COMP3161/9164 23T3 Assignment 1

`hindsight`

Version 1.0.6

Marks : 17.5% of the mark for the course.

Due date: **Friday, Week 8, 1st of November 2024, 23:59:59 Sydney time**

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.

However, we will not evaluate MinHS directly; instead, we'll first compile it to an intermediate language we call `hindsight`. In `hindsight` we use neither call-by-value nor a call-by-name evaluation, but *call-by-push-value*. This means the programmer gets to decide the evaluation order herself with explicit operators to steer the control flow. Once we have implemented an evaluator for `hindsight`, we can then give MinHS either a call-by-value or a call-by-name evaluator, by going to `hindsight` via different compilation strategies.

The assignment consists of a base compulsory component, worth 70%, and four additional components which collectively are worth 50%, meaning that not all must be completed to earn full marks.

Your total mark can go up to 120%. Any marks above 100% will be converted to bonus exam marks, at a 20-to-3 exchange rate. For example, earning 110% on the assignment will yield 1.5 bonus marks on the final exam.

- **Task 1** (70%)
  Implement an interpreter for `hindsight`, using an environment semantics, including support for recursion and closures.

- **Task 2** (10%)
  Extend the interpreter to support partially applied primops.
- **Task 3** (10%)
  Extend the interpreter to support multiple bindings in the one `let` form.
- **Task 4** (10%)
  Implement an optimisation pass for `hindsight`.
- **Task 5** (20%)
  Implement a call-by-name compiler from MinHS to `hindsight`.

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 `Hindsight/Evaluator.hs`) and an implementation stub. The function *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 `Hindsight/Evaluator.hs`. It is this file you will submit for Task 1. The only other files that can be modified are `Hindsight/Optimiser.hs` (for Task 4) and `Hindsight/CBNCompile.hs` (for Task 5)

You can assume the typechecker has done its job and will only give you type-correct programs to evaluate. The type checker will, in general, rule out type-incorrect programs, so the interpreter does not have to consider them.

Please use the Ed forum for questions about this assignment.

Submission

Submit your (modified) `Hindsight/Evaluator.hs`, `Hindsight/Optimiser.hs` and `Hindsight/CBNCompile.hs` using the CSE give system, by typing the command

```
give cs3161 Eval Evaluator.hs Optimiser.hs CBNCompile.hs
```

or by using the CSE give web interface. Note that `Optimiser.hs` and `CBNCompile.hs` are optional, and should only be included if you completed the corresponding bonus tasks.

1 Primer on call-by-push-value

As mentioned, `hindsight` is a call-by-push-value language. The core of the language is similar to MinHS as seen in the lectures. This section will describe some of the key differences.

Following the call-by-push-value paradigm, `hindsight` distinguishes between two kinds of expressions: *value expressions* and *computation expressions*. A value expression denotes a value, and a computation expression denotes a process that might produce a value if we run it.

Computations can be suspended using the `thunk` operator, and suspended computations can be passed around as value expressions, and later resumed using `force`. Here's an example program:

```
main :: F Bool
  = let y :: U(F Bool) = thunk(1 == 2);
    in
      reduce 1 < 2
      to x in
        if x
          then produce True
          else force y
```

The type annotation $main :: F\ Bool$ means that $main$ is a computation expression which produces a boolean result. The U in $y :: U(F\ Bool)$ means that y is a *suspended* computation which, if resumed, would produce a boolean result. $reduce\ 1 < 2\ to\ x$ means that the computation $1 < 2$ is evaluated, producing a value which is saved in the local binding x. If the $True$ branch is chosen, we'll run the trivial computation $produce\ True$ which immediately produces a value $True$. Otherwise, we'll resume the suspended computation from before.

