")
     t@(RoComplex  r i ) → return (base "complex")
     t@(RoImaginary  i ) → return (base "imaginary")
     t@(RoReal       r ) → return (base "real")
     t@(RoPosReal    r ) → return (base "posreal")
     t@(RoNegReal    r ) → return (base "negreal")
     t@(RoRational n d ) → return (base "rational")
     t@(RoInt        i ) | i == 1 || i == 0 → return (base "bit")
                         | otherwise        → return (base "integer")
     t@(RoPosInt     i ) → return (base "posint")
     t@(RoNegInt     i ) → return (base "negint")
     t@(RoNatural    n ) → return (base "natural")
     t@(RoBit        b ) → return (base "bit")

toRoExpr self (RoLambda  qvars  params range body) = do
    syta ← ask
    q2 ← sequence qvars
    p2 ← sequence params
    r2 ← range
    let allparams = q2 ++ p2
    let (qnames, pnames) = case unpack self of
      (RoLambda q' p' r' b')→ (getPnames q',getPnames p')
    let lengthsmatch = (length p2 == length pnames)
        && (length q2 == length qnames)
    if lengthsmatch
      then do
        let env2 = extendSyTaList (qnames ++ pnames)
```

```
            allparams (getemptySytas $ length allparams) syta
        b2 ← local (const env2) body
        if sto b2 r2
          then do
            if (length allparams > 1)
              then return $ mkRoDepProduct allparams b2
              else return $ mkRoDepProduct [head allparams] b2
          else fail $ concat ["Bad␣function␣range␣found;",
              "λnλnλtAscribed:␣", show r2,"λnλtFound:␣",
              show b2,"λn"]
      else fail $ concat ["Badly-formed␣NRAST:␣number␣",
          "of␣parameters␣given␣(", show (length p2),
          ")␣doesn't␣match␣number␣of␣parameter␣names␣(",
          show (length pnames),")␣given.λn"]

toRoExpr self (RoApp t1 params apptype ) = do
    depfunTy ← t1
    actuals2 ← sequence params
    -- make sure it's a mapping
    case ((prjF ∘ out) depfunTy) of
         (Just (RoDepProduct depparams deprange)) → do
             let constraints = zip actuals2 depparams

             --constraint-solving will ensure the types match.
             tell constraints
             if (length actuals2 < length depparams)
               then do
                   let leftovers = drop (length actuals2) depparams
                   return $ mkRoDepProduct leftovers deprange
               else if length actuals2 == length depparams
                    then return deprange
                    else fail (concat["Arguments␣to␣an␣app␣(including␣",
                        operations) do not match formal parameters.",
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣λnλtGiven:␣",(show actuals2),"λnλtExpected:␣",
                        (show depparams),"λnλn.(funTy:)",(show
                        depfunTy), "λnλn",(show $length actuals2),
                        "λnλn",(show $length depparams),"λnλn",
                        (show constraints)])
         (Just(RoTyVar tyvar)) → do
             rvar ← gensym
             let retvar = mkRoTyVar $ "somefunc"++(show rvar)
             tell $ [((mkRoTyVar tyvar),mkRoDepProduct actuals2 retvar)]
             return retvar
         (Just _) → do
```

```
            let (theappterm,theparams) = case unpack self of
                                           (RoApp x y z) → (x,y)
            actuals2 ← sequence params
            fail ("Tried␣to␣apply␣to␣a␣non-function;λnλnApp␣Term:"
                  ++(show theappterm)++"λnType:"++(show depfunTy)
                  ++"λnλnthe␣param(s):"++(show theparams)++"λnType:"
                  ++(show actuals2)
                 )
        (Nothing) → error
            $ "couldn't␣manage␣to␣unpack␣in␣an␣app;␣tried:␣"
            ++ (show depfunTy)
toRoExpr _ (RoLet decs body) =
           do d2 ← sequence  decs
              syta ← ask
              b2 ← withEnv syta body
              return b2

toRoExpr _ (RoIf b t f) = do
    b2 ← b
    t2 ← t
    f2 ← f
    let guardconstraint = if (istyvar b2)
        then [(b2,(base "bool"))] else []
    let branchconstraint = if (istyvar t2) || (istyvar f2)
        then [(t2,f2)] else []
    tell (guardconstraint++branchconstraint)
    let boolokay =  (not(istyvar b2) && (sto b2 (base "bool")))
        || (istyvar b2)
    let branchescheckable = ((not(istyvar t2)) && (not(istyvar f2)))
    if (not boolokay)
      then fail ("If-expression␣must␣have␣boolean␣guard␣"
          ++"statement,␣given␣type␣"++(show b2)++".")
      else if branchescheckable
             then if (sto t2 f2) then return f2 else
                if (sto f2 t2) then return t2 else
                   fail ("Branches␣of␣if-expression␣must␣match"
                      ++"␣types.␣(given␣"++(show t2)++",␣"
                      ++(show f2)++").")
             else do
                s ← gensym
                let branchty = mkRoTyVar $ "v"++(show s)
                return t2
toRoExpr _ (RoCase c alts) = do
    c2 ← c
```

