HASKELL-RL
AN EQUATIONAL SPECIFICATION OF HASKELL IN MAUDE

BY

ANDREW DOUGLAS BENNETT

B.S., University of Illinois at Urbana-Champaign, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

This paper serves to culminate work done on specifying a lazy language, Haskell, within a rewriting logic framework such as Maude. The features and capabilites of the resultant language, Haskell-RL, are similar in spirit to the base language Haskell. This writing shall also serve as a manual for the specification we have created. Performance runs of Haskell-RL show that the specification is very powerful, as it can produce similar results to other languages implemented in Maude.

Acknowledgments

I would like to thank my advisor, Professor Grigore Roşu, for introducing me to Programming Language design, and for guiding my research that is summarized here.

I would also like to thank Mark Hills for his support and helpful comments while working on Haskell-RL.

Table of Contents

List of Figures and Tables

Chapter 1

Introduction

Combining programming language specification with an equational rewriting system provides an immediate working implementation of that language. We have started with the features and their syntax within Haskell-98 [1], and attempted to mirror them into a Maude [2] specification, forming a language which has the same properties, but slightly different syntax: Haskell-RL.

1.1 Programming Languages

Most of the time, a computer scientist will deal with a programming language at a higher level, writing programs that follow the defined syntax, in the hope of solving a problem. Other scientists work at a lower level, optimizing compilation for size, runtime, or other variables. It is also possible, to extract certain features of a set of languages and redesign them into a new language, perhaps one that has features that no other language has. But first, one should start by defining a single language, with a single set of features.

Call-by-value, call-by-reference, call-by-need. Static method dispatch, dynamic method dispatch. Multiple inheritance. Higher-order functions. These are features that may or may not exist in a programmer's language of choice. Given the source code of a programming language (usually in a lower-level language like C or assembly), one should be able to alter it to add any feature he desires. The status quo, however, does not easily allow for that.

In contrast, a specification of a programming language in Maude is easily understandable, and intuitive to change, either for changing existing features or adding additional ones.

1.2 Lazy Evaluation

Lazy evaluation provides a methodology for *if* and *when* to evaluate some expression within the program code of a language. In Figure 1.1, `f` is nothing more than a project-first function; that is, the entire expression would evaluate to `Exp1`. Without lazy evaluation, `a` and `b` are processed until they are in some normal form (usually a simple data type). If in the process of simplifying `b` we reach an infinite loop or recursion, or

```
let f x y = x
    a     = Exp1
    b     = Exp2
 in f a b
```

Figure 1.1: Example of Lazy Evaluation

```
let f 0 _ = 0
    f _ x = x
    a     = 0
    b     = 3 / a
 in f a b
```

Figure 1.2: Lazy Evaluation avoids the Exception

cause an exception, the entire expression will never be given a value. Lazy evaluation will freeze both `a` and `b` (as well as `f`) along with the current environment, and only evaluate them by need.

In a more complex example, such as in Figure 1.2, fully evaluating `b` would have caused a divide-by-zero exception; lazy evaluation would avoid this exception.

Lazy evaluation is also used in Haskell-RL to delay evaluation of recursive data types. For example, there is no need to evaluate completely any left subtree in a binary search tree if you desire to know the greatest value stored therein. To know the length of a list, you don't need to expand out a cell, you just need to know that one is present there.

1.3 Haskell

Haskell is a typed, fully lazy, functional language. Functions are values, and for the most part they cannot modify existing state. Haskell is the best example of a lazy language, which is why it was the subject of our research. The importance of a typed language is mostly for static verification, and sometimes more efficient compilation. We did not implement this part of the language, and will discuss why in Section 4.7.

Use of Haskell is usually limited to research, as most programmers have an easier time designing procedural code. Furthermore, functional language programmers usually ignore the benefits of Haskell, claiming the overhead needed to do proper lazy evaluation takes more processor cycles and memory space than would be taken by non-lazy evaluation of their programs.

Nevertheless, in a purely theoretic view of functional programming languages, Haskell should be listed at the top. The vulnerabilities most languages suffer from are completely negated by lazy evaluation. Also, Haskell allows for the ability to do much static verification using the polymorphic type system.

```
function Fib(x : integer) : integer;
begin
  if x = 0 then
    Fib = 0
  else
    if x = 1 then
      Fib = 1
    else
      Fib = Fib(x - 1) + Fib(x - 2)
end
```

Figure 1.3: Traditional Approach to Finding Fibonacci Numbers

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

Figure 1.4: Equational Approach to Finding Fibonacci Numbers

1.4 Rewriting Logic

Traditionally, to determine the solution to a problem, one writes a program that has functions and variables, and makes use of these (perhaps recursively) to solve one instance of the problem. In Figure 1.3, we can see a simple algorithm for computing a particular Fibonacci number: very procedural, conditions to check explicitly. This code must still be lexed and parsed, and then it can be evaluated.

Contrast this with Figure 1.4. The main difference is simplicity. Instead of coding *what* and *how*, we just need to provide *what*. The left side of those equations, when matched with the current top-level state, are simply transformed into the right side. There is no need to translate the definition into any other form; we can do simple pattern matching to determine which equation we should use, doing a top to bottom traversal of all the equations.

Rewriting logic can solve any problem that can be expressed equationally. Let's play a game [3]. We start with a big bag of black and white balls. Multiple people can draw blindly from the bag two balls at a time. If they match in color, one black ball is placed into the bag; otherwise, a white ball is placed into the bag. Figure 1.5 provides the simplest complete solution. Only one equation is needed to reduce any input

```
fmod BALL is
  sorts Ball Balls .  subsort Ball < Balls .
  ops white black : -> Ball .
  op empty : -> Balls .
  op __ : Balls Balls -> Balls [assoc comm id: empty] .

  vars X Y : Ball .
  eq X Y = if X == Y then black else white fi .
endfm
```

Figure 1.5: Simple Equational Solution to the Ball Game

to a single `Ball`. The rewriting step will match two `Balls`, naming them `X` and `Y`, check them for equality, and replace into the set of `Balls` a properly colored `Ball`.

1.5 Maude

Maude provides the rewriting logic framework necessary to use a system of equations to reduce an expression. Maude keeps track of some current expression, and can match any pieces appearing at the same level to the left hand side of some equation that has been defined, replacing that match by what appears on the right hand side. Maude is the rewriting engine.

Expanding on this, Maude can show a trace of execution, so it can be seen exactly how some input is reduced into a simplified output. Specified correctly, Maude can take the syntax and semantics of a programming language, along with a program, and reduce it step by step to reach the final result of that program. An evaluation of a program is merely an equational transformation from the program text to the result; a program is mathematically equivalent to its output.

By turning particular equations into rules, Maude becomes a state machine, with the same properties as before. We can now view program evaluation as some input to a state machine in a start state that eventually reaches a final state by following certain transitions. Furthermore, we can tell Maude to search this state space for all possibilities of traversal, allowing us to search for particular orders of executions, or more exemplary, deadlocks.

Chapter 2

Programming Language State

The state of a language is the collection of attributes of that language, such as the environment, which provides a mapping from variable names to memory locations; the memory, which provides a mapping from locations to values; a list of running threads; or just the program text itself. We need to keep track of all of this information in order for the language to work properly. In a purely functional language, we may not need to keep around all of these state values, because the state is contained within the program itself. The program, as written, has some value, and the more of the program we compute, the more concrete that value becomes. We would not need to keep around memory because the values that we care about are in the current context. However, if we wanted to do anything more interesting, such as introducing threads or synchronization, additional state infrastructure would become necessary.

2.1 Standard Implementation

For any normal specification in Maude, we first design the environment and the store (memory). The environment keeps track of name/location pairs, and the store holds location/value pairs. So in any context, if we want to lookup the value of a particular variable, we need to match that name in the environment with a location, and then find that location in the store. We will demonstrate a memory lookup for Haskell-RL in Section 4.1.

If we have an object-oriented language, we would need to keep track of a set of classes, as well as a place to store our objects (if we don't keep them in the store). If we want to simulate I/O, we need two buffers, input and output, that we can match against in order to read data in, or print data out. Finally, if we wanted to do multithreading, we need to keep track of multiple copies of any thread-specific information, separate from the global information. For example, if we wanted a shared memory-implementation, our memory would need to exist at the top level, and our threads at a lower level, separated from each other. We could still use the memory at a higher level, but would need to match against more of the current state to get to that.

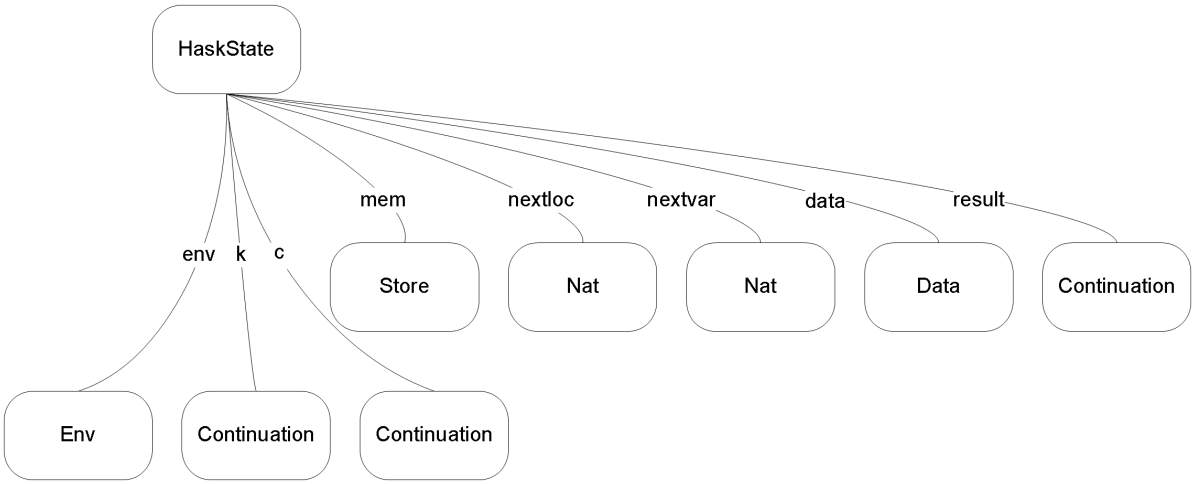

Figure 2.1: Haskell-RL State Infrastructure

2.2 Haskell-RL State

Specifically, for Haskell-RL, the state is designed as shown in Figure 2.1 (also shown as Maude code in the Appendix, module `HASK-STATE`). We create a soup of state attributes, that can flow around and be matched in any quantity and in any order. That is, if we need to access the memory and the environment, we would only include `env` and `mem` in our equation, without mentioning any of the other attributes.

The k-continuation is where the actual evaluation is performed. We would start with some expression which represents the entire program, and break off pieces at a time, choosing to evaluate one now, and sending the resultant value to what remains of the expression. More concrete examples appear in Chapter 4.

If Haskell-RL was multithread-capable, the attributes `env`, `k`, and `c` would need to be stored per thread, while the rest are per execution. The c-continuation stores information about the current case clause, and is used extensively for pattern matching.

The `result` continuation is used for output streaming. Haskell normally can output infinite data types, because there is some module requesting the next element in that structure. Because of the nature of Maude, data types must be finite (however, we can still work lazily on infinite types, just not display them). So, before Maude returns control to the terminal, we need to put the final output into `result`. If we had not done this, the only output would be the lazy result of a data structure; more examples of this are available in Sections 3.2 and 4.6.

To support fully lazy evaluation, we must encapsulate expressions that should not be evaluated into

some structure. We have chosen a `frozen` operator, which takes in the expression, the environment that expression should later be evaluated in, and the location we are storing it in memory. Whenever we get to an expression formed like in Figure 1.2, we perform a few steps. First, we add all of the names to our environment, then freeze all right hand sides along with that new environment. Whenever we refer to a name that is frozen, we pull it from the store, evaluate it a single step, and save it back to the location it came from.

