EECS 662 Homework 4:
Hindley-Milner Type Inference

J. Garrett Morris

November 6, 2017

Introduction

This homework will help you develop and demonstrate an understand of *implicit polymorphism*, as realized by the Hindley-Milner type system, and of *type inference* for such systems.

The homework distribution contains three Haskell files:

`Core.hs`	contains the interpreter for the core language (all your changes will go here)
`Main.hs`	contains scaffolding code to run both parsers on either standard input or files
`Sugar.hs`	contains parsing and desugaring functions for an extension of the core language with data types

The `Main.hs` file includes a driver program to simplify interaction with the interpreter. You can build the driver either using the included `Makefile`, or using the following GHC command:

```
ghc -o hw4 --make Main.hs
```

There are two modes of interacting with the interpreter. If you just call the interpreter (without passing any file names on the command line) then it functions as an interactive interpreter. You can write expressions, and those expression will be parsed, checked, and evaluated. For example:

```
> 2 + 2
VInt 4
::  Int
> 2 + True
ERROR: "type error"
> \x -> x
VFun [] "x" (CVar "x")
::  b -> b
> :q
$
```

Or, you can pass the interpreter a file (or series of files) to interpret. For example, if the file `Add.stlc` contains the text `2+2`, then:

```
$ ./hw3 Add.stlc
VInt 4
::  Int
$
```

The interpreter prints the type of each term as well as its evaluation. This allows us to observe the inferred polymorphism—for example, the identity function can be applied to arguments of arbitrary type.

The homework distribution also include the sample source files `Id.hm` and `Error.hm`. The expected result is a comment on the first line of each file. Note that `Error.hm` is intended to produce a type error.

Type Inference for Hindley-Milner Polymorphism

Your task in this assignment is to implement type inference for the Hindley-Milner type system (see Appendix A). In doing so, we will be following a general approach known as *Algorithm M*, a variation on Milner's original *Algorithm W*.

The key observation in Milner's inference algorithm is that, while we may not immediately be able to figure out the type of any term, we will always be able to figure it out from the structure inside the term. For example, consider the lambda term

$$\backslash x \to x + x$$

Suppose we attempt to start constructing a typing derivation of this term. We have no (immediate) information available about the type of x. However, when we consider the typing of $x + x$, we can determine that x must have been of type `Int`, and so the original term must have been of type `Int` $\to$ `Int`.

To capture points where we need more information about a type, we will use *unification varaibles*. Intuitively, a unification variable is a "box" into which we will later put type information. Correspondingly, when we are checking our expectations for a type, we may have to fill in some boxes. This process of trying to fill in boxes to make two types match is called *unification*. Let us consider a psuedo-code execution of type inference corresponding to the example above. We begin by calling type checking on the main term, without any information in the environment:

```
check [] (CLam "x" (CAdd (CVar "x") (CVar "x")))
```

In previous assignments, the code for checking `CLam` terms knew what type to give the parameter. Instead, we will now use a unification variable a to stand in for the type.

```
check [("x", a)] (CAdd (CVar "x") (CVar "x"))
```

There's nothing interesting to say about type checking the variables (yet): they each have type a, but do not contribute any information to what type a might be. However, consider the typing of `CAdd`. In previous assignments, we have insisted that the two subexpression of `CAdd` must each have type `CTInt`. Now, instead, we will attempt to *unify* the computed types of the subterms with `CTInt`. So, the computed type of the left-hand subterm is the unification variable a; unifying this with `CTInt` succeeds, and "fills in" the a box with `CTInt`. The computed type of the right-hand subterm is also a, but this box has now be filled in with `CTInt`. So, unifying a with `CTInt` succeeds, and does not fill in any more boxes.

We consider a case where type checking fails. Our term is

$$\backslash x \to x + (\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 2)$$

The problem is that the term makes inconsistent requirements on the type x: for the addition to be well typed, x must be an integer, but for the conditional to be well-typed, x must be Boolean. Again, we consider a pseudo-execution of type checking on this term.

```
check [] (CLam "x" (CAdd (CVar "x") (CIf (CVar "x") (CInt 1) (CInt 2))))
```

As before, we start out with a unification variable a for the type of variable `x`.

```
check [("x", a)] (CAdd (CVar "x") (CIf (CVar "x") (CInt 1) (CInt 2)))
```

To check the `CAdd`, we compute the type of its left-hand argument (a), and then unify with `CTInt`. This succeeds, putting `CTInt` into the a box. Now, we must check the conditional. The checking for `CIf` will compute the type of the condition and attempt to unify with `CTBool`. The type of the condition (a) has already been unified with `CTInt`, so this means that we must unify `CTInt` with `CTBool`. There is no way we can fill in the remaining boxes such that these types are equal, so unification (and so type inference) fails.