```
    a2 ← sequence (map (λ(a,b)→a) alts)
    b2 ← sequence (map (λ(a,b)→b) alts)
    if (tysmatch ((mkRoSetType c2):a2)) && (tysmatch b2)
       then case (b2) of
                (b:bs)→ return b
                ([]) → fail ("TC:␣cannot␣have␣a␣case␣statement"
                             ++"␣with␣no␣alternates!")
       else fail ("Cases␣must␣match␣case_expr␣with␣alternatives'␣"
                  ++"types,␣and␣result␣types␣must␣match.")

toRoExpr _ (RoSet exprs) = do
    tys2 ← sequence exprs
    s ← gensym
    let s' = "set" ++ (show s)
    tell $ zip tys2 (repeat $ mkRoTyVar s' )
    return $ mkRoSetType $ mkRoTyVar s'

toRoExpr _ (RoSequence exprs) = do
    tys2 ← sequence exprs
    s ← gensym
    let s' = "seq" ++ (show s)
    tell $ zip tys2 $ repeat $ mkRoTyVar s'
    return $ mkRoSeqType $ mkRoTyVar s'

toRoExpr _ (RoQuant q params body) = do
    p2 ← sequence params
    b2 ← body
    if (sto b2 (base "bool"))
      then return (base "bool")
      else fail ("Quantified␣expressions␣must␣be␣boolean."
                 ++"λnλtFound␣as:␣"++(show b2))

toRoExpr _ (RoParen e) = do
    e2 ← e
    return e2

toRoExpr _ (RoTypeName ty) = do
    t2 ← ty
    return t2

toRoExpr _ (RoAnnotated e annots) = do
    e2 ← e
    return e2
```

```
toRoExpr _ (RoUnInterpreted) = return (base "uninterpreted")

-------------------------------------------------------------

toRoImportSpec :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoImportSpec (m TyLang) → m TyLang
toRoImportSpec _ (RoImport names) = do
    n2 ← sequence names
    return $ base "foo_toRoImportSpec␣incomplete"

-------------------------------------------------------------

toRoMode :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoMode (m TyLang) → m TyLang
toRoMode _ (RoParamMode a) = do
    a2 ← a
    return a2

toRoMode _ (NoMode) = return $ base "foo_toRoMode␣incomplete"

-------------------------------------------------------------

-- name is the observer's label, and expr is the type of the
-- projected piece.

toRoObserver :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoObserver (m TyLang) → m TyLang

toRoObserver _ (RoObserver name expr) =
    fail ("RoObserver␣would␣return␣TyFoo ∘ ")

-------------------------------------------------------------

toRoParameter :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoParameter (m TyLang) → m TyLang

toRoParameter _ (RoParameter name expr mode) = do
    e2 ←expr
    return e2
```

```
------------------------------------------------------------

toRoQName :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoQName (m TyLang) → m TyLang
toRoQName _ n@(QName listnames lastname) = do
    syta ← ask
    lookupEnv n syta

------------------------------------------------------------

toRoTerm :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoTerm (m TyLang) → m TyLang

toRoTerm _ (RoLabTerm name expr) = do
    expr

toRoTerm _ (RoUnlabTerm expr) = do
    expr

------------------------------------------------------------

toRoType :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoType (m TyLang) → m TyLang

toRoType _ (Type)           = return mkType
toRoType _ (RoBaseType s)   = case s of
    ("boolean")   → return (base "bool")
    ("char")      → return (base "char")
    ("string")    → return (base "string")
    ("element")   → return (base "element")
    ("complex")   → return (base "complex")
    ("imaginary") → return (base "imaginary")
    ("real")      → return (base "real")
    ("posreal")   → return (base "posreal")
    ("negreal")   → return (base "negreal")
    ("rational")  → return (base "rational")
    ("integer")   → return (base "integer")
    ("posint")    → return (base "posint")
    ("negint")    → return (base "negint")
    ("natural")   → return (base "natural")
    ("bit")       → return (base "bit")
```

```
    (_) → fail ("badly␣formed␣hack-type:␣"++s)

toRoType _ (RoSubtype ty)     = do ty2 ← ty; return $ mkRoSubtype ty2
toRoType _ (RoSetType ty)   = do ty2 ← ty; return $ mkRoSetType ty2
toRoType _ (RoArrayType ty) = do ty2 ← ty; return $ mkRoSeqType ty2
toRoType _ (RoDepProduct params b) = do
    p2 ← sequence params
    b2 ← b
    return $ mkRoDepProduct p2 b2
toRoType _ (RoTyVar name)   = do
    syta ← ask
    lookupEnv (QName [] name) syta
toRoType _ (RoSeqType ty)   = do
    ty2 ← ty
    return $ mkRoSeqType ty2


-------------------------------------------------------------

toRoTypeVar :: (MonadWriter Sigma m, MonadReader (SyTa) m,
    MonadState (Int,String) m)
    ⇒ Lang → RoTypeVar (m TyLang) → m TyLang
toRoTypeVar _ (RoTypeVar name expr) = do
    e2 ← expr
    return $ mkRoTyVar name

-- ═══════════════════════════════════════════════════════════
```

A.2 TypeChecker/Common.hs

This section provides the code of TypeChecker/Common.hs, which is where a large number of helper functions are created, both for the algebras, testing, and the overall combination of type checking into a single algebra. Some parts of this are not particularly relevant to the shown techniques of interpreter design, but they may be necessary to understand the code written with these helper functions and standard environment definitions.