Thus, in this case the equality comparison $1 == 2$ is never evaluated. If we want the equality comparison to be evaluated first (despite the fact that we don't need its result), we can refrain from suspending it:

```
main :: F Bool
  = reduce 1 == 2
    to y in
      reduce 1 < 2
      to x in
        if x
           then produce True
           else produce y
```

2 Task 1

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

- variables. x, y, z
- integer constants. $1, 2, ..$
- boolean constants. $True, False$
- some primitive arithmetic and boolean operations. $+, *, <, <=, ..$
- constructors for lists. $Nil,\ Cons$
- destructors for lists. $head,\ tail$
- inspectors for lists. $null$
- function application. $f\ x$
- `if` v `then` c_1 `else` c_2
- suspending computations. `thunk` c
- resuming suspended computations. `force` v
- `let` $x :: \tau_x = v;$ `in` c
- `reduce` c_1 `to` x `in` c_2
- `produce` v
- `recfun` $f :: (\tau_1 \rightarrow \tau_2)\ x = c$ expressions

The conceptual meaning of these expressions is explained in detail below, and their semantics are specified more precisely in a big-step style in Section 3. The abstract syntax defining these syntactic entities is in `Hindsight/Syntax.hs`, which inherits some definitions from `MinHS/Syntax.hs` You should understand the `Hindsight` data types $VExp$, $CExp$, $CBind$ and $VBind$ well.

In the syntax above and elsewhere in this section, variables named v, v_1 etc represent value expressions, and variables named c, c_1 etc represent computation expressions. The types of the constructors of the $VExp$ and $CExp$ types also clarify this.

Your implementation is to follow the dynamic semantics described in 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 $error :: String \to a$ 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).

2.1 Program structure

A program in `hindsight` may evaluate to either an integer, a list of integers, or a boolean, depending on the type assigned to the $main$ function. The $main$ function is always defined (this is checked by the implementation). You need only consider the case of a single top-level binding for $main$, as e.g. here:

```
main :: F Int = 1 + 2
```

2.2 Variables, Literals and Constants

`hindsight` is a spartan language. We have to consider the following six forms of types:

```
Int
Bool
[Int]
U ct
F vt
vt -> ct
```

The first four are *value types*, and the latter two are *computation types*. We use vt to denote value types and ct to denote computation types.

Note the `Int` type of MinHS and `hindsight` denotes an unbounded precision integer, which is the same as the `Integer` type in Haskell. This is different to the `Int` type of Haskell, which is either a 32-bit or 64-bit integer depending on the platform.

The only literals you will encounter are integers. The only non-literal constructors are $True$ and $False$ for the $Bool$ type, and Nil and $Cons$ for the $[Int]$ type.

2.3 Function application

A function in `hindsight` accepts exactly one argument, which must be a value. The body of the function must be a computation. Inside the body of a recursive function $f :: vt -> ct$, any recursive references to f are considered suspended; that is, they are regarded as having type $f :: U(vt -> ct)$.

The result of a function application may in turn be a function.

2.4 Primitive operations

You need to implement the following primitive operations:

```
+        :: Int -> Int -> F Int
-        :: Int -> Int -> F Int
*        :: Int -> Int -> F Int
/        :: Int -> Int -> F Int
%        :: Int -> Int -> F Int

negate   :: Int -> F Int

>        :: Int -> Int -> F Bool
>=       :: Int -> Int -> F Bool
<        :: Int -> Int -> F Bool
<=       :: Int -> Int -> F Bool

==       :: Int -> Int -> F Bool
/=       :: Int -> Int -> F Bool

head :: [Int] -> F Int
tail :: [Int] -> F [Int]
null :: [Int] -> F Bool
```

These operations are defined over Ints, $[Int]$s, and $Bool$s, as usual. $negate$ is the primop representation of the unary negation function, i.e. $negate$ applied to `1` results in `-1`. The abstract syntax for primops is inherited from `MinHS/Syntax.hs`.

Note the `Int` type of MinHS and `hindsight` denotes an unbounded precision integer, which is the same as the `Integer` type in Haskell. This is different to the `Int` type of Haskell, which is either a 32-bit or 64-bit integer depending on the platform.

2.5 `if-then-else`

`hindsight` has an `if` v `then` c_1 `else` c_2 construct. The types of c_1 and c_2 are the same. The type of v is $Bool$.

