3161/9161 17s2 Assignment 2
Type Inference for Polymorphic MinHs

Version 1.0

Marks : 30% of the class mark for 3161 students
30% of the class mark for 9161 students

Due date: **13st October 2017, 23:55** (11:55pm)

History

22nd September Initial version released

Overview

In this assignment you will implement a type inference pass for MinHS. The language used in this assignment differs from the language of the first assignment in two respects: it has a polymorphic type system, and it has aggregate data structures.

The assignment involves:

- (100%) implement type synthesis for polymorphic MinHS with sum and product data types.
- (5% bonus mark) adjust the type inference pass to include various simple syntax extensions from Assignment 1.
- (5% bonus mark) extend the type inference pass to elaborate mutually recursive `letrec` bindings.
- (10% bonus mark) adjust the type inference pass to allow optional type annotations provided by the user.

Each of these parts is explained in detail below.

The parser, and an interpreter backend are provided for you. You do not have to change anything in any module other than `TyInfer.hs` (even for the bonus parts).

Your type inference pass should annotate the abstract syntax with type information where it is missing. The resulting abstract syntax should be fully annotated and correctly typed.

Your assignment will only be tested on correct programs, and will be judged correct if it produces correctly typed annotated abstract syntax, up to α-renaming of type variables.

Please consult with your teachers and fellow students on the class web page.

1 Task 1

Task 1 is worth 100% of the marks of this assignment. You are to implement type inference for MinHS with aggregate data structures. The following cases must be handled:

- the MinHS language of the first task of assignment 1 (without n-ary functions or `letrec`s, or lists)
- product types: the 0-tuple and 2-tuples.
- sum types
- polymorphic functions

These cases are explained in detail below. The abstract syntax defining these syntactic entities is in `Syntax.hs`. You should not need to modify the abstract syntax definition in any way.

Your implementation is to follow the definition of aggregates types found in the lecture on data structures, and the rules defined in the two lectures on type inference. Additional material can be found in the lecture notes on polymorphism, and the reference materials. The full set of rules is outlined below.

2 Bonus Tasks

These tasks are all optional, and are worth a total of an additional 20%.

2.1 Bonus Task 1: Simple Syntax Extensions

This bonus task is worth an additional 5%. In this task, you should implement type inference for n-ary functions, and multiple bindings in the one `let` expression, with the same semantics as the extension tasks for Assignment 1.

You will need to develop the requisite extensions to the type inference algorithm yourself, but the extensions are very similar to the existing rules.

2.2 Bonus Task 2: Recursive Bindings

This bonus task is worth an additional 5%. In this task you must implement type inference for `letrec` constructs.

Each binding will require a fresh unification variable, and after every binding has an inferred type, the types must be appropriately generalised (similarly to `let` bindings).

2.3 Bonus Task 3: User-provided type signatures

This bonus task is worth an additional 10%. In this task you are to extend the type inference pass to accept programs containing *some* type information. You need to combine this with the results of your type inference pass to produce the final type for each declaration. That is, you need to be able to infer correct types for programs like:

```
main = let f :: (Int -> Int)
             = letfun g x = x;
       in f 2;
```

You must ensure that the type signatures provided are not overly general. For example, the following program should be a type error, as the user-provided signature is too general:

```
main :: (forall a. a) = 3;
```

3 Aggregate Data Types

This section covers the extensions to the language of the first assignment. In all other respects (except lists) the language remains the same, so you can consult the reference material from the first assignment for further details on the language.

3.1 Product Types

We only have 0-tuples and 2-tuples in MinHS.