Memory usually would store only concrete values, but because we wish to support this delayed evaluation, we sometimes need to store expressions. As such, we have the notion of both `Value` and `PreValue`. Memory will actually only store location/prevalue pairs. If we attempt to store a `Value`, it will be stored as a special `PreValue` (encapsulated in a `static` operator).

Chapter 3

Haskell Programs

Haskell programs have the same benefits as the example in Figure 1.4: concise, yet powerful, and very easily understood. In this chapter, we will reference many figures of examples we have created in Hugs [4]. We then show side-by-side the look and feel of the same program written in Haskell-RL.

3.1 Hugs/GHC Examples

Hugs provides an interactive prompt to evaluate expressions. One can also write a separate module and import it from that prompt. Hugs also includes a `Prelude` module, wherein is defined a collection of helper functions and definitions, ranging from the `Bool` data type, to list functions such as `head` and `tail`, and to higher-order functions such as `fold` and `map`. It is still possible to write programs without these functions, but it would usually be more tedious. Section 3.2 below redefines some of these functions.

GHC [5] works similarly to Hugs in its interactive mode, but also features a GNU licensed compiler, which will allow for much faster execution of all programs, while still providing the same benefits Haskell is known for. Chapter 5 includes performance benchmarks for it, but for now, the examples will be only Hugs-compliant.

Figure 3.1(a) shows a simple factorial definition in Hugs. We need to specify only the base case and the simple inductive case, and Haskell will do the rest. Note that we are not specifying what to do when `f` is passed a negative integer, but that isn't important for this example. Since the function definition above does not include any let expressions, it must be included in a file and loaded after the `Prelude`. The examples we show will be in the same manner, eg the first set of lines is meant to be placed in a file and loaded by Hugs, while the last line represents an expression that should be typed at the interactive prompt.

Figure 3.2(a) demonstrates a slightly more complex Haskell program than the factorial example. Computing Fibonacci numbers recursively takes exponential time, whereas generating a list (perhaps through list comprehensions) would be very quick. This example also illustrates the use of guards, which are just syntactic sugar for if-then-else statements.

(a)

```
f :: Int -> Int
f 0 = 1
f x = x * f (x - 1)

f 500
```

(b)

```
eval(
f 0 = 1 ;
f x = *(x,f (+((-1),x)))
,
f 500
) .
```

Figure 3.1: Example Program: Factorial

(a)

```
fib :: Int -> Int
fib n
  | n == 0 = 0
  | n == 1 = 1
  | n > 1  = fib(n - 2) + fib(n - 1)

fib 14
```

(b)

```
eval(
'fib n
  |=(==(n,0),0)
  |=(==(n,1),1)
  |=(>(n,1), (+('fib (+(n,-2)), ('fib (+(n,-1))))))
,
'fib 14
) .
```

Figure 3.2: Example Program: Fibonacci

(a)

```
f :: Int -> [Int]
f 0 = []
f x = x : (f (x - 1))

f 100
```

(b)

```
eval**(
data 'List = dt('Cons, (Int,'List)) || dt('Nil, ()) ;
f 0 = dt('Nil, ()) ;
f x = dt('Cons, (x,(f (+(-1,x)))))
,
f 100
) .
```

Figure 3.3: Example Program: List Building

(a)

```
data Node = Tree Node Node | Leaf Int
f :: Node -> Int
f (Leaf x) = x
f (Tree a b) = f a + f b

f (Tree (Leaf 3) (Tree (Leaf 4) (Leaf 1)))
```

(b)

```
eval(
data 'Node = dt('Tree, ('Node, 'Node)) ||
             dt('Leaf, Int) ;
f dt('Leaf, x) = x ;
f dt('Tree, (x,y)) = +(f x, f y)
,
f dt('Tree,(dt('Leaf,3),
            dt('Tree,(dt('Leaf,4), dt('Leaf,1)))))
) .
```

Figure 3.4: Example Program: Trees

(a) - untyped

```
len [] = 0
len (x:xs) = 1 + len xs
isnull [] = True
isnull _ = False
perm 0 = [[]]
perm n = insert n (perm (n - 1))
insert _ l
  | isnull l = []
insert n (l:ls) = append (interleave n l)
                         (insert n ls)
interleave n l
  | isnull l = [[n]]
interleave n (l:ls) = (n:l:ls):
              (mycons l (interleave n ls))
mycons x l
  | isnull l = []
mycons x (l:ls) = (x:l):(mycons x ls)
append u v
  | isnull u = v
append (u:us) v = u : (append us v)

len (perm 4)
```

(b)

```
eval(
data 'List = dt('Cons, (Int,'List)) || dt('Nil, ()) ;
'len dt('Nil, $) = 0 ;
'len dt('Cons, ($,'ls)) = (+(1,'len 'ls)) ;
'null dt('Nil, $) = True ;
'null $ = False ;
'perm 0 = dt('Cons, (dt('Nil, ()), dt('Nil, ()))) ;
'perm n = 'insert n ('perm (+((-1),n))) ;
'insert ($ l)  |=('null l, dt('Nil, ())) ;
'insert (n dt('Cons, (l,'ls))) =
      'append ('interleave n l) ('insert n 'ls) ;
'interleave (n l)
  |=('null l, dt('Cons, (dt('Cons, (n,dt('Nil, ()))),
       dt('Nil, ())))) ;
'interleave (n dt('Cons, (l,'ls))) =
       dt('Cons, (dt('Cons, (n,dt('Cons, (l,'ls)))),
                  ('mycons l ('interleave n 'ls)))) ;
'mycons (x l)  |=('null l, dt('Nil, ())) ;
'mycons (x dt('Cons, (l,'ls))) =
       dt('Cons, (dt('Cons, (x,l)),('mycons x 'ls))) ;
'append (u v)  |=('null u, v) ;
'append (dt('Cons, (u,'us)) v) =
       dt('Cons, (u,('append 'us v)))
,
'len ('perm 4)
) .
```

Figure 3.5: Example Program: Counting Permutations

In Figure 3.3(a), we have a simple list builder, that generates a list in descending order, from 100 to 1. Hugs also has built-in list comprehensions that can handle this, resulting in a headerless program of the simple expression `[100,99..1]`. The example shown is written out to be closest to how it appears in Haskell-RL, and similarly, the test runs will be done using programs written as similarly as possible, in an attempt to generate the same amount of overhead.

For the example in Figure 3.4(a), we have created a tree data type, and written a function that computes the sum over all leaves. Pattern matching is as graceful as it was for matching single integers, and thus allows us to write what would be complex iterative functions in fewer lines of code. We could further extend this example to create a polymorphic function `f` which works on trees of any data type that has the + operator defined over it.

Finally, Figure 3.5(a) shows the computation of the number of permutations of some list of the argument size. For most other programming languages, the computation would usually break down around lists of 6 or 7 elements. Haskell, due to being lazy and avoiding some computation of the list itself, can easily get to 9, and with a change in memory allowances, does 10 as well, on a modern computer. This example further illustrates pattern matching and the match-any operator. Note that this example is untyped, and is still accepted by Hugs (and GHC), but ignoring this functionality of the language is not recommended.

3.2 Haskell-RL Examples

The first thing that is noticeable in the Haskell-RL examples is the obfuscated syntax. Maude's parsing engine is not as good as the specialized ones that ship with Hugs and GHC. As such, sometimes an infix operator needs to be changed to be a prefix operator, or more parentheses need to be added. Most problematic is Maude's lack of whitespace parsing. Haskell makes extreme use of line breaks and hard spaces to know how to parse expressions; to make a Haskell program work in Maude, we had to insert sequential composition operators (;) into the programs. This last conversion alone is acceptable, as Hugs and GHC are setup to handle non-whitespace parsing as well with the explicit semicolons, but the other changes that were made make Haskell-RL programs not directly runnable in Hugs.

Currently, conversion from Haskell to Haskell-RL must be done by hand. However, the Formal Systems Laboratory research group [6] has started using SDF [7] with spectacular results. This would mean the ability to parse original Haskell programs with only slight modifications, if any, and feed them directly into Maude. At the time of this writing, we have not attempted to write an SDF for Haskell, but we are aware of one group who has [8].

For all Haskell-RL programs, we specify two parameters as input to Maude. The first one is what would have appeared in a .hs file and loaded at runtime with Hugs or GHC. The second argument is the actual expression we want to compute. Because Maude cannot maintain state between runs, anytime we want to run an example with the same function definitions but different expression to compute, we must still specify those definitions. Also undesireable is the additional overhead of translating those definitions every run into the specific format the semantics operates on. Nevertheless, this is how it must be for interactive input.

Booleans are defined simply as `True` and `False`, hardcoded into the language; and at one time, lists were available without having to specify the additional data type. Other definitions that would appear in the prelude could be precompiled and added into the language at runtime, to soften up this overhead of retransforming the function definitions into the form needed for semantics, but at this time, there is no nice set of functions or data types that we would always want included.

To evaluate actual programs, we surround the function definition part and the expression by an `eval` operator, which is first rewritten into the full initial state of Haskell-RL. At that time, we start rewriting down through the k-continuation, matching various expressions, attempting to equationally rewrite the program into the concrete solution. `eval` is provided to do as little work as possible to produce a result. As such, if there is a data type that is the final result, it will usually be found very quickly, and the recursive entries in the type will be left unevaluated. While we get an answer quickly, we usually get something that isn't very helpful, nor a complete solution. In these cases, we would desire to use `eval*` instead, which will

fully evaluate the final result (but not unnecessarily evaluate any expressions in the normal execution of the program), to produce the complete tree, or list, that we desire. Finally, `eval**`, will fully compute a data type, and translate any lists into a prettier output form, as would usually be seen when working with functional languages. This step is unnecessary, and is mostly useful for debugging programs involving lists.

Starting with Figure 3.1(b), we see exactly what we would feed to Maude (preceeded by the `red` keyword to tell Maude to reduce the expression) in order to compute the factorial of 500. We are not using complex data types, so `eval` is sufficient. Note that all of the operators that are so nicely written in infix notation in Haskell, must be explicitly written as prefix in Haskell-RL. The minus operator has not been implemented (nor has integer division), so the best we can do is add -1 in place of subtracting 1. Nevertheless, our final result is the same, and is returned after 42071 rewrites.

Figure 3.2(b) demonstrates the altered guard syntax, as well as the prefix binary operators that evaluate to booleans. As mentioned above, Fibonacci computation on demand incurs a lot of overhead, and because of the substantial number of rewrites and memory usage to store arguments in a call-by-need fashion for simple function calls, Maude quickly allocates a large chunk of memory (46 MB) to compute the 20th Fibonacci number.

In Figure 3.3(b), we really start to see the extra syntax, as well as use of the `eval**` operator. If we defined lists in Haskell using the same `Cons` notation, we would write it as `data List = Cons Int List | Nil`. Due to parsing difficulties, both in the definition of the data type and the use of a data type in a function, we had to surround all data types with a `dt` operator. `dt` takes two arguments, the first being the name of the constructor we wish to use, and the second being a non-empty list of expressions. We ran into more difficulties when we left the list of expressions as a single expression (using the function application operator to also build data types). Then, to traverse the elements of the data type, we would tear them apart anytime we had two next to each other. This lead to treating recursively embedded data types as additional arguments to the first type.