2.6 `let`

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

```
main :: F Int
    = let
        x :: Int = 3;
      in produce x
```

or

```
main :: F Int
    = let f :: U (Int -> F Int)
            = thunk (recfun f :: (Int -> F Int) x = x + x);
      in force f 3
```

For the base component of the assignment, 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 (suspended) recursive function defined with `recfun`.

2.7 `force` and `thunk`

`thunk` c is a value expression called a *thunk* or a *suspended computation*. A suspended computation value v can be evaluated later in the computation expression `force` v.

2.8 `reduce`

`reduce` c_1 `to` x `in` c_2 is a computation which first executes c_1 until a value is produced. This value is then bound to the name x in the evaluation of c_2. It is similar to `let`, but instead of binding a value expression to a name, it binds the value produced by a computation expression to a name.

2.9 `recfun`

The `recfun` expression introduces a new, named function computation. It has the form:

```
(recfun f :: (Int -> F Int) x = x + x)
```

Unlike in Haskell (and MinHS), a `recfun` is *not* a value, but a computation. It can be bound in `let` expressions, but only if suspended by `thunk`. The value 'f' is bound in the body of the function, so it is possible to write recursive functions:

```
recfun f :: (Int -> F Int) x =
  reduce x < 10
  to b in
    if b then
      reduce x + 1
      to y in
        force f y
    else produce x
```

Note that inside the body of 'f', 'f' is considered suspended, hence `force` must be used to explicitly resume recursive calls.

Be very careful when implementing this construct, as there can be problems when using environments in a language allowing functions to be returned by functions.

2.10 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 `MinHS/Env.hs` provides a data type suitable for most uses. The lecture notes may give a guide on use of environments in dynamic semantics. In general, you will need to use: *empty*, *lookup*, *add* and *addAll* to begin with an empty environment, lookup the environment, or to add binding(s) to the environment, respectively.

As these functions clash with functions in the `Prelude`, a good idea is to import the module `Env` qualified:

```
import qualified Env
```

This makes the functions accessible as $\mathit{Env.empty}$ and $\mathit{Env.lookup}$, to disambiguate from the Prelude versions.

3 Dynamic Semantics of **`hindsight`**

Big-step semantics

We define two mutually recursive judgements, a big step semantics for value expressions, $\Gamma \vdash v \Downarrow_v V$ and a big step semantics for computational expressions, $\Gamma \vdash e \Downarrow_e T$. The first relates an environment mapping variables to values Γ and a value expression v to the resultant value of that expression V. The second maps the same kind of environment Γ and a computation expression e to a *terminal computation* T. Our value set for V will, to start with, consist of:

- *Machine integers*
- *Boolean values*
- *Lists of integers*

Our terminal computations T consist of:

- $P\ V$, a computation that immediately produces the value V.
- *function terminals*, whose shape you must decide.

We will use t to range over terminal computations, and v to denote values. Note that v can also denote value expressions; it should be clear from context which one is intended.

We will also need to add *closures* or *function terminals* to our terminal computation set, to deal with the `recfun` construct in a sound way, and a constructor for *thunk values* to our value set to deal with `thunk`. There are some design decisions to be made here, and they're up to you.

Environment

The environment Γ maps variables to values, and is used in place of substitution. 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 thunk values created with `thunk` whose bodies contain variables bound in an outer scope. We must bundle them with their associated environment. The same problem will also arise for computations created with `recfun`, and requires introducing *closures*. Care must also be taken to support suspended functions in thunk values.

Constants and Boolean Constructors

$$\frac{}{\Gamma \vdash \texttt{Num}\ n \Downarrow_v n} \qquad \frac{}{\Gamma \vdash \texttt{Con True} \Downarrow_v \mathit{True}} \qquad \frac{}{\Gamma \vdash \texttt{Con False} \Downarrow_v \mathit{False}}$$

Primitive operations

$$\frac{\Gamma \vdash v_1 \Downarrow_v v_1' \quad \Gamma \vdash v_2 \Downarrow_v v_2'}{\Gamma \vdash \texttt{Add}\ v_1\ v_2 \Downarrow_c P(v_1' + v_2')}$$

Similarly for the other arithmetic and comparison operations (as for the language of arithmetic expressions)

Note that division by zero should cause your interpreter to throw an error using Haskell's `error` function.