```
{-# OPTIONS -fglasgow-exts #-}
```

```
{-# OPTIONS -fno-monomorphism-restriction #-}
{-# OPTIONS -fcontext-stack=40 #-}

module Rosetta.TypeChecker.Common where

import Rosetta.Roast.All
import Rosetta.Roast.Common

import Regular.Functor
import Regular.Algebra
import Regular.Instances
import MMS.HoFix

import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Writer
import Control.Monad.Identity

import Maybe
import Monad

----------------------------------------

--The TypeChecking Monad...
type TCMonad v  = StateT (Int,String)
                    (WriterT Sigma (ReaderT SyTa Identity))

runTCMonad {-evm-}  env x = runIdentity im
           where im = runReaderT rm env
                 rm = runWriterT wm
                 wm = runStateT x (0,"")

-------------------

newtype SyTa = SyTa [(String, Lang, (SyTa))]
emptySyta :: SyTa
emptySyta = SyTa []

--alias, so we know when it's just a type and when it's truly a Lang.
type TyLang = Lang

base s    = mkRoBaseType s
dp        = mkRoDepProduct
element   = base "element"
```

```
char      = base "char"
bool      = base "bool"
string    = base "string"
number    = base "number"
complex   = base "complex"
imaginary = base "imaginary"
real      = base "real"
posreal   = base "posreal"
negreal   = base "negreal"
rational  = base "rational"
int       = base "integer"
posint    = base "posint"
negint    = base "negint"
natural   = base "natural"
bit = base "bit"

type Env v = [(String,v)]
-------------------
-- constraint-based typing bits and bytes

type Constraint  = (Lang, Lang)
type Sigma = [Constraint]

unpack :: (SubFunctor f f1, Show (Fix f1)) ⇒ Fix f1 → f (Fix f1)
unpack a = case (prjF ∘ out) a of
              (Just x) → x
              (Nothing) → error $ "couldn't␣unpack1:λnλn␣" ++ (show a)


unpack2 :: (SubFunctor f f1, Show (Fix f1)) ⇒ Fix f1 → f (Fix f1)
unpack2 a = case (prjF ∘ out) a of
              (Just x) → x
              (Nothing) → error $ "couldn't␣unpack2:λnλn␣" ++ (show a)

unpackM a = do
            a' ← (prjF ∘ out ) a
            case a' of
              (Just x) → return x
              (Nothing) → error $ "couldn't␣unpack3:λnλn␣" ++ (show a)


pack = inn ∘ injF
packpair (x,y) = (pack x, pack y)
-------------------------------------------------------------
```

```
isty :: Lang → Bool
isty s = case unpack s of
         (Type) → True
         (RoBaseType _ ) → True
         (RoSubtype _ ) → True
         (RoSetType _ ) → True
         (RoSeqType _ ) → True
         (RoMultisetType _ ) → True
         (RoArrayType _ ) → True
         (RoDomainType _ ) → True
         (RoDepProduct _ _ ) → True
         (RoTyVar _ ) → True
         (RoDesUnits _ ) → True
         (a) → False

--formatshow :: String → SyTa → String
formatshow amt (SyTa xs) = concatMap (λ(x,y,z) → "λnλnλt(" ++ amt
    ++ (show x)++","++(show y)++","++(formatshow "λt" z)++")") xs

--formatshownames :: String → SyTa → String
formatshownames amt (SyTa xs) = concatMap (λ(x,y,z) → "λnλt" ++
    amt ++ "␣"++(show x)) xs

--formatshownamessmall :: SyTa → String
formatshownamessmall (SyTa xs) =
    concatMap (λ(x,y,z) → "λt"++(show x)) xs

--getSytaEntry :: String → SyTa → (String, Lang, SyTa)
getSytaEntry msg s syta@(SyTa xs) = do
    let x = lookup3 s syta
    case x of
     (Just a) → a
     (Nothing) → do
         error  $ "(via␣"++msg++"):␣tried␣to␣getSytaEntry␣for␣"
                  ++ (show s) ++",␣but␣it␣wasn't␣in␣the␣symbol␣table:"
                  ++ (formatshownamessmall syta)


addDecsFrom msg s sy@(SyTa xs) = case (getSytaEntry msg s sy) of
                                 (a,b,SyTa c) → SyTa $ c++xs


lookup3 :: String → SyTa → Maybe (String,TyLang,SyTa)
```

```
lookup3 a (SyTa []) = Nothing
lookup3 a (SyTa ((x,y,z):xs)) =
     if a==x then Just (x,y,z) else lookup3 a $ SyTa xs

lookupEnv :: (Monad m) ⇒ RoQName c → SyTa → m Lang
lookupEnv (QName [] name) symtab = lookupLocal name symtab
  where lookupLocal s l_env = case lookup3 s l_env of
     Just (_,ty,_) → return ty
     Nothing       → fail $ s ++ "␣not␣found."
lookupEnv (QName (x:xs) name) symtab =
    let (Just (_,_,xcomp)) = lookup3 x symtab
      in lookupEnv (QName xs name) xcomp