Moving onto Figure 3.4(b), we can better provide an example of poor parsing in Maude. This example specifically shows how to use a constructor on the left hand side of an equation, to then initiate pattern matching binding. We will talk about this much more in depth in Section 4.4. If we asked Maude to parse the expression `Tree (Tree (Leaf 3) (Leaf 4)) (Leaf 6)`, it will do its best to remove and store the expressions as it sees them. The name of the constructor is stored separately, so this becomes a problem of finding a split in `(Tree (Leaf 3) (Leaf 4)) (Leaf 6)`. First, due to the left-associativity of the function application operator, `Leaf 6` would be extracted and saved. Next, Haskell-RL would continue until it has split off as many expressions as possible, which means it recurses on `Tree (Leaf 3) (Leaf 4)`, pulling

off `Leaf 4`. This time, the name of the constructor is not saved separately, and the equations break what remains into two pieces, `Tree` and `Leaf 3`. It searches for a constructor with the name we pulled aside originally (`Tree`), taking the number of arguments we found (4) and fails. This failed parse is unacceptable, and can be avoided with some extra syntax. And given proper lexing and parsing into an abstract syntax tree, we can easily generate the Haskell-RL code from the Haskell example it was based upon.

The ultimate test to a successful Haskell implementation would be computing the length of a list we do not need to fully evaluate. The permutation example in Figure 3.5(b) attempts exactly that. Through test results in Chapter 5, it will be clear that computational steps are saved, about twenty percent in total. Figures 3.5(a) and (b) appear very similar, and are semantically equivalent as well. Extensively recursive functions are still handled gracefully in Haskell-RL, as well as the match-any operator. When a function is called in Haskell-RL, its arguments are saved into the store. Later, when the pattern matching semantics attempt to match that location in memory with a `$`, it will succeed without any evaluation, and without any variable bindings.

Chapter 4

Implemented Features

This section presents the subset of features of Haskell that have a working implementation in Haskell-RL. We will also be including Maude equations inline, and attempt to document their intended purpose. Afterwards, in Section 4.7, we discuss the largest of the features that are left out of our specification.

Maude supports integers and natural numbers internally, and we used both in our specification. Integers become one of the built-in types, along with booleans. Natural numbers are used for internal bookkeeping within Haskell-RL's state infrastructure.

4.1 Laziness

Haskell-RL was designed from the ground up, keeping laziness first and foremost. Haskell-RL, in its current state, delays execution of as many subexpressions as possible. Also, an expression that is evaluated once, is saved back to the memory location it originated from, meaning after being referenced once and computed, a single memory fetch is all that is required to get the already computed value. This defines the at-most-once computation aspect of Haskell.

```
var V : Value .  var L : Location .  var K : Continuation .  var Mem : Store .
eq k(val(V) -> save(L) -> K) mem(Mem) = k(val(V) -> K) mem(Mem[L <- static(V)]) .
```

This is the exact equation used for saving a value back into memory. As mentioned in Section 2.2, only `PreValue`'s are actually stored in memory. To accomplish this task with a concrete value then, we apply the `static` operator.

```
var C : Name .  var K : Contination .  var N : Nat .  var ECl : ExpCommaList .  var E : Exp .
var Env : Env .  var Mem : Store .  vars L L' : LocationList .
eq k(exp(dt(C,ECl)) -> K) = k(savet(ECl) -> construct(C) -> K) .
eq k(savet(E, ECl) -> K) = k(savet(E) -> savet(ECl) -> K) .
eq k(locs(L') -> locs(L) -> K) = k(locs(L,L') -> K) .
eq k(savet(E) -> K) env(Env) mem(Mem) nextloc(N) =
   k(locs(loc(N)) -> K) env(Env) mem(Mem[loc(N) <- frozen(E,Env,loc(N))]) nextloc(N + 1) .
```

When we process a data type, we strip off the constructor name, and set it aside, focusing on saving the expressions first. Haskell-RL will now iterate through the `ExpCommaList`, saving each expression to a new location in the store. We freeze the expression `E` along with the current environment, and its target memory location.

```
var X : Name .  var V : Value .  var K : Continuation .  vars Env Env' : Env .
var Mem : Store .  var L : Location .  var E : Exp .
eq k(exp(X) -> K) env(Env) mem(Mem) = k(pval(Mem[Env[X]]) -> K) env(Env) mem(Mem) .
eq k(pval(frozen(E,Env',L)) -> K) env(Env) = k(exp(E) -> save(L) -> recover(Env) -> K) env(Env') .
eq k(pval(static(V)) -> K) = k(val(V) -> K) .
```

These equations serve to show the steps for processing a memory lookup and evaluation of a frozen expression. If we are given a variable name to the top of the k-continuation, it means that some function or procedure has requested to have that value. We are going to need to lookup and return the value associated with that name. This boils down to be a simple access into the store for a particular location, the one referenced by that name in the environment. We can write this succinctly as `Mem[Env[X]]`, the `PreValue` associated with the name `X`.

Once we have the prevalue on top of the k-continuation, we can check to see if it has been encapsulated in the `frozen` operator, as would have occurred in the example above with data types. If so, we switch to the stored environment (making sure to recover our current environment after computation), and attempt to compute the expression stored within. After evaluation, we call our memory save mentioned above, to put that value back into memory. If we skipped this step, we would be using a call-by-name system instead of call-by-need. We only want to evaluate the expression `E` once, and this will ensure that.

If instead we had stored a concrete value in the store, we would have encapsulated it in that `static` operator, so now we can just convert that `PreValue` directly into a `Value`.

4.2 General Expressions

```
var I : Int .  var K : Continuation .
eq k(exp(I) -> K) = k(val(int(I)) -> K) .
```

This is the simplest expression to evaluate. Given some integer `I` in expression form, encapsulate it in an `int` operator, which is of sort `Value`, and place it back on the k-continuation. In the same manner, we can evaluate the unit expression `()` or booleans.

```
var El : ExpCommaList .  var K : Continuation .  var V : Value . var Vl : ValueList .  vars I I' : Int .
eq k(exp(+(El)) -> K) = k(exp(El) -> + -> K) .
```

```
eq k(val(int(I),int(I'),Vl) -> + -> K) = k(val(int(I + I'),Vl) -> + -> K)
eq k(val(V) -> + -> K) = k(val(V) -> K) .
```

Because arithmetic operations have been implemented in a prefix fashion, we are capable of handling them very elegantly. If there is ever a `+` operator applied to a list of expressions, we simply evaluate all of those expressions to values, and then add them one by one. To do the actual addition of the integers, we just use the built-in Maude operator, defined in the same module that provides the integers. The list of values `Vl` has an identity set, so it can match `nil`, meaning we do not need a separate case for when there's only two `Value`'s on the continuation. If we are ever left with a single value on top of the `+` on the k-continuation, we discard the operator and return the value.

Multiplication is (and subtraction, division could be) implemented in the very same fashion. Haskell-RL needs to handle the evaluation of an `ExpCommaList` in only one place, and then we can reference that any time we need to perform an operation like this.

```
vars E E' : Exp .  vars I I' : Int .  var K : Continuation .
eq k(exp(not(E)) -> K) = k(exp(E) -> not -> K) .
eq k(val(True) -> not -> K) = k(val(False) -> K) .
eq k(val(False) -> not -> K) = k(val(True) -> K) .
eq k(exp(<(E,E')) -> K) = k(exp(E,E') -> lt -> K) .
eq k(exp(>=(E,E')) -> K) = k(exp(E,E') -> lt -> not -> K) .
eq k(val(int(I),int(I')) -> lt -> K) = k(val(if I < I' then True else False fi) -> K) .
```

`not` is an easy operator to define, as it will just evaluate whatever expression it contains, and then, matching one of two equations, rewrite to be the boolean negation of that value.

Once we have booleans and unary/binary operators over expressions, we can use the new operators to build larger expressions. The operators `<` and `>=` are two examples of binary comparison operators, taking two expressions. Similarly to above, we break them apart from their subparts, evaluate the subparts, then proceed to produce a final result. To check if one integer is less than another, we again use built-in operators of Maude. If we wanted to know if an integer was greater than or equal to another integer, we would check if it were stricly less than the other, and then recurse back even more, taking the negation of that boolean value using our previously defined `not` operator.

Reuse is one of many reasons to adopt an equational rewriting system. If at any point we need to load a value from memory that we have a name hook on, we just put that name on that k-continuation; if we need to recover the environment, we just add to the continuation a `recover` operator with a copy of the environment we wish to recover; etc.

4.3 Let, Where, and Case

For Haskell-RL, expressions can be evaluated, whereas statements are used to convey information, and direct execution. The function definitions that appear as the first argument in `eval` are statements composed into statement lists. We can use that parameter to globally define functions and variables, or we can define them locally within a `let` or `where` expression.

```
op let_in_ : StmtList Exp -> Exp .
op _where_ : Exp StmtList -> Exp .
var E : Exp .  var Sl : StmtList .
eq E where Sl = let Sl in E .
```

The operator definitions provide syntax identical to Haskell's, while the equation does a simple reduction of `where` into `let`. Now, given either, we transform, at a syntactical level, into just one of them. We need only to specify semantics for the let-expression now.

```
var LL : StmtList .  var E : Exp .  var Env : Env .  var Mem : Store .  var N : Nat .  var K : Continuation .
eq k(exp(let LL in E) -> K) env(Env) mem(Mem) nextloc(N) =
   k(exp(E) -> recover(Env) -> K) makeEnvStore(env(Env) mem(Mem) nextloc(N) stmtpre(LL)) .
```

`let` definitions can be nontrivial, eg `let f 3 = 0 ; f _ = 1 in f 2`. It is not possible to just take all of the names, throw them into the environment, and then bind the right sides to those names. We need to process these functions first, combining same-named functions, and providing binding names. We have created a separate top-level operator to handle this, `makeEnvStore`. Because the environment and store are kept inside it, execution cannot continue until they are returned. We will show the semantics of this operator in Section 4.5.

Defined for operators like `let` and the top-level `eval` was a `_=_` operator, that produced a `Stmt` sort. To stay close to the Haskell syntax, Haskell-RL also defines a `_->_` operator which produces a statement, and is used for case statements. A case expression in Haskell-RL would appear much like `case 3 of (2 -> 4 ; 3 -> 2)`, with this expression evaluating to 2.

```
op _->_ : Exp Exp -> Stmt .
op case_of_ : Exp StmtList -> Exp .
vars E E' E'' XE : Exp .  var Sl : StmtList .  var Sl : StmtList .
vars ECl ECl' : ExpCommaList .  var Env : Env .  vars C K : Continuation .
eq k(exp(case E of (E' -> E'' ; Sl)) -> K) = k(exp(split(E, Sl, E', E''))) -> K) .
eq split(XE, nil , ECl , ECl') = caseof(XE, ECl , ECl') .
eq split(XE, (E -> E' ; Sl) , ECl , ECl') = split(XE, Sl , (ECl, E) , (ECl', E')) .
eq k(exp(caseof(XE, ECl, ECl')) -> K) env(Env) c(C) =
   k(endcase(0)) env(Env) c(case(XE, ECl, ECl', Env, K) -> C) .
```

(a)

```
case Exp of
  Exp1 -> Exp1' ;
  Exp2 -> Exp2' ;
  ...
  ExpN -> ExpN'
```

(b)

```
caseof(Exp ,
  Exp1,Exp2,...,ExpN ,
  Exp1',Exp2',...,ExpN')
```

Figure 4.1: Rewriting Case Statements

To keep things lazy, case expressions are going to introduce many helper operators and equations. The syntax to parse these is very straightforward, but once we try to evaluate it, we need to change around its form. We start with an expression of the form shown in Figure 4.1(a), and make a single pass through to convert it to the form in Figure 4.1(b).

Finally, we will make use of the c-continuation to store what was remaining on the k-continuation (the continuation we will pass the resultant value of this case to), as well as the environment that existed when we began. We go through, one-by-one, trying to match `Exp` to `ExpI`, and if we find a match, we evaluate `ExpI'`.