The abstract syntax of the interpreter re-uses function application to represent application of primitive operations, so $\texttt{Add}\ e_1\ e_2$ is actually represented as:

$$\texttt{App}\ (\texttt{App}\ (\texttt{Prim Add})\ e_1)\ e_2$$

For this first part of the assignment, you may assume that primops are never partially applied — that is, they are fully supplied with arguments, so the term $\texttt{App}\ (\texttt{Prim Add})\ e_1$ will never occur in isolation.

Evaluation of if**-expression**

$$\frac{\Gamma \vdash v \Downarrow_v \mathit{True} \quad \Gamma \vdash c_1 \Downarrow_c t}{\Gamma \vdash \texttt{If}\ v\ c_1\ c_2 \Downarrow_c t}$$

$$\frac{\Gamma \vdash v \Downarrow_v \mathit{False} \quad \Gamma \vdash c_2 \Downarrow_c t}{\Gamma \vdash \texttt{If}\ v\ c_1\ c_2 \Downarrow_c t}$$

Variables

$$\frac{\Gamma(x) = v}{\Gamma \vdash \texttt{Var}\ x \Downarrow_v v}$$

List constructors and primops

$$\frac{}{\Gamma \vdash \texttt{Con Nil} \Downarrow_v []} \qquad \frac{\Gamma \vdash x \Downarrow_v v_x \quad \Gamma \vdash xs \Downarrow_v v_{xs}}{\Gamma \vdash \texttt{(force (Con Cons))}\ x\ xs \Downarrow_c P(v_x : v_{xs})}$$

$$\frac{\Gamma \vdash x \Downarrow_v v : vs}{\Gamma \vdash \texttt{head}\ x \Downarrow_c P(v)} \qquad \frac{\Gamma \vdash x \Downarrow_v v : vs}{\Gamma \vdash \texttt{tail}\ x \Downarrow_c P(vs)} \qquad \frac{\Gamma \vdash x \Downarrow_v v : vs}{\Gamma \vdash \texttt{null}\ x \Downarrow_c P(\mathit{False})}$$

$$\frac{\Gamma \vdash x \Downarrow_v []}{\Gamma \vdash \texttt{head}\ x \Downarrow_c \texttt{error}} \qquad \frac{\Gamma \vdash x \Downarrow_v []}{\Gamma \vdash \texttt{tail}\ x \Downarrow_c \texttt{error}} \qquad \frac{\Gamma \vdash x \Downarrow_v []}{\Gamma \vdash \texttt{null}\ x \Downarrow_c P(\mathit{True})}$$

For the first part of the assignment, you may assume that `Cons` is also never partially applied, as with primops.

Produce

$$\frac{\Gamma \vdash v_1 \Downarrow_v v_2}{\Gamma \vdash \texttt{Produce}\ v_1 \Downarrow_c P(v_2)}$$

Variable Bindings with Let and Reduce

$$\frac{\Gamma \vdash v_1 \Downarrow_v v_2 \quad \Gamma, x{=}v_2 \vdash c \Downarrow_c t}{\Gamma \vdash \texttt{Let}\ v_1\ (x.c) \Downarrow_c t}$$

$$\frac{\Gamma \vdash c_1 \Downarrow_c P(v) \quad \Gamma, x{=}v \vdash c_2 \Downarrow_c t}{\Gamma \vdash \texttt{Reduce}\ c_1\ (x.c_2) \Downarrow_c t}$$

Thunk values

To maintain soundness with thunk values, we need to pair a function with its environment, forming a *closure*. We introduce the following syntax for thunk values:

$$\langle\!\langle \Gamma; c \rangle\!\rangle$$

You will need to decide on a suitable representation of thunk values as a Haskell data type. Also consider how you will represent suspended functions — can you reuse your existing representation, or do you need something different?