-----------
getemptySytas :: Int → [SyTa]
getemptySytas x = take x $ repeat emptySyta

extendSyTa :: (String,(Lang),SyTa) → SyTa → SyTa
extendSyTa next (SyTa xs) = SyTa $ next : xs

extendSyTaList :: [String] → [(Lang)] → [SyTa] → SyTa → SyTa
extendSyTaList strs tys sytas st =
    if length strs == length tys && length tys == length sytas
    then foldr extendSyTa st (zip3 strs tys sytas)
    else error $ "can't␣doit."

conjoin2Sytas (SyTa a)  (SyTa b)  = SyTa $ a++b

conjoinSytaList = foldr conjoin2Sytas emptySyta

withEnv:: (MonadReader (SyTa) m) ⇒ SyTa → m c → m c
withEnv = local ∘ const

-----------

initEnv = conjoin2Sytas stdEnv trusted_state_basedEnv

stdEnv = SyTa
    [("boolean",  tyBool      , emptySyta),
     ("char",     tyChar      , emptySyta),
     ("string",   tyString    , emptySyta),
     ("element",  tyElement   , emptySyta),
     ("complex",  tyComplex   , emptySyta),
     ("imaginary",tyImaginary , emptySyta),
```

```
    ("real",     tyReal      , emptySyta),
    ("posreal",  tyPosReal   , emptySyta),
    ("negeal",   tyNegReal   , emptySyta),
    ("rational", tyRational  , emptySyta),
    ("integer",  tyInt       , emptySyta),
    ("posint",   tyPosInt    , emptySyta),
    ("negint",   tyNegInt    , emptySyta),
    ("natural",  tyNatural   , emptySyta),
    ("bit",      tyBit       , emptySyta),
    ("+",  dp [int,int] int, emptySyta),
    ("-",  dp [int,int] int, emptySyta),
    ("*",  dp [int,int] int, emptySyta),
    ("=",  dp [element,element] bool, emptySyta),
    ("'",  dp [element] element, emptySyta),
    ("%",  dp [bit] bool, emptySyta),
    ("and",dp [bool, bool] bool, emptySyta),
    ("or", dp [bool, bool] bool, emptySyta),
    ("not",dp [element] element, emptySyta),
    (">",  dp [int,int] bool, emptySyta),
    ("<",  dp [int,int] bool, emptySyta),
    ("≥", dp [int,int] bool, emptySyta),
    ("=<", dp [int,int] bool, emptySyta),
    ("mod",dp [int,int] int, emptySyta),
    ("div",dp [int,int] int, emptySyta),
    ("event",dp [bit] bool, emptySyta),

--the domains should always be referrable, though their insides may not.
    ("logic",tyDomain "logic", emptySyta),
    ("static",tyDomain "static", emptySyta),
    ("state_based",tyDomain "state_based", emptySyta),
    ("finite_state",tyDomain "finite_state", emptySyta),
    ("infinite_state",tyDomain "infinite_state", emptySyta),
    ("discrete_temporal",tyDomain "discrete_temporal", emptySyta),
    ("discrete_time",tyDomain "discrete_time", emptySyta),
    ("continuous_temporal",tyDomain "continuous_temporal", emptySyta),
    ("continuous_time",tyDomain "continuous_time", emptySyta),
    ("frequency",tyDomain "frequency", emptySyta),
    ("signal_based",tyDomain "signal_based", emptySyta),
    ("process_based",tyDomain "process_based", emptySyta),
    ("CSP",tyDomain "CSP", emptySyta),
    ("pi_calculus",tyDomain "pi_calculus", emptySyta),
    ("discrete_event",tyDomain "discrete_event", emptySyta),
    ("tagged_event",tyDomain "tagged_event", emptySyta),
    ("trusted_state_based",tyDomain "trusted_state_based", emptySyta)
```

```
    ]