Types	τ	$\rightarrow$	$\tau_1 * \tau_2$
		$\mid$	`Unit`
Expressions	exp	$\rightarrow$	(e_1, e_2)
		$\mid$	`fst` e $\mid$ `snd` e
		$\mid$	`()`

3.2 Sum Types

Types	τ	$\rightarrow$	$\tau + \tau$
Expressions	exp	$\rightarrow$	`Inl` e_1
		$\mid$	`Inr` e_2
		$\mid$	`case` e `of`
			`Inl` x `->` e_1 `;`
			`Inr` y `->` e_2 `;`

3.3 Polymorphism

The extensions to allow polymorphism are relatively simple, two new type form has been introduced, the `TypeVar t` form, and the `Forall t e` form.

Types	τ	$\rightarrow$	`forall` τ. ..τ..

For example, consider the following code fragment before and after type inference:

```
main =
  let f = letfun g x = x;
  in if f True
       then f (Inl 1)
       else f (Inr ());

main :: (Int + Unit) =
  let f :: forall a. (a -> a) = letfun g :: (a -> a) x = x;
  in if f True
       then f (Inl 1)
       else f (Inr ());
```

4 Type Inference Rules

Sections coloured in blue can be viewed as *inputs* to the algorithm, while red text can be viewed as outputs.

Constants and Variables

$$\frac{}{\Gamma \vdash n : \texttt{Int}} \qquad \frac{x : \forall a_1 \ldots \forall a_n . \tau \in \Gamma}{\Gamma \vdash x : [a_1 := \beta_1] \ldots [a_n := \beta_n]\tau} \; \beta_i \textit{ fresh}$$

Constructors and Primops

$$\frac{\textit{constructorType}(C) = \forall a_1 \ldots \forall a_n . \tau}{\Gamma \vdash C : [a_1 := \beta_1] \ldots [a_n := \beta_n]\tau} \qquad \frac{\textit{primopType}(o) = \forall a_1 \ldots \forall a_n . \tau}{\Gamma \vdash o : [a_1 := \beta_1] \ldots [a_n := \beta_n]\tau} \qquad \beta_i \textit{ fresh}$$

(*constructorType* and *primopType* are defined in `TyInfer.hs`)

Application

$$\frac{T\Gamma \vdash e_1 : \tau_1 \qquad T'T\Gamma \vdash e_2 : \tau_2 \qquad T'\tau_1 \overset{U}{\sim} (\tau_2 \to \alpha)}{UT'T\Gamma \vdash \texttt{Apply}\ e_1\ e_2 : U\alpha} \; \alpha \textit{ fresh}$$

If-Then-Else

$$\frac{T\Gamma \vdash e : \tau \quad \tau \overset{U}{\sim} \texttt{Bool} \quad T_1UT\Gamma \vdash e_1 : \tau_1 \quad T_2T_1UT\Gamma \vdash e_2 : \tau_2 \quad T_2\tau_1 \overset{U'}{\sim} \tau_2}{U'T_2T_1UT\Gamma \vdash \texttt{If}\ e\ e_1\ e_2 : U'\tau_2}$$

Case

$$\frac{T\Gamma \vdash e : \tau \qquad \begin{array}{c} T_1T(\Gamma \cup \{x : \alpha_l\}) \vdash e_1 : \tau_l \\ T_2T_1T(\Gamma \cup \{y : \alpha_r\}) \vdash e_2 : \tau_r \end{array} \qquad T_2T_1T(\alpha_l + \alpha_r) \overset{U}{\sim} T_2T_1\tau \qquad UT_2\tau_l \overset{U'}{\sim} U\tau_r}{U'UT_2T_1T\Gamma \vdash \texttt{Case}\ e\ \ x.e_1\ \ y.e_2 : U'U\tau_r} \; \alpha_l,\ \alpha_r \textit{ fresh}$$

Recursive Functions

$$\frac{T(\Gamma \cup \{x : \alpha_1\} \cup \{f : \alpha_2\}) \vdash e : \tau \quad T\alpha_2 \overset{U}{\sim} T\alpha_1 \to \tau}{UT\Gamma \vdash \texttt{Letfun}\ f.x.e : U(T\alpha_1 \to \tau)} \; \alpha_1, \alpha_2 \textit{ fresh}$$