Thunk and Force semantics

Now we can give the semantics of suspending and resuming computations:

$$\frac{}{\Gamma \vdash \texttt{Thunk}\ c \Downarrow_v \langle\!\langle \Gamma; c \rangle\!\rangle} \qquad \frac{\Gamma \vdash v \Downarrow_v \langle\!\langle \Gamma'; c \rangle\!\rangle \quad \Gamma' \vdash c \Downarrow_c t}{\Gamma \vdash \texttt{Force}\ v \Downarrow_c t}$$

Function terminals

For similar reasons, unapplied functions can be regarded as terminal computations, and you need to keep track of the environment in which they've been defined. We can reuse a similar representation:

$$\langle\!\langle \Gamma; \texttt{Recfun}\ \tau_1\ \tau_2\ f.x.e \rangle\!\rangle$$

The types are not needed at runtime, but included here for completeness. You will need to decide on a suitable representation of function terminals.

Now we can specify how to introduce closed function terminals:

$$\frac{}{\Gamma \vdash \texttt{Recfun}\ \tau_1\ \tau_2\ f.x.e_1 \Downarrow_c \langle\!\langle \Gamma; \texttt{Recfun}\ \tau_1\ \tau_2\ f.x.e_1 \rangle\!\rangle}$$

Function Application

$$\frac{\Gamma \vdash c_1 \Downarrow_c t_1 \quad t_1 = \langle\!\langle \Gamma' ; \texttt{Recfun}\ \tau_1\ \tau_2\ f.x.c_f \rangle\!\rangle \qquad \Gamma \vdash v_1 \Downarrow_v v_2 \quad \Gamma', f{=}t_1, x{=}v_2 \vdash c_f \Downarrow_c t_2}{\Gamma \vdash \texttt{App}\ c_1\ v_1 \Downarrow_c t_2}$$

This rule involves some notational abuse: t_1 is a terminal, not a value. But we need to put it in the environment. Do you need to extend your value type to accommodate this?

---

4 Additional Tasks

In order to get full marks in the assignment, you must do some of the following five tasks:

4.1 Task 2: Partial Primops (10%)

In the base part of the assignment, you are allowed to assume that all primitive operations (and the constructor `Cons`) are fully provided with arguments. In this task you are to implement *partial* application of primitive operations (and `Cons`), which removes this assumption. For example:

```
main :: F Int
     = let inc :: U (Int -> F Int) = thunk ((+) 1);
       in force inc 2  -- returns 3
```

Note that the expression `(+) 1` partially applies the primop `Add` to `1`, returning a *function* from `Int` to `Int`.

You will need to develop a suitable dynamic semantics for such expressions and implement it in your evaluator. The parser and type checker are already capable of dealing with expressions of this form.

4.2 Task 3: Multiple bindings in `let` (10%)

In the base part of the assignment, we specify that `let` expressions contain only one binding. In this task, you are to extend the interpreter to `let` expressions with multiple bindings, like:

```
main :: F Int
     = let a :: Int = 3;
           b :: Int = 2;
        in a + b
```

These are evaluated the same way as multiple nested `let` expressions:

```
main :: F Int
     = let a :: Int = 3;
        in let b :: Int = 2;
            in a + b
```

Once again the only place where extensions need to be made are in the evaluator, as the type checker and parser are already capable of handling multiple `let` bindings.

4.3 Task 4: Code optimisation (10%)

In `Hindsight/Optimiser.hs` you'll find the skeleton for an optimisation pass, which we can think of as a compiler from `hindsight` to `hindsight`. Here, you can implement a code optimiser to eliminate certain redundant patterns. The main purpose of this phase is to get simpler, cleaner code after compiling from MinHS. You need to optimise away at least the following patterns:

Pattern to optimise	Desired result	
$\texttt{force}\,(\texttt{thunk}\ v)$	v	
$\texttt{reduce produce}\ v\ \texttt{to}\ x\ \texttt{in}\ c$	$c[x := v]$	
$(\texttt{recfun}\ f :: (\tau_1 \to \tau_2)\ x = c)\ v$	$c[x := v]$	if $f \notin \mathsf{FV}(c)$

You can do more optimisations if you want to; it shouldn't influence your marks, since the marking makes no attempt to measure code quality. But make sure your optimiser output is semantically equivalent to the input program, and contains no instances of the above three patterns.