{- anything that is a standard part of a particular domain should be
   added to these domains in the appropriate [].  In the future, these
   will be used to extend the current environment with the standard
   things available for current typechecking.  I assume that'll be a
   slight change to lookup.
-}
nullEnv                = emptySyta
staticEnv              = conjoin2Sytas emptySyta nullEnv
state_basedEnv         = conjoin2Sytas emptySyta staticEnv
finite_stateEnv        = conjoin2Sytas emptySyta state_basedEnv
infinite_stateEnv      = conjoin2Sytas emptySyta state_basedEnv
discrete_temporalEnv   = conjoin2Sytas emptySyta infinite_stateEnv
discrete_timeEnv       = conjoin2Sytas emptySyta discrete_temporalEnv
continuous_temporalEnv = conjoin2Sytas emptySyta infinite_stateEnv
continuous_timeEnv     = conjoin2Sytas emptySyta continuous_temporalEnv
frequencyEnv           = conjoin2Sytas emptySyta continuous_temporalEnv
signal_basedEnv        = conjoin2Sytas emptySyta staticEnv
process_basedEnv       = conjoin2Sytas emptySyta signal_basedEnv
cspEnv                 = conjoin2Sytas emptySyta process_basedEnv
pi_calculusEnv         = conjoin2Sytas emptySyta process_basedEnv
trace_basedEnv         = conjoin2Sytas emptySyta signal_basedEnv
discrete_eventEnv      = conjoin2Sytas emptySyta trace_basedEnv
tagged_eventEnv        = conjoin2Sytas emptySyta trace_basedEnv
trusted_state_basedEnv = conjoin2Sytas
                         (SyTa [("confidentiality",tyBool, emptySyta),
                                ("integrity",tyBool, emptySyta),
                                ("availability",tyBool, emptySyta),
                                ("nonrepudiation",tyBool, emptySyta)
                               ]
                          ) state_basedEnv

----------------------------------------

gensym = do (s,str) ← get
            put (s+1,str)
            return (s+1)

traceMessage s  = do
    (a,b) ← get
    put (a, b ++ "λnλn" ++ s )

errorMessage :: (MonadState (Int, String) m) ⇒ String → m a
```

```
errorMessage (s::String) = do
    (a,b) ← get
    error $ s ++ "λnλnλn␣⟹␣trace:␣λnλnλt" ++ (show b)

-------------------------------------------------------------

data FacetPortion = FullFacet | Interface | Body deriving (Show, Eq)

tyElement     = mkRoBaseType "element"
tyBool        = mkRoBaseType "bool"
tyChar        = mkRoBaseType "char"
tyString      = mkRoBaseType "string"
tyNumber      = mkRoBaseType "number"
tyComplex     = mkRoBaseType "complex"
tyImaginary   = mkRoBaseType "imaginary"
tyReal        = mkRoBaseType "real"
tyPosReal     = mkRoBaseType "posreal"
tyNegReal     = mkRoBaseType "negreal"
tyRational    = mkRoBaseType "rational"
tyInt         = mkRoBaseType "integer"
tyPosInt      = mkRoBaseType "posint"
tyNegInt      = mkRoBaseType "negint"
tyNatural     = mkRoBaseType "natural"
tyBit         = mkRoBaseType "bit"
tyTop         = mkRoBaseType "top"
tyBottom      = mkRoBaseType "bottom"
tyUnit        = mkRoBaseType "unit"
tyFoo         = mkRoBaseType "foo"
tyType        = mkRoBaseType "type"
tySubtype  ty = mkRoSubtype ty
tySet ty      = mkRoSetType ty
tySeq ty      = mkRoSeqType ty
tyMultiset ty = mkRoMultisetType ty

tyRoDepProduct xs x = mkRoDepProduct xs x

tyVar nm = mkRoTyVar nm
tyDomain s = mkRoDomainType s

tyA t1 t2 = mkRoDepProduct [t1] t2

-------------------------------------------------------------

-- SUBTYPEOF, but really just for base types for now...
```

```
sto :: (Lang) → (Lang) → Bool
sto a b = if a==b then True else sto' a b

sto' a b = case (unpack2 a, unpack2 b) of
           (RoBaseType "bit", RoBaseType "natural") → True
           (RoBaseType "natural", RoBaseType "integer") → True
           (RoBaseType "negint", RoBaseType "integer") → True
           (RoBaseType "posint", RoBaseType "integer") → True
           (RoBaseType "integer", RoBaseType "rational") → True
           (RoBaseType "rational", RoBaseType "real") → True
           (RoBaseType "negreal", RoBaseType "real") → True
           (RoBaseType "posreal", RoBaseType "real") → True
           (RoBaseType "real", RoBaseType "complex") → True
           (RoBaseType "imaginary", RoBaseType "complex") → True
           (RoBaseType "complex", RoBaseType "number") → True
           (RoBaseType "number", RoBaseType "element") → True
           (RoBaseType "bool", RoBaseType "element") → True
           (RoBaseType "char", RoBaseType "element") → True
           (RoBaseType "bit", _ ) → sto (base "natural") b
           (RoBaseType "natural", _ ) → sto (base "integer") b
           (RoBaseType "negint", _ ) → sto (base "integer") b
           (RoBaseType "posint", _ ) → sto (base "integer") b
           (RoBaseType "integer", _ ) → sto (base "rational") b
           (RoBaseType "rational", _ ) → sto (base "real") b
           (RoBaseType "negreal", _ ) → sto (base "real") b
           (RoBaseType "posreal", _ ) → sto (base "real") b
           (RoBaseType "real", _ ) → sto (base "complex") b
           (RoBaseType "imaginary", _ ) → sto (base "complex") b
           (RoBaseType "complex", _ ) → sto (base "number") b
           (RoBaseType "number", _ ) → sto (base "element") b
           (RoBaseType "bool", _ ) → sto (base "element") b
           (RoBaseType "char", _ ) → sto (base "element") b
           (_, RoBaseType "top") → True
           (RoBaseType "bottom",_) → True
           (x,y) → False

