3161 06s1 Assignment 1  
MinHs  
Draft 1.0  
March 30, 2006  

Marks:  
30% of the classmark for 3161 students  
35% of the classmark for 9161 students  

Due date:  
Mon May 1 07:59:59 EST  

History  
Mar 30  Initial version released  

Overview  
In this assignment you will implement an interpreter for MinHs, a small functional  
language similar to ML and Haskell. It is fully typed, with types specified by the  
programmer. This is a strict language (unlike Haskell), and is based on the language  
MinML used in Harper [4]. The assignment consists of a compulsory component, and  
some bonus exercises.  

• Task 1 (100%)  
Implement an interpreter for the MinHs language presented in the lectures, using  
an environment semantics.  

• Bonus 1 (10% bonus mark)  
Extend the interpreter to handle functions of more than 1 argument.  

• Bonus 2 (5% bonus mark)  
Implement constant value pattern matching on function arguments.  

• Bonus 3 (5% bonus mark)  
Add support for mutually recursive let bindings.
Each of these parts is explained in detail below.

The front end of the interpreter (lexer, parser, type checker) is provided for you, along with the type of the `evaluate` function (found in the file `Eval.hs`). `evaluate` returns an object of type `Value`. You may modify the constructors for `Value` if you wish, but not the type for `evaluate`. The return value of `evaluate` is used to check the correctness of your assignment.

You must provide an implementation of `evaluate`, in `Eval.hs`. It is this file you will submit for Task 1. No other files can be modified. If you complete any of the bonus tasks, you may have to submit other files you modified. Submission guidelines will be provided closer to the due date.

You can assume the typechecker has done its job and will only give you correct programs to evaluate. The type checker will, in general, rule out incorrect programs, so the interpreter does not have to consider them. We also maintain the assumption that all variable and function names are unique.

Please consult the FAQ on the class message board, and send bug reports about the assignment specification to dons.

1 Task 1

This is the core part of the assignment. You are to implement an interpreter for MinHs. The following expressions must be handled:

- variables. $v, u$
- integer constants. $1, 2, \ldots$
- boolean constants. $True, False$
- some primitive arithmetic and boolean operations. $+, *, <, \leq, \ldots$
- function application. $f \, x$
- $if \, e \, then \, e_1 \, else \, e_2$
- $let \, x = e_1 \, in \, e_2$
- $let\, fun \, f :: (\tau_1 \rightarrow \tau_2) \, x = e \, expressions$

These cases are explained in detail below. The abstract syntax defining these syntactic entities is in `Syntax.hs`. You should understand the data type `Exp` and `Bind` well.

Your implementation is to follow the dynamic semantics described in the lectures, and this document. You are not to use substitution as the evaluation strategy, but must use an environment/heap semantics. If a runtime error occurs, which is possible, you should use Haskell’s `error :: String \rightarrow a` function to emit a suitable error message (the error code returned by `error` is non-zero, which is what will be checked for – the actual error message is not important).
1.1 Program structure

A program in MinHs may evaluate to either an integer or a boolean, depending on the type assigned to the main function. The main function is always defined (this is checked by the type checker). In Task 1 programs, you need only consider the case of a single top-level binding for main, like so:

\[
\text{main :: Int} = 1 + 2;
\]

or

\[
\text{main :: Bool}
\]
\[
\text{= let x :: Int} = 1;
\]
\[
\text{in if x + ((letfun f :: (Int \to Int) y = y \ast y) 2) == 0}
\]
\[
\text{then True}
\]
\[
\text{else False;}
\]

1.2 Variables, Literals and Constants

MinHs is a spartan language. We only have to consider 3 types:

\[
\text{Int}
\]
\[
\text{Bool}
\]
\[
\text{t1 \to t2}
\]

The only literals you will encounter are integers. The only non-literal constructors are True and False for the Bool type.

1.3 Function application

MinHs is a strict language. This is specified in the dynamic semantics. Evaluation proceeds from left to right. An argument to a function is fully evaluated before calling the function with that argument. The result of a function application may in turn be a function.