The next question is how we can realize these pseudo-executions. The first observation is that, in the past, the `check` function took the environment and expression as arguments and returned the computed type. However, now we must also keep track of our progress in filling in unification variables. We will do this by passing a mapping from unification variables to types into the `check` function (the state of the unification variables before checking that term) and returning such a map from `check` (the state of the unification variables after checking the term). You should recognize this as the pattern that we have previously used to capture state in a monad, and indeed the provided code implements type checking in a state monad.

At this point, we can make an engineering observation. Rather than returning both a type and the updated unification variables, we could take the expected type of each subexpression as an *input*. In cases where we do not have any particular expectation for the result type, we can use another unification variable. This is the approach we will take in this assignment. It has several advantages: it tends to discover type errors earlier in type inference, produces better error messages, and (I think) improves the structure of type inference code. Look at the provided cases for `CAdd` and `CLetPair` to see how this works in practice.

You have two implementation tasks for this homework. First, you will implement unification, the core of Hindley-Milner type inference. Second, you will implement the remaining cases in inference.

Provided Code

The provided code includes a suitable monad for implementing type inference, called `TcM`, along with a collection of utility functions for manipulating types and schemes (listed below). You should *not* need to ever manipulate monadic values via the `TcM` constructor or other details of the implementation of `TcM`.

Function	**Purpose**
`assumeType`	`assumeType x t g` adds the assumption that `x` has type `t` to type environment `g`, returning the resulting environment
`assumeScheme`	`assumeScheme x s g` adds the assumption that `x` has scheme `s` to type environment `g`, returning the resulting environment
`typeError`	Signals that type inference has failed; takes one string argument for an error message
`fresh`	Creates a fresh (i.e., not used before) unification variable, and returns it as a `CType`
`bind`	`bind a t` binds unification variable `a` to type `t` in the type checker's state; `bind` does not check to see whether `a` is already in the state, but you should not need to worry about this if you've called `expect` (or `applyM`) at the right times
`apply`	`apply s t` applies type substitution `s` to type `t`, returning the resulting type
`applyM`	`applyM t` applies the substitution in the type checker's state to type `t`, returning the resulting type
`applyS`	`applyM s` applies the substitution in the type checker's state to *scheme* `s`, returning the resulting scheme
`generalize`	`generalize g t` converts type `t` into a type scheme, by quantifying over all the variables that do not appear in `g`. You do not need to worry about unification variables appearing in `g` or `t`; `generalize` takes them into account
`instantiate`	`instantiate s` converts scheme `s` into a type by replacing all the quantified variables in `s` with fresh unification variables
`expect`	A small wrapper around unification; `expect t u` applies the stored substitution to `t` and `u` before unifying them

1 Unification

Your first task is to implement unification. The type of unification is

```
unify ::  CType -> CType -> TcM ()
```

Unification either succeeds, possibly introducing new bindings for the unification variables in its argument; otherwise, it causes type inference to fail. Both of these effects are captured in the `TcM` monad, so `unify` just returns a unit value.

The intuition for how to unify two types t and u is as follows.

- If the types are equal, then succeed.
- If either type is a unification variable, say type t is unification variable a, then bind a to type u. The provided function `bind` handles the details of adding bindings to the monad.
- If the types are incomparable (say, one is `Int`and the other is `Bool`, or one is a product and the other is a function), then fail. The provided `typeError` function handles the details of signaling failure.
- Otherwise, the types must both be comparable compound types (for example, both function types or both product types). In this case, unify the component types. For example, if t is a function $t_1 \to t_2$ and u is a function $u_1 \to u_2$, then you should unify t_1 and u_1 and then unify t_2 and u_2.

Note that the results of earlier unifications need to inform the results of later unifications. For example, unifying the types $a \to a$ and `Int` $\to$ `Bool` should fail, even though independently the unifications of a and `Int` or a and `Bool` would succeed. The provided function `applyM`, which applies the stored substitutions to any unification variables in a type, may be useful.