-- subdomains... a record of the domain semi-lattice.
sdo a b = if a==b then True else sdo' a b
sdo' (RoDomainType "static")
     (RoDomainType "null") = True
sdo' (RoDomainType "state_based")
     (RoDomainType "static") = True
sdo' (RoDomainType "finite_state")
     (RoDomainType "state_based") = True
```

```
sdo' (RoDomainType "infinite_state")
     (RoDomainType "state_based") = True
sdo' (RoDomainType "discrete_temporal")
     (RoDomainType "infinite_state") = True
sdo' (RoDomainType "discrete_time")
     (RoDomainType "discrete_temporal") = True
sdo' (RoDomainType "continuous_temporal")
     (RoDomainType "infinite_state") = True
sdo' (RoDomainType "continuous_time")
     (RoDomainType "continuous_temporal") = True
sdo' (RoDomainType "frequency")
     (RoDomainType "continuous_temporal") = True
sdo' (RoDomainType "signal_based")
     (RoDomainType "static") = True
sdo' (RoDomainType "process_based")
     (RoDomainType "signal_based") = True
sdo' (RoDomainType "CSP")
     (RoDomainType "process_based") = True
sdo' (RoDomainType "pi_calculus")
     (RoDomainType "process_based") = True
sdo' (RoDomainType "trace_based")
     (RoDomainType "signal_based") = True
sdo' (RoDomainType "discrete_event")
     (RoDomainType "trace_based") = True
sdo' (RoDomainType "tagged_event")
     (RoDomainType "trace_based") = True
sdo' (RoDomainType "trusted_state_based")
     (RoDomainType "trusted_state_based") = True
-----
sdo' (RoDomainType "static") x =
    sdo (RoDomainType "null") x
sdo' (RoDomainType "state_based")  x =
    sdo (RoDomainType "static") x
sdo' (RoDomainType "finite_state") x =
    sdo (RoDomainType "state_based") x
sdo' (RoDomainType "infinite_state") x =
    sdo (RoDomainType "state_based") x
sdo' (RoDomainType "discrete_temporal") x =
    sdo (RoDomainType "infinite_state") x
sdo' (RoDomainType "discrete_time") x =
    sdo (RoDomainType "discrete_temporal") x
sdo' (RoDomainType "continuous_temporal") x =
    sdo (RoDomainType "infinite_state") x
sdo' (RoDomainType "continuous_time")     x =
```

```
    sdo (RoDomainType "continuous_temporal") x
sdo' (RoDomainType "frequency")           x =
    sdo (RoDomainType "continuous_temporal") x
sdo' (RoDomainType "signal_based")        x =
    sdo (RoDomainType "static") x
sdo' (RoDomainType "process_based")       x =
    sdo (RoDomainType "signal_based") x
sdo' (RoDomainType "CSP")                 x =
    sdo (RoDomainType "process_based") x
sdo' (RoDomainType "pi_calculus")         x =
    sdo (RoDomainType "process_based") x
sdo' (RoDomainType "trace_based")         x =
    sdo (RoDomainType "signal_based") x
sdo' (RoDomainType "discrete_event")      x =
    sdo (RoDomainType "trace_based") x
sdo' (RoDomainType "tagged_event")        x =
    sdo (RoDomainType "trace_based") x
sdo' (RoDomainType "trusted_state_based") x =
    sdo (RoDomainType "state_based") x

-----
sdo' _ (RoDomainType "null") = True
sdo' x y = x==y

-------------------------------------------------------------

tysmatch (x:(y:xs)) = if x==y then tysmatch (y:xs) else False
tysmatch (x:[]) = True
tysmatch ([]) = True

unListing x = case x of
                    (RoDepProduct xs x) → xs

-------------------------------------------------------------

isDomain x = case unpack2 x of
                   (RoDomainType s) → True
                   (_) → False

-------------------------------------------------------------

boolsOrFacets (x:xs) = case (unpack2 x) of
                          (RoBaseType "bool") → boolsOrFacets xs
                          (a) → False
```

```
boolsOrFacets ([]) = True

-------------------------------------------------------------

getPnames p =  map (λa → case (unpack2 a) of
                            (RoParameter x y z) → x ::String
                  ) p

getDnames :: [Lang] → [String]
getDnames d =  map getDnames' d
    where
 getDnames' a = case (unpack2 a) of
               (RoItem x y z) → x :: String
               (RoAnnotatedDecl dec annots) → getDnames' dec
               (a) → error $ show a