The `endcase` operator on the top of the continuation indicates we failed at matching, and need to try the next case. We use pattern matching, which appears in Section 4.4 to pair up parts of the argumented expression and the current case. If we run out of cases to try, our c-continuation operator shrinks, and then in the next round, disappears as the result of the case expression becomes `val(bottom)`.

```
vars E E' E'' XE : Exp .  vars ECl ECl' : ExpCommaList .  vars Env Env' : Env .  vars C K : Continuation .
eq k(endcase(E'')) env(Env') c(case(XE, (E,ECl), (E',ECl'), Env, K) -> C) =
   k(checkcase(XE) -> matchbind(E) -> endcase(E')) c(case(XE, ECl, ECl', Env, K) -> C) env(Env) .
eq k(endcase(E'')) c(case(XE, E, E', Env, K) -> C) env(Env') =
   k(checkcase(XE) -> matchbind(E) -> endcase(E')) c(case(Env, K) -> C) env(Env) .
eq k(endcase(E)) c(case(Env, K) -> C) env(Env') = k(val(bottom) -> K) c(C) env(Env) .
```

The first equation handles the case when we have more than one case left to check, the second when we are attempting the last case, and the third for when we have exhausted all cases and will return `val(bottom)`. Also notice that the environment stored in the c-continuation is refreshed every iteration of the case check. Pattern matching has the side-effect that it can bind new names in the environment. Examples are shown below. Because of this, we discard the environment from the failed attempt, and replace it with the environment we had at the time of first evaluation of the case expression.

4.4 Pattern Matching

Returning to case expressions, we have a general form that appeared in the last section. The argumented expression can be any simple or complex expression (a single integer, or even a let-expression), and the expressions appearing to the left of the arrow operator (->) can be one of many patterns. A pattern can be a literal, a wildcard ($), a variable name, a tuple, or some named data type, referring recursively to other patterns. Thus, we can build a simple recursive method to evaluate if an expression matches a pattern.

```
var E : Exp .  vars X X' : Name .  var K : Continuation .  var Env : Env .  var Mem : Store .  var N : Nat .
eq k(checkcase(E) -> matchbind($) -> K) = k(good -> K) .
eq k(checkcase(X) -> matchbind(X') -> K) env(Env) = k(good -> K) env(Env[X' <- Env[X]]) .
eq k(checkcase(E) -> matchbind(X) -> K) env(Env) mem(Mem) nextloc(N) =
   k(good -> K) env(Env[X <- loc(N)]) mem(Mem[loc(N) <- frozen(E, Env, loc(N))]) nextloc(N + 1) .
eq k(checkcase(E) -> K) env(Env) = k(exp(E) -> K) env(Env) [owise] .
```

The first three equations will provide the first few cases of matching and binding. Given any expression, without evaluation, it will match with the wildcard. Given two names, just point the new name to the location of the old one. Given an expression and a name, freeze that expression in the current environment, and bind the location to the name in the match. Otherwise, we cannot match at this level, and need to evaluate further.

```
vars X X' : Name . var Ll : LocationList . vars I I' : Int . var K : Continuation . var ECl : ExpCommaList .
 eq k(val(()) -> matchbind(()) -> K) = k(good -> K) .
 eq k(val(int(I)) -> matchbind(I) -> K) = k(good -> K) .
ceq k(val(int(I)) -> matchbind(I') -> K) = k(casebad -> K)
 if I =/= I' .
 eq k(val(construct(X,Ll)) -> matchbind(dt(X,ECl)) -> K) = k(makeande(Ll, ECl) -> K) .
ceq k(val(construct(X,Ll)) -> matchbind(dt(X',ECl)) -> K) = k(casebad -> K)
  if X =/= X'
```

Above are some of the pattern matching equations for values. If we have a basic type, such as an integer, the pattern matching will succeed if the integers are equal, and fail if they are different. Similarly, if we have a user-defined data type (or tuples, as they work the same way), and we are able to match the names of the constructors, then we can attempt to match the fields in the data type recursively. However, a `construct` value will be holding location pointers to possibly frozen subexpressions, whereas the `dt` expression that appears in the `matchbind` operator holds unevaluated and unfrozen expressions. Not only do we need to split up the lists in the case where there are multiple expressions to match, but also we need a way to evaluate location-expressions.

```
var L : Location .  var Ll : LocationList .  var E : Exp .
var ECl : ExpCommaList .  var K : Continuation .  var Mem : Store .
eq k(makeande(L, $) -> K) = k(good -> K) .
eq k(makeande((Ll,L), (ECl, $)) -> K) = k(makeande(Ll, ECl) -> K) .
eq k(makeande(L, E) -> K) = k(checkcasel(L) -> matchbind(E) -> K) .
eq k(makeande((Ll, L), (ECl, E)) -> K) = k(makeande(Ll, ECl) -> cand -> checkcasel(L) -> matchbind(E) -> K) .
eq k(good -> cand -> K) = k(K) .
eq k(good -> endcase(E)) = k(exp(E) -> casegood) .

eq k(lexp(L) -> K) mem(Mem) = k(pval(Mem[L]) -> K) mem(Mem) .
```

Location-expressions are easier to fetch from memory than variable-expressions, as we do not need to do an environment lookup. Haskell-RL takes advantage of this, and attempts to store locations whenever it will not need to refer to an expression by name. The `makeande` operator splits up cases with ease, and discards any immediately that are going to be matched against the wildcard expression.

And in the event we were unable to match a pattern at any level, we write `casebad` on top of the continuation, and it will consume all continuation items until the end; and then the next case is tried. Note the difference between a "good" case and a "bad" one. The successful case will proceed as above, evaluating the expression stored in the last continuation item, and return that value to outside of the case; a failed case will result in simply the `endcase` operator, which is replaced by the next case (ie never evaluated).

```
var E : Exp .  var CI : ContinuationItem .  var K : Continuation .
eq k(casebad -> CI -> K) = k(casebad -> K) .
eq k(casebad -> endcase(E)) = k(endcase(E)) .
```

4.5 Functions

Functions can appear in any context, bound to a name, or anonymous. Functions declared at the top-level (inside `eval`) or within a let-expression can also be expressed over multiple lines. After the function name can appear one or more literals, data types, or possibly recursive patterns. And when one function definition fails to match its binding variables, we want to proceed and try the next one. This is exactly `case`; thus, we will rewrite all function definitions using a case expression.

Haskell-RL takes a couple precautions first. Take, for example, Figure 4.2. The binding variable `a` appears multiple times, but in different places. We don't want to bind a variable for a function definition that we are not using, so we are going to need to create new variables. Instead of searching for free variables within our alphabet (and because there may not exist any), there is another top-level state member, `nextvar`. We

```
f (0 a) = a ;
f (7 $) = 9 ;
f (a 3)
  |=(>(a,3),a) ;
f (a $) = 0
```

Figure 4.2: Multiline Function Definition

```
f = \ fv(0) -> \ fv(1) -> case Tuple(fv(0),fv(1)) of
                           ( Tuple(0,a) -> a ;
                             Tuple(7,$) -> 9 ;
                             Tuple(a,3) -> if(>(a, 3), a, nomatch) ;
                             Tuple(a,$) -> 0 )
```

Figure 4.3: Converted Function Definition

can use this to construct a new variable anytime we need one. `fv(5)` refers to the fifth free variable we have allocated in this manner.

So now we can, for a two-argument function, freeze and bind arguments to the function to two free variables. If our two variables were `fv(3)` and `fv(4)`, in order to check the first function definition we would see if `fv(3)` was equal to 0, and if so, we run pattern matching on `fv(4)` to bind it to `a`, and then evaluate the expression `a`. This gives us ultimate flexability in variable name choices.

This rewriting happens once at runtime for the globally defined functions, and again anytime a let-expression is used. Haskell-RL saves the environment and store into a top-level container, `MakeEnvStore`, and then processes what appears. The `stmt` operator holds the current list of functions to be processed, after their initial run through `stmtpre`, which binds the names to locations in the environment, so that `let` works more like a `letrec` (and so that all global functions can see each other).

```
vars Cl Sl : StmtList .  vars E P P' : Exp .  var X : Name .  var N : nat .  var Env : Env .
eq stmtpre(Sl) = stmtprebind(Sl) saved(Sl) .
eq stmtprebind(nil) saved(Sl) = stmt(Sl) .
eq stmtprebind(X = E ; Sl) nextloc(N) env(Env) = stmtprebind(Sl) nextloc(N + 1) env(Env[X <- loc(N)]) .

eq stmt(X P = E ; Sl) = stmt(X P = (case($) of (maketuple(P) -> E)) ; Sl) .
eq stmt(X P = (case($) of Cl) ; X P' = E' ; Sl) = stmt(X P = (case($) of (Cl ; maketuple(P') -> E')) ; Sl) .
eq stmt(X P = (case($) of Cl) ; Sl) = stmt(free(X,P) = (case($) of Cl) ; Sl) .
```

After binding the names to the environment, we rewrite all properly grouped same-named functions into one case statement. `maketuple` creates a `Tuple` data type out of the arguments that would have appeared on the lefthand side of the equation. Once we have seen all definitions of that name, we can look for those free names, which will eventually be made into a `Tuple` as well, and then become the binding variables for this function. Finally, we will curry the function until we have just the function name on the left side of the

equation, at which point we will freeze it and place it into the store. The expression `case($) of ...` is not valid in Haskell, so we make use of that at this point for bookkeeping purposes; in the final conversion, it has been removed. See Figure 4.3 for the completely rewritten version of Figure 4.2.

If you look closely at Figure 4.3, you see the conversion for a guard as well. Basic conversion to if-then-else, but there's also a final result of `nomatch` if none of the guards are true. This allows the guards to be interspersed with other function definitions. If a `nomatch` value is returned from a case statement, Haskell-RL catches it and realizes it needs to proceed to the next case, and does just that.

After we have curried the function, we are also capable of partial application. More importantly, only one environment update is performed during a partial application. At that point, a new closure is formed, and returned as a value, perhaps to be saved into the store, or used immediately on another expression.

Using free variables also emphasizes more lazy evaluation. When we call the function, we freeze the expression, and the location is bound to the free variable. If that free variable never occurs during the case statements, the expression is never unfrozen. If a name is seen in the closure binding pattern field, a simpler function call can take place. Otherwise, we simply build a case-expression and let it do the pattern matching.

```
vars E AE Pat : Exp .  var K : Continuation .  var X : Name .
var Env : Env .  vars N N' : Nat .  var Mem : Store .
eq k(exp(E AE) -> K) = k(exp(E) -> store(AE) -> K) .
eq k(val(closure(X,E,Env)) -> store(AE) -> K) env(Env') mem(Mem) nextloc(N') nextvar(N) =
   k(exp(E) -> recover(Env') -> K) nextloc(N' + 1) nextvar(N)
    env(Env[X <- loc(N')]) mem(Mem[loc(N') <- frozen(AE,Env',loc(N'))]) .
eq k(val(closure(Pat,E,Env)) -> store(AE) -> K) env(Env') mem(Mem) nextloc(N') nextvar(N) =
   k(exp(case (fv(N)) of Pat -> E) -> recover(Env') -> K) nextloc(N' + 1) nextvar(N + 1)
    env(Env[fv(N) <- loc(N')]) mem(Mem[loc(N') <- frozen(AE,Env',loc(N'))]) .
```

Due mainly to the lack of a typing system, functions are polymorphic. If a typing system existed, it would also be possible to define functions that were explicitly polymorphic (supported in Haskell). This, however, makes it very easy to define functions in Haskell-RL, and also slightly more efficient to run programs.

4.6 Data Types

Expanding slightly on what was mentioned in Section 4.1, data types are basically wrappers of other data types along with a constructor name. Tuples are specialized data types in that they do not have constructor names; instead they are just a list of previously defined data types, eg `(3,'a',4.5)`.