Let Bindings

$$\frac{T\Gamma \vdash e_1 : \tau \qquad T'(T\Gamma \cup \{x : \textit{Generalise}(T\Gamma, \tau\}) \vdash e_2 : \tau'}{T'T\Gamma \vdash \texttt{Let}\ e_1\ x.e_2 : \tau'} \qquad \frac{T\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Main}\ e : \textit{Generalise}(\Gamma, \tau)}$$

where $\textit{Generalise}(\Gamma, \tau) = \forall\, (TV(\tau) \setminus TV(\Gamma))\, .\, \tau$

4.1 Implementing the algorithm

The type inference rules imply an algorithm, where the expression and the environment can be seen as input, and the substitution and the type of the expression can be seen as output, as seen from the color coding above.

Such an algorithm would probably have a type signature like:

```
inferExp :: Gamma -> Exp -> TC (Type, Subst)
```

However our goal isn't just to *infer* the type of the expression, but to also *elaborate* the expression to its explicitly-typed equivalent. In other words, we want to add typing annotations to our expressions, meaning we must return a *new expression* as well as the type and substitution.

```
inferExp :: Gamma -> Exp -> TC (Exp, Type, Subst)
```

The cases for `let` and `letfun` (and `letrec`, in the bonus) must add a (correct) type signature to the resultant expression, and all other cases must be sure to return a new expression consisting of the updated subexpressions.

Because we add annotations to the expressions on the way, those type annotations wouldn't contain the information we get from typing successive expressions. Consider the following example:

$$\{+ : Int \to Int \to Int, x : a\} \vdash \texttt{pair (let z = x in x , x+1)}$$

When typing `let z = x in x`, we only know that `x` is of type `a`, so we would add the type annotation `let z :: a = x in x`. Only when we type the second expression, `x+1`, we know that a has to be `Int` (which will be reflected in T'). That's why, when we're done typing the top-level binding, we have to traverse the whole binding again, applying the substitution to each type annotation anywhere in the binding.

The functions `mapTypes`, `mapTypesBind` and `mapTypesAlt`, defined in `Syntax.hs`, allow you to update each type annotation in an expression, making it easy to perform this substitution.

5 Substitution

Substitutions are implemented as an abstract data type, defined in `Subst.hs`. `Subst` is an instance of the `Monoid` type class, which is defined in the standard library as follows:

```
class Monoid a where
  mappend :: a -> a -> a   -- also written as the infix operator <>
  mempty  :: a
```

For the `Subst` instance, `mempty` corresponds to the empty substitution, and `mappend` is *substitution composition*. That is, applying the substitution `a <> b` is the same as applying `a` and `b` simultaneously. It should be reasonably clear that this instance obeys the *monoid laws*:

```
mempty <> x == x               -- left identity
x <> mempty == x               -- right identity
x <> (y <> z) == (x <> y) <> z -- associativity
```

It is also *commutative* (`x <> y == y <> x`) assuming that the substitutions are *disjoint* (i.e that $dom(x) \cap dom(y) = \emptyset$). In the type inference algorithm, your substitutions are all applied in order and thus should be disjoint, therefore this property should hold.

You can use this `<>` operator to combine multiple substitutions into your return substitution.

You can construct a singleton substitution, which replaces one variable, with the `=:` operator, so the substitution `("a" =: TypeVar "b") <> ("b" =: TypeVar "c")` is a substitution which replaces a with c and b with c.

The `Subst` module also includes a variety of functions for running substitutions on types, quantified types, and environments.

6 Unification

The unification algorithm is quite simple:

- *input:* two type terms t_1 and t_2, where forall quantified variables have been replaced by fresh, unique variables
- *output:* the most general unifier of t_1 and t_2 (if it exists). The unifier is a data structure or function that specifies the substitutions to take place to unify t_1 and t_2.