1.4 Primitive operations

You need to implement the following primitive operations:

\[
+ :: \text{Int} \to \text{Int} \to \text{Int}
\]
\[
- :: \text{Int} \to \text{Int} \to \text{Int}
\]
\[
* :: \text{Int} \to \text{Int} \to \text{Int}
\]
\[
/ :: \text{Int} \to \text{Int} \to \text{Int}
\]
\[
\text{negate} :: \text{Int} \to \text{Int}
\]
\[
> :: \text{Int} \to \text{Int} \to \text{Bool}
\]
\[
>= :: \text{Int} \to \text{Int} \to \text{Bool}
\]
<    :: Int -> Int -> Bool
<=   :: Int -> Int -> Bool
==   :: Int -> Int -> Bool
/=   :: Int -> Int -> Bool

These operations are defined over Ints and Bools, as usual. negate is the primop representation of the unary negation function, i.e. \(-1\). The abstract syntax for primops is defined in PrimOp.hs.

1.5 if-then-else

MinHs has an if \(e_1\) then \(e_2\) else \(e_3\) construct. The types of \(e_1\) and \(e_2\) are the same. The type of \(e\) is Bool.

1.6 let

For the first task you only need to handle simple let s of the kind we have discussed in the lectures. Like these:

\[
\text{main :: Int} \\
\quad = \text{let} \\
\qquad \begin{array}{l}
\text{x :: Int} = 1 + 2; \\
\text{in x;}
\end{array}
\]

or

\[
\text{main :: Int} \\
\quad = \text{let f :: (Int -> Int)} \\
\qquad = \text{letfun f :: (Int -> Int) x = x + x; in f 3;}
\]

You do not need to handle let bindings of more than one variable at a time (as is possible in Haskell). Remember, a let may bind a recursive function defined with letfun.

1.7 letfun

The letfun expression introduces a new, named function value. It has the form:

\[
(\text{letfun f :: (Int -> Bool) x = x + x})
\]

A letfun value is a first-class value, and may be bound to a variable with let. The value \('f'\) is bound in the body of the function, so it is possible to write recursive functions:

\[
\text{letfun f :: (Int -> Int) x =} \\
\qquad \text{if x < 10 then f (x+1) else x}
\]
You can consult chapter 11.2 of the textbook for a discussion of some of the problems of using environments in a language allowing functions to be returned by functions.

1.8 Evaluation strategy

We have seen in the tutorials how it is possible to evaluate expressions via substitution. This is an extremely inefficient way to run a program. In this assignment you are to use an environment instead. You will be penalised for an interpreter that operates via substitution. The module Env.hs provides a data type suitable for most uses. Consult Harper [4] on how environments are to be used in dynamic semantics. The strategy is to bind variables to values in the environment, and look them up when required.

In general, you will need to use: newEnv, getEnv and union to begin with an empty environment, lookup the environment, or to add a binding to the environment, respectively.

2 Dynamic Semantics of MinHs

Structured operational semantics

- **Initial states:** all well-typed expressions, empty environment
- **Final states:** boolean and integer machine values

Machine values

<table>
<thead>
<tr>
<th>n value</th>
<th>b value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>b</td>
</tr>
</tbody>
</table>

Environment

The environment $\Gamma$ maps variables to values (because we’re in a call-by-value language), and is used in place of substitution. In the following rules we assume the environment, and only specify it explicitly when it is accessed. It is specified as follows:

$$\Gamma ::= \cdot \mid \Gamma, x = v$$

Values bound in the environment are closed – they contain no free variables. This requirement creates a problem with function values created with letfun whose bodies contain variables bound in an outer scope. We must bundle them with their associated environment as a closure.

Constants

| num(n) $\mapsto$ n | bool(b) $\mapsto$ b |
Primitive operations

\[
e_1 \mapsto e'_1 \\
\text{plus}(e_1, e_2) \mapsto \text{plus}(e'_1, e_2) \\
e_2 \mapsto e'_2 \\
\text{plus}(\text{num}(n), e_2) \mapsto \text{plus}(\text{num}(n), e'_2) \\
\text{plus}(\text{num}(n), \text{num}(m)) \mapsto \text{num}(n + m)
\]

Similarly for the other operations (as for the language of arithmetic expressions, and in chapters 7, 8 of Harper [4]). Note the rule for division by zero:

\[
div(\text{num}(n), \text{num}(0)) \mapsto \text{error}(\text{"divide by zero"})
\]

Evaluation of if-expression

\[
e_1 \mapsto e'_1 \\
\text{if}(e_1, e_2, e_3) \mapsto \text{if}(e'_1, e_2, e_3)
\]

\[
\text{if(const(true),}e_1, e_2) \mapsto e_1 \\
\text{if(const(false),}e_1, e_2) \mapsto e_2
\]