-------------------------------------------------------------

--for replacing a tyVar throughout a type.
searchAndReplace x name rep = case unpack2 x of
   (RoSubtype ty) → mkRoSubtype (searchAndReplace ty name rep)
   (RoSetType ty) → tySet (searchAndReplace ty name rep)
   (RoSeqType ty)→  tySeq (searchAndReplace ty name rep)
   (RoMultisetType ty)→ tyMultiset (searchAndReplace ty name rep)
   (RoDepProduct ys y) → tyRoDepProduct
     (map (λa → searchAndReplace a name rep) ys)
        (searchAndReplace y name rep)
   t@(RoTyVar label) → if label==name then rep else pack t
   (otherstuff) → pack otherstuff

-- --------

mapty f x = case unpack2 x of
            (RoSubtype a) → tySubtype  (f a)
            (RoSetType a) → tySet      (f a)
            (RoSeqType a) → tySeq      (f a)
            (RoMultisetType a) → tyMultiset (f a)
            (RoDepProduct xs x) → mkRoDepProduct (map (mapty f) xs) (f x)
            (z) → pack z

-- --------

dregty ::  ((Lang)  → a) → (Lang)  → [a]
dregty f x = case unpack2 x of
```

```
            (RoDepProduct xs x) →  concat $ (map (dregty f) xs)++[[f x]]
            (RoSubtype a)       → dregty f a
            (RoSetType a)       → dregty f a
            (RoSeqType a)       → dregty f a
            (RoMultisetType a)  → dregty f a
            otherwise           → [f x]

-- --------

--replace :: (Lang, Lang) → Lang → Lang
replace this@(s,ty) v = case unpack2 v of
                  (Type) → pack Type
                  a@(RoBaseType str) → v
                  a@(RoSubtype exp) → mkRoSubtype $ replace this exp
                  a@(RoSetType exp) → mkRoSetType $ replace this exp
                  a@(RoSeqType exp) → mkRoSeqType $ replace this exp
                  a@(RoMultisetType exp) →
                     mkRoMultisetType $ replace this exp
                  a@(RoArrayType exp) → mkRoArrayType $ replace this exp
                  a@(RoDomainType str) → v
                  a@(RoDepProduct xs x) →
                    dp (map (replace this) xs) (replace this x)
                  (RoTyVar n) → if s==v then ty else v
                  (RoDesUnits xs) → mkRoDesUnits $ map (replace this) xs
                  a → error $ "not␣a␣good␣type␣to␣replace␣"
                  ++ "through!λnλt␣" ++ (show a)

-- --------

nulltys = [tyElement,tyBool, tyChar, tyString, tyNumber,
           tyComplex, tyImaginary, tyPosReal, tyNegReal, tyRational,
           tyInt, tyPosInt, tyNegInt, tyNatural,  tyBit, tyTop,
           tyBottom, tyUnit, tyFoo]

isnullary x
    | foldl (||) False (map (λy→ y==x)nulltys)
       = True
    | otherwise = case unpack2 x of
          (RoTyVar _) → True
          (RoDomainType s) → True
          (_) → False

-- --------
```

```
notIn :: (Lang)  → (Lang)  → Bool
notIn x y = case (unpack2 x,unpack2 y) of
            (RoTyVar s, RoTyVar t) → s/=t
            (RoTyVar s, y') | isnullary y → True
            (RoTyVar s, y') → foldl (||) False (dregty (notIn x) $ y)

-- --------

genvar s = do
     fv ← gensym s
     return (tyVar fv)


istyvar x = case unpack2 x of
            (RoTyVar _) → True
            (_) → False

unisub :: (Lang,Lang) → Sigma  → Sigma
unisub (s,t) tys =
                     let ls = map (replace (s,t)) (map fst tys) in
                     let rs = map (replace (s,t)) (map snd tys) in
                         zip ls rs

unify :: Sigma  → Sigma  → Sigma
unify v@((s,t):cs) ans =
                   if (sto s t || sto t s) then (unify cs ans) else
                   if (istyvar s) && (s `notIn` t)
                   then if (istyvar t)
                        then (unify
                              (unisub (s,t) cs)
                              (((s,t)):(unisub ( (s,t)) ans))
                             )
                        else (unify
                              (unisub (s,t) cs)
                              (unisub (s,t) ans))
                   else if (istyvar t) && (t `notIn` s)
                        then if (istyvar s)
                             then (unify
                              (unisub (t,s) cs)
                              (unisub (t,s) ans)
                                  )
                             else (unify
                              (unisub (t,s) cs)
                              ((t,s):(unisub (t,s) ans))
```

```
                        )
                    else case (unpack2 s,unpack2 t) of
                    (RoSubtype s, RoSubtype t) →
                     unify ((s,t):cs) ans
                    (RoSetType s, RoSetType t) →
                     unify ((s,t):cs) ans
                    (RoSeqType s, RoSeqType t) →
                     unify ((s,t):cs) ans
                    (RoMultisetType s, RoMultisetType t) →
                     unify ((s,t):cs) ans
                    (RoDepProduct xs x, RoDepProduct ys y) →
                     unify ((zip xs ys)++[(x,y)]++cs) ans
                    (dom, RoDepProduct xs x) →
                        unify ((pack dom, x):cs) ans
                    (RoBaseType "uninterpreted",a) → unify cs ans
                    (_) → error ("couldn't unify... "++(show v))
unify [] ans = ans
```

A.3 TypeChecker.hs

This file takes the algebras written and puts them all into one algebra; then, it defines the paramorphism over that algebra, and yields a simple term-to-type function, `typeof initEnv`.