Your optimiser must terminate for all input programs. Hence, your optimiser can't be an evaluator, and you probably never want to unfold or inline functions that feature recursive calls.

The optimiser is the only place in the assignment where you can and should use substitution.

The optimiser is not invoked by default, but can be invoked stand-alone using `minhs` with `--dump optimiser foo.hst`, or executed after compilation from MinHS by using `--dump compiler --optimise foo.mhs`

4.4 Task 5: A call-by-name translation from MinHS (20%)

The main purpose of `hindsight` is to serve as an intermediate representation of MinHS programs, as part of a compiler or (in this case) interpreter. This requires a compiler from MinHS to `hindsight`. By choosing different compilation strategies, we can execute MinHS with either call-by-name semantics or call-by-value semantics.

In `Hindsight/CBVCompile.hs` you'll find a call-by-value compiler already implemented.

This task is to develop a call-by-name compiler in `Hindsight/CBNCompile.hs`

The reference we used when implementing the call-by-value compiler was the translation from call-by-value λ-calculus to CBPV from Paul Levy's PhD thesis [3, Figure 3.5]. The thesis also includes a translation from call-by-name λ-calculus [3, Figure 3.6], which you should consult for inspiration. Not every MinHS primitive has a direct λ-calculus equivalent though, so there are some gaps you need to figure out how to bridge. Also, beware of idiosynchrasies in Levy's notation.[1]

It is occasionally necessary for the compiler to invent new names. Obviously, these names should not clash with names occuring in the source program. Make sure you have a strategy for dealing with this. The CBV compiler solves this problem by prefixing an underscore to all names the programmer wrote, meaning there's no risk of clashes so long as we don't invent names starting in underscores.

The call-by-name compiler can be invoked stand-alone by using `minhs` like this (assuming you're on `stack`):

```
stack exec minhs-1 -- --dump compiler --cbn foo.mhs
```

[1] For example, Levy writes function application as $x\ ‘\ f$ instead of $f\ x$

To immediately run your `hindsight` evaluator after CBN compilation, use `--cbn foo.hst`. Note that evaluation may fail on correct programs, unless you've implemented extension tasks 1 and 2.

There are plenty of tests you can use in the `mhs_tests` directory.

MinHS has some features that your compiler is *not* expected to handle. These are:

- Recursive function definitions with multiple arguments.
- Let bindings that accept arguments.
- Recursive functions with zero arguments—this need only be supported in the `main` function.

Your compiler output needs to contain type annotations—you can see if these are correct by invoking the typechecker on the compiler output.

5 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.

The tests that come with this assignment tarball, which are also run on submission as a dryrun, cover the *base part* (the first 70%) of the assignment only. You will be responsible for testing each extension adequately.

6 Building `minhs`

`minhs` (the compiler/interpreter) is written in Haskell, and requires GHC and the `cabal` build tool included in the Haskell Platform, or the `stack` tool that is also popular for building Haskell projects. If you are using CSE machines, follow instructions for `cabal`, however if you are working on your own machine you may find it more convenient to use `stack`.

6.1 Building with `cabal` on CSE machines

All testing will occur on standard CSE Linux machines. Make sure you test your program on a CSE Linux machine.

- `cabal update` to set up the package database.
- `cabal configure` to set up the build environment
- `cabal install --only-dependencies` to install the libraries on which the interpreter depends.
- `cabal build` to build the compiler
- `cabal run minhs-1 -- --help` will help you find several useful debugging and testing options.

To run the interpreter on a file `foo.hst`:

```
$ cabal run minhs-1 -- foo.hst
```

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.