Variables

\[
\Gamma(x) = v \\
\Gamma \vdash \text{var}(x) \mapsto v
\]

Variable Bindings with Let

\[
e_1 \mapsto e'_1 \\
\text{let}(e_1, x.e_2) \mapsto \text{let}(e'_1, x.e_2)
\]

\[
\Gamma \vdash \text{let}(v, x.e) \mapsto \Gamma, x = v \vdash e
\]

Function values

To maintain soundness with function values, we need to pair a function with its environment, forming a closure. We add a new value form (the types are included for completeness, but are not required at runtime):

\[
\langle\langle \Gamma; \text{letfun}(\tau_1, \tau_2, f.x.e) \rangle\rangle \text{ value}
\]

Now we can specify how to introduce closed function values:

\[
\Gamma \vdash \text{letfun}(\tau_1, \tau_2, f.x.e) \mapsto \langle\langle \Gamma; \text{letfun}(\tau_1, \tau_2, f.x.e_1) \rangle\rangle
\]
Function Application

\[
\begin{align*}
    e_1 & \mapsto e'_1 \\
    \text{app}(e_1, e_2) & \mapsto \text{app}(e'_1, e_2) \\
    e_2 & \mapsto e'_2 \\
    \text{app}(v, e_2) & \mapsto \text{app}(v, e'_2)
\end{align*}
\]

\[
\Gamma \vdash \text{app}((\Gamma'; \text{letfun}(\tau_1, \tau_2, f.x.e_1)), v) \mapsto \Gamma', f = (\Gamma'; \text{letfun}(\ldots)), x = v \vdash e_1
\]

3 Bonus 1

Bonus 1 is optional, and worth 10% extra. In this task you are to implement \(n\)-ary functions.

3.1 \(n\)-ary functions

You should modify the interpreter to handle bindings of functions of more than 1 argument.

```haskell
main :: Bool
  = let eq :: (Int -> Int -> Bool)
     = letfun sum :: (Int -> Int -> Bool) x y = x == y;
      in eq 3 4;
```

We haven’t discussed the semantics of such functions in the lectures so you will need to work out a reasonable dynamic semantics for \(n\)-ary functions on your own, based on the semantics for unary functions.

4 Bonus 2

Bonus 2 is also optional, and is worth a bonus 5%. In this task you are to implement simple constant patterns in place of variable bindings in function definitions. Like so:

```haskell
letfun fib :: (Int -> Int)
  0 = 0;
  1 = 1;
  n = fib (n - 2) + fib (n - 1);
```

This will require modifying the abstract syntax, the parser, and the type checker, to allow constants and constructors where previously only variable names were allowed, in bindings. You will also have to develop a semantics, both static and dynamic, for what constant patterns mean, and how you will evaluate them. No other help is provided, however the reference material for this course should be a good start. In particular, the Haskell Report describes a semantics for Haskell pattern matching, of which this is a tiny subset.
5 Bonus 3

A third bonus task is available. Currently bindings must be specified in dependency order. However, in Haskell, the order of declarations is irrelevant, which allows mutually recursive bindings. For a bonus 5% implement Haskell-style mutually recursive bindings, like so:

```haskell
main :: Int
    = letrec a :: Int = b;
       b :: Int = c;
       c :: Int = 7;
       in c + a;
```

6 Testing

Your assignments will be tested very rigorously: correctness is a theme of this subject, after all. You are encouraged to test yourself. minhs comes with a regress tester script, and you should add your own tests to this.

7 Building minhs

minhs (the compiler/interpreter) is written in Haskell, and requires GHC 6.2 or higher. All testing will occur on standard CSE Linux machines. Make sure you test your program on a CSE Linux machine. The README distributed with MinHs gives more advice on how to build and run the minhs program, and it is generally very simple:

- `./configure`
- using GNU make, type `make` to build the compiler
- `./minhs --help` will help you find any useful debugging options.

To run the interpreter:

```
$ ./minhs foo.mhs
```

You may wish to experiment with some of the debugging options to see, for example, how your program is parsed, and what abstract syntax is generated.