When a data type is encountered as an expression, all of the parameter expressions are saved into the store, and their locations are placed in a `construct` operator along with the constructor name (or `'Tuple'`

```
eval(
f 0 = dt('Nil, ()) ;
f x = dt('Cons, (x,(f (+(-1,x)))))
,
f 10
) .
```

Figure 4.4: eval* Example: Program

```
(a) result Value: construct('Cons, loc(6),loc(7))
(b) result Value: cv('Cons, int(10),cv('Cons, int(9),cv('Cons, int(8),cv('Cons, int(7),cv('Cons, int(6),
  cv('Cons, int(5),cv('Cons, int(4),cv('Cons, int(3),cv('Cons, int(2),cv('Cons, int(1),cv('Nil, ()))))))))))
(c) result List: [int(10),int(9),int(8),int(7),int(6),int(5),int(4),int(3),int(2),int(1)]
```

Figure 4.5: eval* Example: Result

in the case of tuples). From here on, there are only two ways to get to the expressions that have been frozen: pattern matching the constructor name against a case-expression, or when operating under `eval*` and trying to expand the data type for proper output display.

For output purposes, you can force an expression to evaluate and print more completely. Starting with the program in Figure 4.4, using `eval` produces the output seen in Figure 4.5(a). Changing to `eval*` produces output like Figure 4.5(b). `eval**` results in Figure 4.5(c).

Data types can be declared only at the top-level, and are processed at the same time as function definitions. When a list of constructors is encountered, they are added one-by-one into the `datas` state attribute.

```
vars X C : Name .  var ECl : ExpCommaList .  var N : Nat .  var E : Exp .
var Sl : StmtList .  var Env : Env .  var Mem : Mem .  var Data : Data .
eq makeEnvStore(stmt(data X = dt(C,ECl) || E ; Sl) env(Env) mem(Mem) nextloc(N)) datas(Data) =
   makeEnvStore(stmt(data X = E ; Sl) env(Env) mem(Mem) nextloc(N)) datas(Data[X,C,ECl]) .
eq makeEnvStore(stmt(data X = dt(C,ECl) ; Sl) env(Env) mem(Mem) nextloc(N)) datas(Data) =
   makeEnvStore(stmt(Sl) env(Env) mem(Mem) nextloc(N)) datas(Data[X,C,ECl]) .
```

As a reminder, data types are both algebraic and recursive, in that they can refer to simple built-in types, or they can reference themselves and other data types. In this sense, it is trivial to specify the data type for a list or tree, as easy as it would be for a pair-like type.

4.7 Features Not Implemented

As mentioned throughout this paper, there is no type-checking done; presently there is not even syntax for it. At one point, it was possible to say `f :: Int -> Int` in Haskell-RL, but this is no longer even parsed. Similarly, there was a `types` state attribute. These were removed because we felt type checking was not necessary to this successful specification of Haskell. If types were added, they could be statically checked,

and then execution would proceed, effectively with typing disabled. Since our work focused on runtime reductions, types would have had no effect on the semantics we provide.

Also not present are the other built-in types frequently found in Haskell. These would include chars, strings, and floats. With a slight extension of the built-in's, we could also generate a list-type that is currently absent from the specifications.

There is no concept of I/O for Haskell-RL. The current implementation is incapable of reading from a state attribute containing an input stream of some sort. This would not be difficult to implement; we would need only a reserved word to signal that we want to read input from some input buffer. Similarly with output, we do not print what we want to have returned. The expression being evaluated is equivalent to the final value. This is more of a personal choice, as we believe functional languages are best utilized as value-returning, not output along the way.

Monadic programming is a newer concept introduced into functional languages, and supported by Haskell-98. While it may provide certain capabilities that are not currently present in Haskell-RL, we believe it would conflict with some of our equations. We expected from the beginning that no expression anywhere in the running of a program will produce side-effects, and if one did (perhaps through a monad), it would violate a lot of constraints we placed on the equations.

List comprehensions is our favorite feature of Haskell. However, we did not have time to implement them. Also, without pretty syntax for lists, it is difficult to demonstrate a list comprehension. For Haskell, a simple example is `[2*x|x<-[1..10]]`. This expression is saying, for each `x` taken from the list `[1..10]`, double it, and add it to the list. List comprehensions are an easy extension of while-loops, however, that feature is also not supported in Haskell-RL.

Chapter 5

Test Runs and Results

To provide a comparison between Haskell and Haskell-RL performance, we have run the programs that appear in Figures 3.1, 3.2, and 3.5. Test results appear in Table 5.1. There will be two versions of the permutation example, one that computes the full list, and one that computes just the length of the list. For the GHC results, we are using ghc-6.4.1, compiling with no special flags, and timing with `time`. We are using Hugs-98 Mar 2005, and have not found a definitive way to time programs, but will include the number of reductions, and a stopwatch time. For Haskell-RL, we are using the build that is included in the Appendix, and using Maude's built-in timing capabilities. For FUN, we are using the latest known speed version; FUN is a toy language created for teaching functional programming concepts in Maude.

The following sections discuss any interesting results.

5.1 Compiled Haskell (GHC)

GHC makes use of a complex rewriting algorithm to turn Haskell code into very obfuscated C, which is then compiled into actual machine code. GHC is not bothered by larger trials of any tests, except the higher factorial ones. Even when Hugs took multiple seconds for a long permutation list, GHC did not see the same increase.

5.2 Hugs

There is no available option in interactive Hugs to show computation time. Instead, we report reductions (which are a very rough estimate of how much work is being done) and stopwatch time, which for many entries, was less than a quarter second.

Hugs keeps up with GHC well, only starting to see deficiencies in the permutation test. Worth noting is the clear decrease in number of reductions from computing the complete permutations of a number to evaluating the length of that list. This represents a 75% decrease.

		GHC	Hugs		Haskell-RL		FUN	
Test Program	Parameter	Time	Reductions	Time	Rewrites	Time	Rewrites	Time
Factorial	1000	0.008s	13034	n/a	84071	628ms	44047	84ms
Factorial	5000	0.340s	65031	1s	420071	3392ms	220047	516ms
Factorial	25000	12.472s	325031	12s	2100071	20437ms	1100047	5348ms
Fibonacci	10	0.004s	3180	n/a	14929	72ms	8715	12ms
Fibonacci	15	0.004s	35243	n/a	166379	844ms	97093	140ms
Fibonacci	20	0.004s	390861	n/a	1845989	11212ms	1077222	1652ms
Permutations	4	0s	1349	n/a	31286	296ms	7161	12ms
Permutations	5	0s	7410	n/a	160921	1628ms	35910	68ms
Permutations	6	0.004s	50143	n/a	1024500	11008ms	226179	480ms
Permutations	7	0.016s	394316	2s	7680359	85521ms	1685688	7416ms
Permutations	len 4	0s	508	n/a	27435	236ms	8001	12ms
Permutations	len 5	0s	2249	n/a	137228	1260ms	40014	88ms
Permutations	len 6	0s	13422	n/a	855775	8272ms	250683	548ms
Permutations	len 7	0.004s	96955	n/a	6311634	63131ms	1857072	7952ms

Table 5.1: Test Results

5.3 Haskell-RL

Haskell-RL, does not see quite the same benefits for being lazy. While we do reduce rewrites and running time, we only get about 20%. While this is still acceptable, it shows that there is a lot of overhead in the non-lazy part of the implementation that may need to be looked at.

For the other tests, Haskell-RL is very competitive in terms of staying close to GHC and Hugs. Taking two to four times longer is very good for a strictly equation-based language. We are pleased with the performance numbers in every run except the permutation list lengths.

5.4 FUN

This version of FUN is an example of high performance within Maude. After many revisions, it runs efficiently wherever it can. Haskell-RL is the only implementation that doesn't scale up as quickly on factorials; even FUN gains 5x rewrites and 10x time, versus 5x rewrites and 7x time with Haskell-RL.

Chapter 6

Conclusion

Haskell-RL provides an example of a lazy language based on rewriting logic, written completely in Maude. The performance numbers are quite decent for as much overhead as the language entails; and these runtimes could be improved on greatly with proper compilation into machine code. Haskell-RL promises to be a very large step towards a complete Haskell-98 implementation under Maude.