To run the test driver, a short shell script is provided. For usage information, type:

```
$ ./run_tests_cabal.sh --help
```

6.2 Building with `stack`

You should be able to build the compiler by simply invoking:

```
$ stack build
```

To see the debugging options, run (after building):

```
$ stack exec minhs-1
```

To run the evaluator with a particular file, run:

```
$ stack exec minhs-1 -- foo.hst
```

And to run all of our tests, type:

```
$ ./run_tests_stack.sh
```

7 Late Penalty

You may submit up to five days (120 hours) late. Each day of lateness corresponds to a 5% reduction of your total mark. For example, if your assignment is worth 88% and you submit it two days late, you get 78%. If you submit it more than five days late, you get 0%.

Course staff cannot grant assignment extensions—if you need an extensions, you have to apply for special consideration through the standard procedure. More information here: `https://www.student.unsw.edu.au/special-consideration`

8 Plagiarism

Many students do not appear to understand what is regarded as plagiarism. This is no defense. Before submitting any work you should read and understand the UNSW plagiarism policy `https://student.unsw.edu.au/plagiarism`.

All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. In this course submission of any work derived from another person, or solely or jointly written by and or with someone else, without clear and explicit acknowledgement, will be severely punished and may result in automatic failure for the course and a mark of zero for the course. Note this includes including unreferenced work from books, the internet, etc.

Do not provide or show your assessable work to any other person. Allowing another student to copy from you will, at the very least, result in zero for that assessment. If you knowingly provide or show your assessment work to another person for any reason, and work derived from it is subsequently submitted you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep your work private until submissions have closed.

If you are unsure about whether certain activities would constitute plagiarism ask us before engaging in them!

References

[1] *Report on the Programming Language Haskell 98*, eds. Simon Peyton Jones, John Hughes, (1999) `http://www.haskell.org/onlinereport/`

[2] Robert Harper, *Programming Languages: Theory and Practice*, (Draft of 19 Sep 2005), `https://people.cs.uchicago.edu/~blume/classes/aut2008/proglang/text/offline.pdf`.

[3] Paul Blain Levy, *Call-by-push-value*. PhD thesis, Queen Mary, University of London, 2001. `https://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf`.

[4] The Implementation of Functional Programming Languages, Simon Peyton Jones, published by Prentice Hall, 1987. Full text online (as jpg page images).

[5] Simon Peyton-Jones, *Implementing Functional Languages : a tutorial*, 2000.

Reference materials describing MinHS are included here as an appendix.

A Lexical Structure

The lexical structure of MinHS is an small subset of Haskell98. See section 2.2 of the Haskell98 report [1]. The lexical conventions are implemented by the Parsec parser library, which we use for our Parser implementation.

B 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 left out. As a result, a program consists of a single top-level binding for a `main` function, of atomic type. There is an `if-then-else` conditional expression. The primitive types of MinHS are `Int`,`Bool` and `[Int]`. MinHS also implements, at least partially, a number of extensions compared to what we've seen in the lectures: 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 `recfun`.

The concrete syntax is described and implemented in the `MinHS/Parser.hs` module, a grammar specified using the Parser combinator library Parsec.

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.

C Abstract syntax

The (first-order) abstract syntax is based closely on the MinHS syntax introduced in the lectures. It is implemented in the file `MinHS/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.

D Static semantics

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

D.1 n-ary functions

The MinHS parser, and typechecker supports functions with more than 1 argument. However, there is no expectation that any of your work supports functions with any more (or less) than 1 argument.

Types	τ	$\rightarrow$	`Int` $\mid$ `Bool` $\mid \tau \rightarrow \tau$
Literals	n	$\rightarrow$	`...` $\mid$ `0` $\mid$ `1` $\mid$ `2` $\mid$ `...`
	b	$\rightarrow$	`True` $\mid$ `False`
Primops	o	$\rightarrow$	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$
		$\mid$	`>` $\mid$ `>=` $\mid$ `==` $\mid$ `/=` $\mid$ `<` $\mid$ `<=`
Expressions	exp	$\rightarrow$	`Var` x
		$\mid$	`Lit` n
		$\mid$	`Con` b
		$\mid$	`Apply` $e_1\ e_2$
		$\mid$	`Let` $decl\ exp$
		$\mid$	`Recfun` $decl$
		$\mid$	`If` $exp\ exp_1\ exp_2$
Decl	$decl$	$\rightarrow$	`Fun` $f\ \tau\ [arg]\ e$
		$\mid$	`Val` $v\ \tau\ e$

Figure 1: The expression abstract syntax of MinHS

E 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.