2 Inference

Your second task is to implement type inference. This mostly follows the rules. We consider two examples.

```
check g (CAdd e1 e2) t =
    do expect CTInt t
       check g e1 CTInt
       check g e2 CTInt
```

This is the case for `CAdd`; `t` is the expected type. The result of an addition is always an integer, so we start out by unifying the expected type with `CTInt`. (The `expect` function is a simple wrapper around your `unify` function; you should not call `unify` directly from inference.) Then, we check the two subterms; each of them must also be an integer, so we pass `CTInt` as the expected type in each case.

```
check g (CLetPair x1 x2 e1 e2) t =
    do u1 <- fresh
       u2 <- fresh
       check g e1 (CTProd u1 u2)
       check (assumeType x1 u1 (assumeType x2 u2 g)) e2 t
```

This is the case for `let` on pairs. First, we need to check the type of the expression e_1. This type needs to be a product, but the component types can be arbitrary types. We capture this by first, creating two new unification variables u_1 and u_2 (this is the function of the provided `fresh` operation). Then, we check that e_1 has type $u_1 \times u_2$. If this succeeds, then we need to check e_2. We know that x_1 should have type u_1 (whatever that happened to be), and x_2 should have type u_2. The provided `assumeType` operation extends the type environment.

The cases for variables and `let` have to handle type schemes as well as types. There are two provided functions to help. The `generalize` function generalizes a type, given the type and the current type environment; the `instantiate` function replaces all the type variables in a type scheme with fresh unification variables.

Submission Instructions

Your submission should include the three Haskell files from the original homework distribution, modified as necessary for your solution, the test files from the original distribution, and (at least) two new test cases, stored in separate files. Each of your test files should being with a comment line (i.e., a line beginning with two dashes) containing their expected result. The provided test follows this format; check there for details. Please submit these as a single file (`.zip` or `.tar`), via Blackboard.

(Provisional) point allocation

Unification	75
Type inference	125
Test cases (2)	40
Total	240

I will run all of the submitted code against all of the submitted test cases. For each bug that one of your test cases finds in one (or more) of your classmates' assignments, you will receive a small amount of bonus credit.

A The Hindley-Milner Type System

$$(\textsc{var})\ \frac{(x : s) \in \Gamma \quad t \in \lfloor s \rfloor}{\Gamma \vdash x : t} \qquad (\textsc{let})\ \frac{\Gamma \vdash e_1 : u \quad s = \mathit{Gen}(\Gamma, u) \quad \Gamma, x : s \vdash e_2 : t}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t}$$

$$(\to\mathrm{I})\ \frac{\Gamma, x : t \vdash e : u}{\Gamma \vdash \backslash x \to e : t \to u} \qquad (\to\mathrm{E})\ \frac{\Gamma \vdash e_1 : u \to t \quad \Gamma \vdash e_2 : u}{\Gamma \vdash e_1\ e_2 : t}$$

$$(\times\mathrm{I})\ \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : u}{\Gamma \vdash (e_1, e_2) : t \times u} \qquad (\times\mathrm{E})\ \frac{\Gamma \vdash e_1 : t_1 \times t_2 \quad \Gamma, x_1 : t_1, x_2 : t_2 \vdash e_2 : u}{\Gamma \vdash \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 : u}$$

$$(+\mathrm{I}_1)\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{Inl}\ e : t + u} \qquad (+\mathrm{I}_2)\ \frac{\Gamma \vdash e : u}{\Gamma \vdash \texttt{Inr}\ e : t + u} \qquad (+\mathrm{E})\ \frac{\Gamma \vdash e : t_1 + t_2 \quad \Gamma, x_1 : t_1 \vdash e_1 : u \quad \Gamma, x_2 : t_2 \vdash e_2 : u}{\Gamma \vdash \texttt{case}\ e\ \texttt{of Inl}\ x_1 \to e_1 \mid \texttt{Inr}\ x_2 \to e_2 : u}$$

Constant rules

$$\frac{}{\Gamma \vdash 1 : \texttt{Int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 * e_2 : \texttt{Int}}$$

$$\frac{}{\Gamma \vdash \texttt{True} : \texttt{Bool}} \qquad \frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{isz}\ e : \texttt{Bool}} \qquad \frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t}$$

$$\frac{\Gamma \vdash e : (t \to u) \to (t \to u)}{\Gamma \vdash \texttt{fix}\ e : t \to u}$$