Appendix A

Complete Haskell-RL Specification

```
(001) fmod NAME is
(002)   including QID .
(003)   including NAT .
(004)   sort Name .
(005)   subsort Qid < Name .
(006)   ops a b c d e f g h i j k l m n o p q r t u v w x y z : -> Name .
(007)
(008)   op $ : -> Name .
(009)   op fv : Nat -> Name .
(010) endfm
(011)
(012) fmod EXP is
(013)   protecting INT .
(014)   protecting NAME .
(015)   sort Exp ExpCommaList .
(016)   subsort Int Name < Exp < ExpCommaList .
(017)   op __ : Exp Exp -> Exp [gather (E e) prec 1] .
(018)   op _,_ : ExpCommaList ExpCommaList -> ExpCommaList [assoc] .
(019)   op _||_ : Exp Exp -> Exp [assoc prec 55] .
(020)
(021)   op `(` : -> Exp .
(022)   op Tuple : ExpCommaList -> Exp .
(023)   op dt : Exp ExpCommaList -> Exp .
(024)
(025)   op bottom : -> Exp .
(026) endfm
(027)
(028) fmod MATH is
(029)   protecting EXP .
(030)   ops + * min max : ExpCommaList -> Exp .
(031)   ops ^ div mod : Exp Exp -> Exp .
(032)   op sub : Exp -> Exp .
```

```
(033)   op abs : Exp -> Exp [ditto] .
(034)   op negate : Exp -> Exp .
(035)
(036)   ops True False : -> Exp .
(037)   ops < <= >= > == /= : Exp Exp -> Exp .
(038)   ops and or : Exp Exp -> Exp .
(039)   op not : Exp -> Exp .
(040)
(041)   op if : Exp Exp Exp -> Exp .
(042) endfm
(043)
(044) fmod STMT is
(045)   sorts Stmt StmtList .
(046)   subsort Stmt < StmtList .
(047)   op nil : -> StmtList .
(048)   op _;_ : StmtList StmtList -> StmtList [prec 70 assoc id: nil] .
(049) endfm
(050)
(051) fmod GUARD-SYNTAX is
(052)   protecting EXP .
(053)   protecting MATH .
(054)   sorts Guard Guards GuardExp .
(055)   subsort Guard < Guards .
(056)   op __ : Guards Guards -> Guards [assoc] .
(057)   op |= : Exp Exp -> Guard .
(058)   op otherwise : -> Exp .
(059)
(060)   op nomatch : -> Exp .
(061) endfm
(062)
(063) fmod FUNCTION-SYNTAX is
(064)   protecting EXP .
(065)   protecting STMT .
(066)   protecting GUARD-SYNTAX .
(067)
(068)   op __ : Exp Guards -> Stmt [prec 59] .
(069)   op _=_ : Exp Exp -> Stmt [prec 59] .
(070)   op \_->_ : Exp Exp -> Exp [prec 58] .
(071)
(072)   op ifit : Guards -> Exp .
```

```
(073)
(074)   var F : Name .  vars E E' P : Exp .  var G : Guards .
(075)
(076)   eq ifit(|=(E,E')) = if(E,E',nomatch) .
(077)   eq ifit(|=(E,E') G) = if(E,E',ifit(G)) .
(078)   eq (F P G) = (F P = ifit(G)) .
(079) endfm
(080)
(081) fmod LET-SYNTAX is
(082)   protecting FUNCTION-SYNTAX .
(083)
(084)   op let_in_ : StmtList Exp -> Exp .
(085) endfm
(086)
(087) fmod WHERE is
(088)   protecting EXP .
(089)   protecting STMT .
(090)   protecting LET-SYNTAX .
(091)   op _where_ : Exp StmtList -> Exp [prec 69] .
(092)
(093)   var E : Exp .  var Sl : StmtList .
(094)
(095)   eq E where Sl = let Sl in E .
(096) endfm
(097)
(098) fmod CASE-SYNTAX is
(099)   protecting EXP .
(100)   protecting STMT .
(101)   protecting FUNCTION-SYNTAX .
(102)
(103)   op _->_ : Exp Exp -> Stmt .
(104)   op case_of_ : Exp StmtList -> Exp .
(105) endfm
(106)
(107) fmod DATA-SYNTAX is
(108)   protecting EXP .
(109)   protecting STMT .
(110)
(111)   op data : -> Exp .
(112)   op T : -> Exp .
```

```
(113) endfm
(114)
(115) fmod HASKELL-SYNTAX is
(116)   protecting NAME .
(117)   protecting EXP .
(118)   protecting MATH .
(119)   protecting STMT .
(120)   protecting WHERE .
(121)   protecting FUNCTION-SYNTAX .
(122)   protecting LET-SYNTAX .
(123)   protecting CASE-SYNTAX .
(124)   protecting DATA-SYNTAX .
(125) endfm
(126)
(127) fmod LOCATION is
(128)   including INT .
(129)   sorts Location LocationList .
(130)   subsort Location < LocationList .
(131)   op loc : Nat -> Location .
(132)   op nil : -> LocationList .
(133)   op _,_ : LocationList LocationList -> LocationList [assoc id: nil] .
(134) endfm
(135)
(136) fmod ENVIRONMENT is
(137)   including NAME .
(138)   including LOCATION .
(139)   sort Env .
(140)   op empty : -> Env .
(141)   op [_,_] : Name Location -> Env .
(142)   op __ : Env Env -> Env [assoc comm id: empty] .
(143)   op _[_<-_] : Env Name Location -> [Env] .
(144)   op _[_] : Env Name -> [Location] .
(145)
(146)   var X : Name .  var Env : Env .  vars L L' : Location .
(147)
(148)   eq ([X,L] Env)[X <- L'] = ([X,L'] Env) .
(149)   eq Env [X <- L] = Env [X, L] [owise] .
(150)   eq ([X,L] Env)[X] = L .
(151) endfm
(152)
```

```
(153) fmod VALUE is
(154)   sorts PreValue Value ValueList ValueColonList .
(155)   subsort Value < ValueList ValueColonList .
(156)   op nil : -> ValueList .
(157)   op nil : -> ValueColonList .
(158)   op _,_ : ValueList ValueList -> ValueList [assoc id: nil] .
(159)
(160)   ops True False : -> Value .
(161)   op bottom : -> Value .
(162)   op bottom : -> PreValue .
(163)   op `(` : -> Value .
(164) endfm
(165)
(166) fmod STORE is
(167)   including LOCATION .
(168)   including VALUE .
(169)   sort Store .
(170)   op empty : -> Store .
(171)   op [_,_] : Location PreValue -> Store .
(172)   op __ : Store Store -> Store [assoc comm id: empty] .
(173)   op _[_<-_] : Store Location PreValue -> [Store] .
(174)   op _[_] : Store Location -> [PreValue] .
(175)
(176)   var L : Location .  var Mem : Store .  vars V V' : PreValue .
(177)
(178)   eq ([L,V] Mem)[L <- V'] = ([L,V'] Mem) .
(179)   eq Mem [L <- V] = Mem [L, V] [owise] .
(180)   eq ([L,V] Mem)[L] = V .
(181) endfm
(182)
(183) fmod DATA is
(184)   including NAME .
(185)   including EXP .
(186)   sort Data .
(187)
(188)   op empty : -> Data .
(189)   op [_,_,_] : Name Name ExpCommaList -> Data .
(190)   op __ : Data Data -> Data [assoc comm id: empty] .
(191) endfm
(192)
```

```
(193) fmod CONTINUATION is
(194)   sorts Continuation ContinuationItem .
(195)   op stop : -> Continuation .
(196)   op _->_ : ContinuationItem Continuation -> Continuation .
(197) endfm
(198)
(199) fmod HASK-STATE is
(200)   protecting ENVIRONMENT .
(201)   protecting STORE .
(202)   protecting CONTINUATION .
(203)   protecting STMT .
(204)   protecting DATA .
(205)
(206)   sorts HaskStateAttribute HaskState .
(207)   subsort HaskStateAttribute < HaskState .
(208)
(209)   op empty : -> HaskState .
(210)   op __ : HaskState HaskState -> HaskState [assoc comm id: empty] .
(211)   op mem : Store -> HaskStateAttribute .
(212)   op k : Continuation -> HaskStateAttribute .
(213)   op c : Continuation -> HaskStateAttribute .
(214)   op nextloc : Nat -> HaskStateAttribute .
(215)   op nextvar : Nat -> HaskStateAttribute .
(216)   op env : Env -> HaskStateAttribute .
(217)   op result : Continuation -> HaskStateAttribute .
(218)
(219)   op datas : Data -> HaskStateAttribute .
(220) endfm
(221)
(222) fmod HASK-HELPING-OPERATIONS is
(223)   including HASK-STATE .
(224)   including INT .
(225)
(226)   op val : ValueList -> ContinuationItem .
(227)   op pval : PreValue -> ContinuationItem .
(228)   op recover : Env -> ContinuationItem .
(229)   op int : Int -> Value .
(230)   op save : Location -> ContinuationItem .
(231)   op static : Value -> PreValue .
(232)   op bindTo : Name -> ContinuationItem .
```

```
(233)   op construct : Name LocationList -> Value .
(234)   op cv : Name ValueList -> Value .
(235)
(236)   var L : Location .  var V : Value .  var K : Continuation .
(237)   vars Env Env' : Env .  var Mem : Store .
(238)
(239)   eq k(val(V) -> save(L) -> K) mem(Mem) = k(val(V) -> K) mem(Mem[L <- static(V)]) .
(240)   eq k(val(V) -> recover(Env) -> K) env(Env') = k(val(V) -> K) env(Env) .
(241) endfm
(242)
(243) fmod GENERIC-EXP-SEMANTICS is
(244)   including HASK-STATE .
(245)   including HASK-HELPING-OPERATIONS .
(246)   including EXP .
(247)   including FUNCTION-SYNTAX .
(248)   including CASE-SYNTAX .
(249)   including STMT .
(250)
(251)   op stmt : StmtList -> HaskStateAttribute .
(252)   op exp : ExpCommaList -> ContinuationItem .
(253)   op lexp : LocationList -> ContinuationItem .
(254)   op store : Exp -> ContinuationItem .
(255)   op closure : Exp Exp Env -> Value .
(256)   op frozen : Exp Env Location -> PreValue .
(257)   op bind : Name Location -> ContinuationItem .
(258)   op tuple : -> ContinuationItem .
(259)
(260)   var K : Continuation .  var I : Int .  var X : Name .  vars E AE Pat : Exp .
(261)   vars Env Env' : Env .  var Mem : Store .  var El : ExpCommaList .
(262)   var V : Value .  var Vl : ValueList .  var L : Location .  var Ll : LocationList .
(263)   vars N N' : Nat .
(264)
(265)   eq k(exp(()) -> K) = k(val(()) -> K) .
(266)   eq k(exp(bottom) -> K) = k(val(bottom) -> K) .
(267)
(268)   eq k(exp(I) -> K) = k(val(int(I)) -> K) .
(269)   eq k(exp(True) -> K) = k(val(True) -> K) .
(270)   eq k(exp(False) -> K) = k(val(False) -> K) .
(271)   eq k(exp(otherwise) -> K) = k(val(True) -> K) .
(272)
```

```
(273)   eq k(exp(X) -> K) env(Env) mem(Mem) = k(pval(Mem[Env[X]]) -> K) env(Env) mem(Mem) .
(274)
(275)   eq k(exp(E,El) -> K) = k(exp(E) -> exp(El) -> K) .
(276)   eq k(val(V) -> exp(El) -> K) = k(exp(El) -> val(V) -> K) .
(277)   eq k(val(Vl) -> val(V) -> K) = k(val(V,Vl) -> K) .
(278)
(279)   eq k(lexp(L) -> K) mem(Mem) = k(pval(Mem[L]) -> K) mem(Mem) .
(280)  ceq k(lexp(L,Ll) -> K) = k(lexp(L) -> lexp(Ll) -> K)
(281)   if Ll =/= nil .
(282)   eq k(val(V) -> lexp(Ll) -> K) = k(lexp(Ll) -> val(V) -> K) .
(283)
(284)   eq k(exp(E AE) -> K) = k(exp(E) -> store(AE) -> K) .
(285)
(286)   eq k(val(closure(X,E,Env)) -> store(AE) -> K) env(Env') mem(Mem) nextloc(N') nextvar(N) =
(287)      k(exp(E) -> recover(Env') -> K) env(Env[X <- loc(N')])
(288)       mem(Mem[loc(N') <- frozen(AE,Env',loc(N'))]) nextloc(N' + 1) nextvar(N) .