```
{-# OPTIONS -fglasgow-exts #-}
{-# OPTIONS -fno-monomorphism-restriction #-}
{-# OPTIONS -fcontext-stack=50 #-}
{-# OPTIONS -fallow-incoherent-instances #-}

module Rosetta.TypeChecker where

import Rosetta.Roast.Common
import Rosetta.Roast.All

import Rosetta.TypeChecker.Common
import Rosetta.TypeChecker.Alg
import Rosetta.TypeChecker.GenSymbolTable
import Rosetta.TypeChecker.Tests
```

```
import InterpreterLib

import Regular.Functor
import Regular.Algebra

import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Writer
import Control.Monad.Identity
import Maybe
import Monad

------------------------------------------------------------

-- NOTE: 'Lang' is in AST_Lang.hs, and always alphabetized.

alg' p =
         ( algRoConstructor p ) @+@
         ( algRoDataType    p ) @+@
         ( algRoDecl        p ) @+@
         ( algRoDeclValue   p ) @+@
         ( algRoDesignUnits p ) @+@
         ( algRoExpr        p ) @+@
         ( algRoImportSpec  p ) @+@
         ( algRoMode        p ) @+@
         ( algRoObserver    p ) @+@
         ( algRoParameter   p ) @+@
         ( algRoQName       p ) @+@
         ( algRoTerm        p ) @+@
         ( algRoType        p ) @+@
         ( algRoTypeVar     p ) @+@
         ( funitAlg           )


to' :: (SequenceAlgebra LangS Lang (m TyLang)
             (SequenceAlgebraT LangS Lang (m TyLang) Identity),
        MonadReader SyTa m,
        MonadWriter Sigma m,
        MonadState (Int,String) m) ⇒
    Lang → m TyLang
to' = para (λself → alg' (inn self) )

{--when you  want to add to the context, or want perusal to be able to
  pipe in its efforts... --}
```

```
to:: SyTa → Lang → TyLang
to syta s = let ctxt = (conjoin2Sytas syta initEnv ) in
      case (runTCMonad ctxt (to' s)) of
            a@(((ty),st),sigwriter) →
              foldr (λx→ replace x) ty (unify sigwriter [])

to2 s = case (runTCMonad initEnv $ to' s) of
            a@(((ty),st),sigwriter) →
              foldr (λx→ replace x) ty (unify sigwriter [])

to3 s = s

typeof = to

constrs syta s =
      let ctxt = (conjoin2Sytas syta emptySyta {--initEnv--}) in ctxt

constraints = constrs

-- stuff to run the tests ◦ ..
test = map (to emptySyta) t == answers
runem  = zip (theints 1) (map (to initEnv) t)
results = zip answers $ map (to initEnv) t
resultsbynumber =  zip (theints 1)
    $ map (λ(a,b) → (a::Lang)==(b::Lang)) results
failures = filter (λ(a,b)→ b==False) resultsbynumber

theints x = x : (theints (x+1))
```

References

[1] The glasgow haskell compiler.

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

[3] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

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

[5] L. Erkok. Solving the repmin problem with io.

[6] J. R. Hindley. The principal type-scheme of an object in combinatory logic. In *Transactions of the American Mathematical Society*, volume 46, pages 29–60, 1969.

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

[8] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27, 1992.

[9] 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.

[10] M. P. Jones and L. Duponcheel. Composing monads. Research report YALEU/DCS/RR-1004, Yale University, Yale University, New Haven, Connecticut, Dec 1993.

[11] S. P. Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. Tutorial at Marktoberdorf Summer School, 2000., March 2002.

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

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

[14] C. McBride. Faking it—simulating dependent types in haskell, 2001.

[15] L. Meertens. Category theore for program construction by calculation. CWI, Amsterdam and Department of Computing Science, Utrecht University, September 1995.

[16] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[17] R. Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, volume 17, pages 348–375, 1978.

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

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

[20] J. A. Robinson. Computational logic: The unification computation. In *Machine Intelligence*, volume 6, pages 63–72, 1971.

[21] J.-W. Roorda. Pure type systems for functional programming. Master's thesis, University of Utrect, 2000.

[22] M. Snyder. Rosetta type checker. Only the code relevant to type checking for Mark's master's thesis., 2005-2007.

[23] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[24] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

[25] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[26] 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.

[27] P. Weaver, G. Kimmell, N. Frisby, and P. Alexander. Software engineering with semantic algebras. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, 2007.