F Dynamic semantics

The (call-by-value) dynamic semantics for MinHS resemble that of Harper [2]. Rules are given in Figure 2. We do not implement it directly; instead, `MinHS/Evaluator.hs` evaluates MinHS programs by first compiling them to `hindsight`, and the running the `hindsight` evaluator.

G Interfaces

The basic types are found in `Syntax.hs`, which contains definitions for the structure of terms, types, primOps, and others.

$$\frac{}{\Gamma \vdash \texttt{Num}\ n \Downarrow n} \qquad \frac{}{\Gamma \vdash \texttt{Con True} \Downarrow \mathit{True}} \qquad \frac{}{\Gamma \vdash \texttt{Con False} \Downarrow \mathit{False}}$$

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\Gamma \vdash \texttt{Add}\ e_1\ e_2 \Downarrow v_1 + v_2} \qquad \frac{\Gamma \vdash e_1 \Downarrow \mathit{True} \quad \Gamma \vdash e_2 \Downarrow x}{\Gamma \vdash \texttt{If}\ e_1\ e_2\ e_3 \Downarrow x}$$

$$\frac{\Gamma \vdash e_1 \Downarrow \mathit{False} \quad \Gamma \vdash e_3 \Downarrow x}{\Gamma \vdash \texttt{If}\ e_1\ e_2\ e_3 \Downarrow x} \qquad \frac{\Gamma(x) = v}{\Gamma \vdash \texttt{Var}\ x \Downarrow v} \qquad \frac{}{\Gamma \vdash \texttt{Con Nil} \Downarrow []}$$

$$\frac{\Gamma \vdash x \Downarrow v_x \quad \Gamma \vdash xs \Downarrow v_{xs}}{\Gamma \vdash \texttt{Cons}\ x\ xs \Downarrow v_x : v_{xs}} \qquad \frac{\Gamma \vdash x \Downarrow v : vs}{\Gamma \vdash \texttt{head}\ x \Downarrow v} \qquad \frac{\Gamma \vdash x \Downarrow v : vs}{\Gamma \vdash \texttt{tail}\ x \Downarrow vs}$$

$$\frac{\Gamma \vdash x \Downarrow v : vs}{\Gamma \vdash \texttt{null}\ x \Downarrow \mathit{False}} \qquad \frac{\Gamma \vdash x \Downarrow []}{\Gamma \vdash \texttt{head}\ x \Downarrow \texttt{error}} \qquad \frac{\Gamma \vdash x \Downarrow []}{\Gamma \vdash \texttt{tail}\ x \Downarrow \texttt{error}}$$

$$\frac{\Gamma \vdash x \Downarrow []}{\Gamma \vdash \texttt{null}\ x \Downarrow \mathit{True}} \qquad \frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma, x{=}v_1 \vdash e_2 \Downarrow v_2}{\Gamma \vdash \texttt{Let}\ e_1\ (x.e_2) \Downarrow v_2}$$

$$\frac{}{\Gamma \vdash \texttt{Recfun}\ \tau_1\ \tau_2\ f.x.e_1 \Downarrow \langle\!\langle \Gamma; \texttt{Recfun}\ \tau_1\ \tau_2\ f.x.e_1 \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Downarrow v_1 \quad v_1 = \langle\!\langle \Gamma'; \texttt{Recfun}\ \tau_1\ \tau_2\ f.x.e_f \rangle\!\rangle \\ \Gamma \vdash e_2 \Downarrow v_2 \quad \Gamma', f{=}v_1, x{=}v_2 \vdash e_f \Downarrow r\end{array}}{\Gamma \vdash \texttt{App}\ e_1\ e_2 \Downarrow r}$$

Figure 2: The dynamic semantics of MinHS. Rules for most operators are elided.

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 `Pretty.hs` for an example.

Testing

`./run_tests_cabal.sh`

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.