8 Late Penalty

Penalty for late submission of assignments will be 4% (of the worth of the assignment) subtracted from the raw mark per day of being late. In other words, earned marks will be lost. For example, assume an assignment worth 25 marks is marked as 20, but had been submitted two days late. The late penalty will be 2 marks, resulting in a mark of 18 being awarded. No assignments will be accepted later than one week after the deadline.
9 Plagiarism

Please note that we will check for plagiarism and read the item on the Yellow Form regarding the originality of Assignments. Assignments are individual assignments and teamwork is not permitted. Penalties include 0F for copying code, negative or 0 marks for the assignment in case of teamwork, depending on the severity.
10  Lexical Structure

The lexical structure of MinHS is a small subset of Haskell98. See section 2.2 of the Haskell98 report [1]. The lexical conventions are implemented in the module Lexer.x. This is an Alex [2] lexer specification.

11  Concrete syntax

The concrete syntax is based firstly on Haskell. It provides the usual arithmetic and boolean primitive operations (most of the Int-type primitive operations of GHC). It has conventional let bindings. At the outermost scope, the let is optional. As a result, multiple outer-level bindings are treated as nested let bindings down the page. It is required that a distinguished main function, of atomic type, exist. There is an if-then-else conditional expression. The primitive types of MinHS are Int and Bool. MinHS also implements, at least partially, a number of extensions to MinML: inline comments, $n$-ary functions, infix notation, more primitive numerical operations and a non-mutually recursive, simultaneous let declaration (treated as a nested-let). Function values may be specified with letfun.

The concrete syntax is described and implemented in the Parser.y module, a Happy [3] grammar.

Features of Haskell we do not provide:

- No nested comments
- No layout rule. Thus, semi-colons are required to terminate certain expressions. Consult the grammar.
- Precendence. There is currently no way to specify the precedence of operators. All binary symbol primops have the same precedence, so explicit parenthesis must be used to disambiguate.

12  Abstract syntax

The (first-order) abstract syntax is based closely on the MinHs syntax introduced in the lectures. It is implemented in the file Syntax.hs. Extensions to the MinHs abstract syntax take their cue from the Haskell kernel language. Presented below is the abstract syntax, with smatterings of concrete syntax for clarity.

13  Static semantics

The static semantics are based on those of the lecture, and of and MinML, from Bob Harper’s book. They are implemented by the module TyCheck.hs.

13.1  $n$-ary functions

Functions may be declared to take more than 1 argument at a time.
### Types
\[ \tau \rightarrow \text{Int} | \text{Bool} | \tau \rightarrow \tau \]

### Literals
\[ n \rightarrow \ldots | 0 | 1 | 2 | \ldots \]
\[ b \rightarrow \text{True} | \text{False} \]

### Primops
\[ o \rightarrow + | - | * | / | > | >= | == | /= | < | <= \]

### Expressions
\[ exp \rightarrow \text{var}(x) \]
\[ \text{lit}(n) \]
\[ \text{con}(b) \]
\[ \text{apply}(e_1, e_2) \]
\[ \text{let}(\text{decl}, \text{exp}) \]
\[ \text{letfun}(\text{decl}) \]
\[ \text{if}(\text{exp}, \text{exp}_1, \text{exp}_2) \]

### Decl
\[ decl \rightarrow \text{fun}(f, \tau, [\text{arg}], e) \]
\[ \text{val}(v, \tau, e) \]

---

**Figure 1:** The expression abstract syntax of MinHS

---

### 14 Environments

*Environments* are required by typechecker and possibly by the interpreter. The typechecker needs to map variables to types, and the interpreter might need to map variables to functions or values (like a heap). This latter structure is used to provide a fast alternative to substitution.

We provide a general environment module, keyed by identifiers, in `Env.hs`. Environments are generally simpler in MinHS than in real Haskell. We still need to bind variables to partially evaluated functions, however.

### 15 Dynamic semantics

The dynamic semantics are described in this document, the lectures, and in Harper [4]. Implemented in the module `Eval.hs`. Consult Harper for the specification.

#### 15.1 Interpreter

The interpreter is the backend that runs by default. It should implement the dynamic semantics of MinML.
16 Interfaces

The basic types are found in:

- Ident.hs (code for handling identifiers)
- PrimOp.hs (handling primops)
- Type.hs (types, constructors, building and taking apart types)

Errors

There are a number of useful error functions. These can be found in Error.hs.

Printing

Most structures in MinHS need to be printed at some point. The easiest way to do this is to make that type an instance of class Pretty. See the bottom of Syntax.hs for an example.

Testing

make check TEST=[dir]

Check directories may have an optional 'Flag' file, containing flags you wish to pass to minhs in that directory, or the magic flag, 'expect-fail', which inverts the sense in which success is defined by the driver.

References


[2] The Alex Haskell lexer