(289)   eq k(val(closure(Pat,E,Env)) -> store(AE) -> K) env(Env') mem(Mem) nextloc(N') nextvar(N) =
(290)      k(exp(case (fv(N)) of Pat -> E) -> recover(Env') -> K) env(Env[fv(N) <- loc(N')])
(291)       mem(Mem[loc(N') <- frozen(AE,Env',loc(N'))]) nextloc(N' + 1) nextvar(N + 1) .
(292)
(293)   eq k(val(V) -> bindTo(X) -> K) env(Env) mem(Mem) nextloc(N) =
(294)      k(K) env(Env[X <- loc(N)]) mem(Mem[loc(N) <- static(V)]) nextloc(N + 1) .
(295)
(296)   eq k(pval(frozen(E,Env',L)) -> K) env(Env) = k(exp(E) -> save(L) -> recover(Env) -> K) env(Env') .
(297)   eq k(pval(static(V)) -> K) = k(val(V) -> K) .
(298) endfm
(299)
(300) fmod MATH-SEMANTICS is
(301)   including GENERIC-EXP-SEMANTICS .
(302)   including MATH .
(303)
(304)   ops + * negate : -> ContinuationItem .
(305)
(306)   var K : Continuation .  var V : Value .  vars I I' : Int .
(307)   vars E E' C : Exp .  vars El : ExpCommaList .  var Vl : ValueList .
(308)
(309)   eq k(exp(+(El)) -> K) = k(exp(El) -> + -> K) .
(310)   eq k(val(V) -> + -> K) = k(val(V) -> K) .
(311)   eq k(val(int(I),int(I'),Vl) -> + -> K) = k(val(int(I + I'),Vl) -> + -> K) .
(312)   eq k(exp(*(El)) -> K) = k(exp(El) -> * -> K) .
```

```
(313)   eq k(val(V) -> * -> K) = k(val(V) -> K) .
(314)   eq k(val(int(I),int(I'),Vl) -> * -> K) = k(val(int(I * I'),Vl) -> * -> K) .
(315)
(316)   eq k(exp(negate(E)) -> K) = k(exp(E) -> negate -> K) .
(317)   eq k(val(int(I)) -> negate -> K) = k(val(int(I * -1)) -> K) .
(318)
(319)   ops lt gt eq : -> ContinuationItem .
(320)   ops and or not : -> ContinuationItem .
(321)
(322)   eq k(exp(not(E)) -> K) = k(exp(E) -> not -> K) .
(323)   eq k(exp(and(E,E')) -> K) = k(exp(E,E') -> and -> K) .
(324)   eq k(exp(or(E,E')) -> K) = k(exp(not(and(not(E),not(E')))) -> K) .
(325)   eq k(val(True) -> not -> K) = k(val(False) -> K) .
(326)   eq k(val(False) -> not -> K) = k(val(True) -> K) .
(327)   eq k(val(True, True) -> and -> K) = k(val(True) -> K) .
(328)   eq k(val(True, False) -> and -> K) = k(val(False) -> K) .
(329)   eq k(val(False, True) -> and -> K) = k(val(False) -> K) .
(330)   eq k(val(False, False) -> and -> K) = k(val(False) -> K) .
(331)
(332)   eq k(exp(<(E,E')) -> K) = k(exp(E,E') -> lt -> K) .
(333)   eq k(exp(>(E,E')) -> K) = k(exp(E,E') -> gt -> K) .
(334)   eq k(exp(<=(E,E')) -> K) = k(exp(E,E') -> gt -> not -> K) .
(335)   eq k(exp(>=(E,E')) -> K) = k(exp(E,E') -> lt -> not -> K) .
(336)   eq k(exp(==(E,E')) -> K) = k(exp(E,E') -> eq -> K) .
(337)   eq k(exp(/=(E,E')) -> K) = k(exp(E,E') -> eq -> not -> K) .
(338)   eq k(val(int(I),int(I')) -> lt -> K) = k(val(if I < I' then True else False fi) -> K) .
(339)   eq k(val(int(I),int(I')) -> gt -> K) = k(val(if I > I' then True else False fi) -> K) .
(340)   eq k(val(int(I),int(I')) -> eq -> K) = k(val(if I == I' then True else False fi) -> K) .
(341)
(342)   op if : Exp Exp -> ContinuationItem .
(343)
(344)   eq k(exp(if(C,E,E')) -> K) = k(exp(C) -> if(E,E') -> K) .
(345)   eq k(val(True) -> if(E,E') -> K) = k(exp(E) -> K) .
(346)   eq k(val(False) -> if(E,E') -> K) = k(exp(E') -> K) .
(347) endfm
(348)
(349) fmod FUNCTION-SEMANTICS is
(350)   including FUNCTION-SYNTAX .
(351)   including GENERIC-EXP-SEMANTICS .
(352)   including CASE-SYNTAX .
```

```
(353)   including DATA-SYNTAX .
(354)
(355)   op makeEnvStore : HaskState -> HaskStateAttribute .
(356)   op saved : StmtList -> HaskStateAttribute .
(357)   op stmtpre : StmtList -> HaskStateAttribute .
(358)   op stmtprebind : StmtList -> HaskStateAttribute .
(359)   op free : Name Exp -> Exp .
(360)   op freen : Name -> Exp .
(361)   op freec : Name -> Exp .
(362)   op maketuple : Exp -> Exp .
(363)   op maketuple2 : Exp -> ExpCommaList .
(364)
(365)   var Env : Env .  var Mem : Store .  var S : Stmt .  vars Sl Cl : StmtList .
(366)   vars X Y : Name .  vars E E' P P' : Exp .  var N : Nat .  var K : Continuation .
(367)   var ECl : ExpCommaList .  var St : HaskState .
(368)
(369)   eq stmtpre(Sl) = stmtprebind(Sl) saved(Sl) .
(370)   eq stmtprebind(nil) saved(Sl) = stmt(Sl) .
(371)   eq stmtprebind(data X = E ; Sl) = stmtprebind(Sl) .
(372)   eq stmtprebind(X P = E ; Sl) nextloc(N) env(Env) =
(373)      stmtprebind(Sl) nextloc(N + 1) env(Env[X <- loc(N)]) .
(374)   eq stmtprebind(X = E ; Sl) nextloc(N) env(Env) =
(375)      stmtprebind(Sl) nextloc(N + 1) env(Env[X <- loc(N)]) .
(376)
(377)   eq maketuple2(E' E) = maketuple2(E'), E .
(378)   eq maketuple2(E) = E .
(379)   eq maketuple(E' E) =  Tuple (maketuple2(E'), E) .
(380)   eq maketuple(E) = Tuple (E) .
(381)
(382)   eq makeEnvStore(stmt(nil) env(Env) mem(Mem) nextloc(N)) = env(Env) mem(Mem) nextloc(N) .
(383)
(384)   eq stmt(X P = (case($) of Cl) ; X P' = E' ; Sl) =
(385)      stmt(X P = (case($) of (Cl ; maketuple(P') -> E')) ; Sl) .
(386)   eq stmt(X P = (case($) of Cl) ; Sl) = stmt(free(X,P) = (case($) of Cl) ; Sl) .
(387)   eq stmt(X P = E ; Sl) = stmt(X P = (case($) of (maketuple(P) -> E)) ; Sl) .
(388)   eq makeEnvStore(St stmt(free(X,(P E)) = E' ; Sl)) nextvar(N) =
(389)      makeEnvStore(St stmt(free(X,P) fv(N) = E' ; Sl)) nextvar(N + 1) .
(390)   eq makeEnvStore(St stmt(free(X,E) = E' ; Sl)) nextvar(N) =
(391)      makeEnvStore(St stmt(freen(X) fv(N) = E' ; Sl)) nextvar(N + 1) .
(392)   eq makeEnvStore(St stmt(free(X,(P E)) P' = E' ; Sl)) nextvar(N) =
```

```
(393)       makeEnvStore(St stmt(free(X,P) (P' fv(N)) = E' ; Sl)) nextvar(N + 1) .
(394)    eq makeEnvStore(St stmt(free(X,E) P' = E' ; Sl)) nextvar(N) =
(395)       makeEnvStore(St stmt(freen(X) (P' fv(N)) = E' ; Sl)) nextvar(N + 1) .
(396)    eq stmt(freen(X) P = (case($) of Cl) ; Sl) = stmt(freec(X) P = (case(maketuple(P)) of Cl) ; Sl) .
(397)    eq stmt(freec(X) (P Y) = E' ; Sl) = stmt(freec(X) P = \ Y -> E' ; Sl) .
(398)    eq stmt(freec(X) Y = E' ; Sl) = stmt(X = \ Y -> E' ; Sl) .
(399)
(400)    eq makeEnvStore(stmt(X = E ; Sl) env(Env) mem(Mem) St) =
(401)       makeEnvStore(stmt(Sl) env(Env) mem(Mem[Env[X] <- frozen(E,Env,Env[X])]) St) .
(402)
(403)    eq k(exp(\ P -> E) -> K) env(Env) = k(val(closure(P, E, Env)) -> K) env(Env) .
(404) endfm
(405)
(406) fmod LET-SEMANTICS is
(407)    including LET-SYNTAX .
(408)    including GENERIC-EXP-SEMANTICS .
(409)    including FUNCTION-SEMANTICS .
(410)
(411)    op makeBinds : StmtList -> ContinuationItem .
(412)    op addToEnv : StmtList -> ContinuationItem .
(413)
(414)    var E : Exp .  var N : Nat .  var Env : Env .  var Mem : Store .
(415)    var LL : StmtList .  var K : Continuation .
(416)
(417)    eq k(exp(let LL in E) -> K) env(Env) mem(Mem) nextloc(N) =
(418)       k(exp(E) -> recover(Env) -> K) makeEnvStore(env(Env) mem(Mem) nextloc(N) stmtpre(LL)) .
(419) endfm
(420)
(421) fmod CASE-SEMANTICS is
(422)    including CASE-SYNTAX .
(423)    including FUNCTION-SEMANTICS .
(424)
(425)    op caseof : Exp ExpCommaList ExpCommaList -> Exp .
(426)    op split : Exp StmtList ExpCommaList ExpCommaList -> Exp .
(427)    op cand : -> ContinuationItem .
(428)    op endcase : Exp -> Continuation .
(429)    op casegood : -> Continuation .
(430)    op casebad : -> ContinuationItem .
(431)    op good : -> ContinuationItem .
(432)    op matchbind : Exp -> ContinuationItem .
```

```
(433)   op makeande : LocationList ExpCommaList -> ContinuationItem .
(434)   op checkcase : Exp -> ContinuationItem .
(435)   op checkcasel : Location -> ContinuationItem .
(436)
(437)   op case : Exp ExpCommaList ExpCommaList Env Continuation -> ContinuationItem .
(438)   op case : Env Continuation -> ContinuationItem .
(439)
(440)   vars K C : Continuation .  vars E E' E'' XE : Exp .  vars N N' : Nat .  var Sl : StmtList .
(441)   vars ECl ECl' : ExpCommaList .  vars X X' : Name .  var V : Value .  vars I I' : Int .
(442)   vars Ll Ll' : LocationList .  var L : Location .  vars Env Env' : Env .
(443)   var CI : ContinuationItem .  var Mem : Store .
(444)
(445)   eq k(exp(case E of (E' -> E'' ; Sl)) -> K) = k(exp(split(E, Sl, E', E''')) -> K) .
(446)
(447)   eq split(XE, nil , ECl , ECl') = caseof(XE, ECl , ECl') .
(448)   eq split(XE, (E -> E' ; Sl) , ECl , ECl') = split(XE, Sl , (ECl, E) , (ECl', E')) .
(449)
(450)   eq k(exp(caseof(XE, ECl, ECl')) -> K) env(Env) c(C) =
(451)      k(endcase(0)) env(Env) c(case(XE, ECl, ECl', Env, K) -> C) .
(452)
(453)   eq k(endcase(E''')) c(case(XE, (E,ECl), (E',ECl'), Env, K) -> C) env(Env') =
(454)      k(checkcase(XE) -> matchbind(E) -> endcase(E')) c(case(XE, ECl, ECl', Env, K) -> C) env(Env) .
(455)   eq k(endcase(E''')) c(case(XE, E, E', Env, K) -> C) env(Env') =
(456)      k(checkcase(XE) -> matchbind(E) -> endcase(E')) c(case(Env, K) -> C) env(Env) .
(457)   eq k(endcase(E)) c(case(Env, K) -> C) env(Env') = k(val(bottom) -> K) c(C) env(Env) .
(458)
(459)   eq k(exp(nomatch) -> casegood) = k(endcase(nomatch)) .
(460)   eq k(exp(nomatch) -> CI -> K) = k(exp(nomatch) -> K) .
(461)
(462)   eq k(good -> cand -> K) = k(K) .
(463)   eq k(good -> endcase(E)) = k(exp(E) -> casegood) .
(464)   eq k(val(V) -> casegood) c(case(XE, ECl, ECl', Env, K) -> C) env(Env') =
(465)      k(val(V) -> K) c(C) env(Env) .
(466)   eq k(val(V) -> casegood) c(case(Env, K) -> C) env(Env') = k(val(V) -> K) c(C) env(Env) .
(467)
(468)   eq k(casebad -> CI -> K) = k(casebad -> K) .
(469)   eq k(casebad -> endcase(E)) = k(endcase(E)) .
(470)
(471)   eq k(checkcase(E) -> matchbind($) -> K) = k(good -> K) .
(472)   eq k(checkcase(X) -> matchbind(X') -> K) env(Env) = k(good -> K) env(Env[X' <- Env[X]]) .
```

```
(473)   eq k(checkcase(E) -> matchbind(X) -> K) env(Env) mem(Mem) nextloc(N) =
(474)      k(good -> K) env(Env[X <- loc(N)]) mem(Mem[loc(N) <- frozen(E, Env, loc(N))]) nextloc(N + 1) .
(475)   eq k(checkcase(E) -> K) env(Env) = k(exp(E) -> K) env(Env) [owise] .
(476)
(477)   eq k(checkcasel(L) -> matchbind($) -> K) = k(good -> K) .
(478)   eq k(checkcasel(L) -> matchbind(X) -> K) env(Env) = k(good -> K) env(Env[X <- L]) .
(479)   eq k(checkcasel(L) -> K) env(Env) = k(lexp(L) -> K) env(Env) [owise] .
(480)
(481)   eq k(val(V) -> matchbind($) -> K) = k(good -> K) .
(482)   eq k(val(()) -> matchbind(()) -> K) = k(good -> K) .
(483)   eq k(val(V) -> matchbind(X) -> K) = k(val(V) -> bindTo(X) -> good -> K) .
(484)
(485)   eq k(val(int(I)) -> matchbind(I) -> K) = k(good -> K) .
(486)  ceq k(val(int(I)) -> matchbind(I') -> K) = k(casebad -> K)
(487)   if I =/= I' .
(488)   eq k(val(construct('Tuple, Ll)) -> matchbind(Tuple(ECl)) -> K) = k(makeande(Ll, ECl) -> K) .
(489)   eq k(val(construct('Tuple, Ll)) -> matchbind(E) -> K) = k(casebad -> K) .
(490)   eq k(val(V) -> matchbind(Tuple(ECl)) -> K) = k(casebad -> K) .
(491)   eq k(val(construct(X,Ll)) -> matchbind(dt(X,ECl)) -> K) = k(makeande(Ll, ECl) -> K) .
(492)  ceq k(val(construct(X,Ll)) -> matchbind(dt(X',ECl)) -> K) = k(casebad -> K)
(493)   if X =/= X' .
(494)
(495)   eq k(makeande(L, $) -> K) = k(good -> K) .
(496)   eq k(makeande((Ll,L), (ECl, $)) -> K) = k(makeande(Ll, ECl) -> K) .
(497)   eq k(makeande(L, E) -> K) = k(checkcasel(L) -> matchbind(E) -> K) .
(498)   eq k(makeande((Ll, L), (ECl, E)) -> K) =
(499)      k(makeande(Ll, ECl) -> cand -> checkcasel(L) -> matchbind(E) -> K) .
(500) endfm
(501)
(502) fmod DATA-SEMANTICS is
(503)   including FUNCTION-SEMANTICS .
(504)   including DATA-SYNTAX .
(505)
(506)   vars C X : Name .  vars E : Exp .  var Sl : StmtList .
(507)   var Env : Env .  var Mem : Store .  var N : Nat .
(508)   var Data : Data .  var ECl : ExpCommaList .
(509)   var K : Continuation .  vars L L' : LocationList .
(510)   vars Vl Vl' : ValueList .
(511)
(512)   eq makeEnvStore(stmt(data X = dt(C,ECl) || E ; Sl) env(Env) mem(Mem) nextloc(N)) datas(Data) =
```

```
(513)      makeEnvStore(stmt(data X = E ; Sl) env(Env) mem(Mem) nextloc(N)) datas(Data[X,C,ECl]) .
(514)   eq makeEnvStore(stmt(data X = dt(C,ECl) ; Sl) env(Env) mem(Mem) nextloc(N)) datas(Data) =
(515)      makeEnvStore(stmt(Sl) env(Env) mem(Mem) nextloc(N)) datas(Data[X,C,ECl]) .
(516)
(517)   op construct : Name -> ContinuationItem .
(518)   op locs : LocationList -> ContinuationItem .
(519)   op savet : ExpCommaList -> ContinuationItem .
(520)
(521)   eq k(exp(dt(C,ECl)) -> K) = k(savet(ECl) -> construct(C) -> K) .
(522)   eq k(exp(Tuple(ECl)) -> K) = k(savet(ECl) -> construct('Tuple) -> K) .
(523)   eq k(savet(E, ECl) -> K) = k(savet(E) -> savet(ECl) -> K) .
(524)   eq k(locs(L) -> savet(ECl) -> K) = k(savet(ECl) -> locs(L) -> K) .
(525)   eq k(locs(L') -> locs(L) -> K) = k(locs(L,L') -> K) .
(526)   eq k(savet(E) -> K) env(Env) mem(Mem) nextloc(N) =
(527)      k(locs(loc(N)) -> K) env(Env) mem(Mem[loc(N) <- frozen(E,Env,loc(N))]) nextloc(N + 1) .
(528)
(529)   eq k(locs(L) -> construct(C) -> K) = k(val(construct(C,L)) -> K) .
(530)
(531)   eq result(val(Vl') -> val(cv(C, Vl)) -> K) = result(val(cv(C, (Vl, Vl'))) -> K) .
(532) endfm
(533)
(534) fmod HASK-SEMANTICS is
(535)   including HASKELL-SYNTAX .
(536)   including HASK-STATE .
(537)   including MATH-SEMANTICS .
(538)   including FUNCTION-SEMANTICS .
(539)   including LET-SEMANTICS .
(540)   including CASE-SEMANTICS .
(541)   including DATA-SEMANTICS .
(542)
(543)   ops eval eval* eval** : StmtList Exp -> ValueList .
(544)   op [_] : HaskState -> ValueList .
(545)   ops pause deeper : -> ContinuationItem .
(546)
(547)   var Sl : StmtList .  var E : Exp .  vars V V' : Value .  vars Vl Vl' : ValueList .
(548)   var Env : Env .  var Mem : Store .  vars N N' : Nat .
(549)   var D : Data .  var C : Name .  var Ll : LocationList .
(550)   vars K K' : Continuation .
(551)
(552)   eq eval(Sl, E) = [k(exp(E) -> stop) c(stop) makeEnvStore(stmtpre(Sl) env(empty)
```

```
(553)       mem([loc(0),static(())]) nextloc(1)) datas(empty) nextvar(0)] .
(554)
(555)   eq [k(val(Vl) -> stop) env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N')] = Vl .
(556)
(557)   eq eval*(Sl, E) = [k(exp(E) -> stop) c(stop) makeEnvStore(stmtpre(Sl) env(empty)
(558)       mem([loc(0),static(())]) nextloc(1)) datas(empty) nextvar(0) result(stop)] .
(559)   eq eval**(Sl, E) = makelist([k(exp(E) -> stop) c(stop) makeEnvStore(stmtpre(Sl) env(empty)
(560)       mem([loc(0),static(())]) nextloc(1)) datas(empty) nextvar(0) result(stop)]) .
(561)
(562)   eq [k(val(construct(C, Ll)) -> stop) env(Env) c(stop) mem(Mem)
(563)       nextloc(N) datas(D) nextvar(N') result(K)] =
(564)      [k(lexp(Ll) -> deeper -> stop) env(Env) c(stop) mem(Mem)
(565)       nextloc(N) datas(D) nextvar(N') result(val(cv(C,nil)) -> deeper -> K)] .
(566)   eq [k(val(construct(C, Ll)) -> pause -> K') env(Env) c(stop) mem(Mem)
(567)       nextloc(N) datas(D) nextvar(N') result(K)] =
(568)      [k(lexp(Ll) -> deeper -> pause -> K') env(Env) c(stop) mem(Mem)
(569)       nextloc(N) datas(D) nextvar(N') result(val(cv(C,nil)) -> deeper -> K)] .
(570)   eq [k(val(construct(C, Ll)) -> deeper -> K') env(Env) c(stop) mem(Mem)
(571)       nextloc(N) datas(D) nextvar(N') result(K)] =
(572)      [k(lexp(Ll) -> deeper -> deeper -> K') env(Env) c(stop) mem(Mem)
(573)       nextloc(N) datas(D) nextvar(N') result(val(cv(C,nil)) -> deeper -> K)] .
(574)   eq [k(val(construct(C, Ll),Vl) -> stop) env(Env) c(stop) mem(Mem)
(575)       nextloc(N) datas(D) nextvar(N') result(K)] =
(576)      [k(lexp(Ll) -> deeper -> pause -> val(Vl) -> stop) env(Env) c(stop) mem(Mem)
(577)       nextloc(N) datas(D) nextvar(N') result(val(cv(C,nil)) -> deeper -> pause -> K)] .
(578)   eq [k(val(construct(C, Ll),Vl) -> pause -> K') env(Env) c(stop) mem(Mem)
(579)       nextloc(N) datas(D) nextvar(N') result(K)] =
(580)      [k(lexp(Ll) -> deeper -> pause -> val(Vl) -> pause -> K') env(Env) c(stop) mem(Mem)
(581)       nextloc(N) datas(D) nextvar(N') result(val(cv(C,nil)) -> deeper -> pause -> K)] .
(582)   eq [k(val(construct(C, Ll),Vl) -> deeper -> K') env(Env) c(stop) mem(Mem)
(583)       nextloc(N) datas(D) nextvar(N') result(K)] =
(584)      [k(lexp(Ll) -> deeper -> pause -> val(Vl) -> deeper -> K') env(Env) c(stop) mem(Mem)
(585)       nextloc(N) datas(D) nextvar(N') result(val(cv(C,nil)) -> deeper -> pause -> K)] .
(586)
(587)   eq [k(val(V) -> stop) env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(K)] =
(588)      [k(stop) env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(val(V) -> K)] .
(589)   eq [k(val(V) -> pause -> K') env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(K)] =
(590)      [k(pause -> K') env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(val(V) -> K)] .
(591)   eq [k(val(V) -> deeper -> K') env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(K)] =
(592)      [k(deeper -> K') env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(val(V) -> K)] .
```

```
(593)   eq [k(val(V,Vl) -> stop) env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(K)] =
(594)      [k(val(Vl) -> stop) env(Env) c(stop) mem(Mem)
(595)       nextloc(N) datas(D) nextvar(N') result(val(V) -> K)] .
(596)   eq [k(val(V,Vl) -> pause -> K') env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(K)] =
(597)      [k(val(Vl) -> pause -> K') env(Env) c(stop) mem(Mem)
(598)       nextloc(N) datas(D) nextvar(N') result(val(V) -> K)] .
(599)   eq [k(val(V,Vl) -> deeper -> K') env(Env) c(stop) mem(Mem)
(600)       nextloc(N) datas(D) nextvar(N') result(K)] =
(601)      [k(val(Vl) -> deeper -> K') env(Env) c(stop) mem(Mem)
(602)       nextloc(N) datas(D) nextvar(N') result(val(V) -> K)] .
(603)
(604)   eq [k(pause -> K') env(Env) c(stop) mem(Mem) nextloc(N)
(605)       datas(D) nextvar(N') result(val(Vl) -> pause -> K)] =
(606)      [k(K') env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(val(Vl) -> K)] .
(607)   eq [k(deeper -> K') env(Env) c(stop) mem(Mem) nextloc(N)
(608)       datas(D) nextvar(N') result(val(Vl) -> deeper -> K)] =
(609)      [k(K') env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(val(Vl) -> K)] .
(610)
(611)   eq [k(stop) env(Env) c(stop) mem(Mem) nextloc(N) datas(D) nextvar(N') result(val(Vl) -> stop)] = Vl .
(612)
(613)   sort List .
(614)   subsort List < Value .
(615)   op makelist : Value -> List .
(616)   op makelist2 : Value -> Value .
(617)   op [_] : ValueList -> List .
(618)   op _++_ : List List -> List [assoc] .
(619)
(620)   eq makelist(cv('Nil, ())) = [nil] .
(621)   eq makelist(cv('Cons, (V,V'))) = [makelist(V)] ++ makelist(V') .
(622)   eq makelist(V) = V [owise] .
(623)   eq [Vl] ++ [Vl'] = [Vl, Vl'] .
(624) endfm
```

References

[1] Haskell 98 Language, http://www.haskell.org/onlinereport/

[2] The Maude System, http://maude.cs.uiuc.edu/

[3] Example Taken from Grigore Rosu's CS 522 Class Slides, http://fsl.cs.uiuc.edu/~grosu/classes/2006/spring/cs522/t08.pdf

[4] Hugs 98, http://www.haskell.org/hugs/

[5] The Glasgow Haskell Compiler, http://www.haskell.org/ghc/

[6] Formal Systems Laboratory, http://fsl.cs.uiuc.edu/

[7] Modular Syntax Definition Formalism, http://www.program-transformation.org/Sdf/WebHome

[8] P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. Technical report, Department of Computer Science, Universiteit Utrecht, 1999.