6.1 Unification Cases

For t_1 and t_2

1. *both are type variables v_1 and v_2:*
   - if $v_1 = v_2$, return the empty substitution
   - otherwise, return $[v_1/v_2]$
2. *both are primitive types*
   - if they are the same, return the empty substitution
   - otherwise, there is no unifier
3. *both are product types, with $t_1 = (t_{11} * t_{12})$, $t_1 = (t_{21} * t_{22})$*
   - compute the unifier S of t_{11} and t_{21}
   - compute the unifier S' of $S\ t_{12}$ and $S\ t_{22}$
   - return S <> S'
4. *function types and sum types (as for product types)*
5. *only one is a type variable v, the other an arbitrary type term t*
   - if v occurs in t, there is no unifier
   - otherwise, return $[t/v]$
6. *otherwise, there is no unifier*

Functions in the `Data.List` library are useful for implementing the occurs check.

Once you generate a unifier (also a substitution), you then need to apply that unifier to your types, to produce the unified type.

7 Errors and Fresh Names

Thus far, the following type signature would be sufficient for implementing our type inference function:

```
inferExp :: Gamma -> Exp -> (Exp, Type, Subst)
```

Unification is a partial function, however, so we need well-founded way to handle the error cases, rather than just bail out with `error` calls.

To achieve this, we'll adjust the basic, pure signature for type inference to include the possibility of a `TypeError`:

```
inferExp :: Gamma -> Exp -> Either TypeError (Exp, Type, Subst)
```

Even this, though, is not sufficient, as we cannot generate fresh, unique type variables for use as unification variables:

```
fresh :: Type -- it is impossible for fresh to return a different
fresh = ?     -- value each time!
```

To solve this problem, we could pass an (infinite) list of unique names around our program, and `fresh` could simply take a name from the top of the list, and return a new list with the name removed:

```
fresh :: [Id] -> ([Id],Type)
fresh (x:xs) = (xs,TypeVar x)
```

This is quite awkward though, as now we have to manually thread our list of identifiers throughout the entire inference algorithm:

```
inferExp :: Gamma -> Exp -> [Id] -> Either TypeError ([Id], (Exp, Type, Subst))
```

To resolve this, we bundle both the `[Id]` state transformer and the `Either TypeError x` error handling into one abstract type, called `TC` (defined in `TCMonad.hs`)

```
newtype TC a = TC ([Id] -> Either TypeError ([Id], a))
```

One can think of `TC a` abstractly as referring to a *stateful action* that will, if executed, return a value of type `a`, or throw an exception.

As the name of the module implies, `TC` is a *Monad*, meaning that it exposes two functions (`return` and `>>=`) as part of its interface.

```
return :: a -> TC a
return = ...
(>>)  :: TC a -> TC b -> TC b
a >> b = a >>= const b
(>>=)  :: TC a -> (a -> TC b) -> TC b
a >>= b = ...
```

The function `return` is, despite its name, just an ordinary function which *lifts* pure values into a `TC` action that returns that value. The function `(>>)` (read *then*), is a kind of composition operator, which produces a `TC` action which runs two `TC` actions in sequence, one after the other, returning the value of the last one to be executed. Lastly, the function `(>>=)`, more general than `(>>)`, allows the second executed action to be determined by the return value of the first.

The `TCMonad.hs` module also includes a few built-in actions:

```
typeError :: TypeError -> TC a -- throw an error
fresh     :: TC Type           -- return a fresh type variable
```

Haskell includes special syntactic sugar for monads, which allow for programming in a somewhat imperative style. Called `do` notation, it is simple sugar for `(>>)` and `(>>=)`.

```
do e                --       e
do e; v             --       e >> do v
do p <- e; v        --       e >>= \p -> do v
do let x = y; v     --       let x = y in do v
```

This lets us write unification and type inference quite naturally. A simple example of the use of the `TC` monad is already provided to you, with the `unquantify` function, which takes a type with some number of quantifiers, and replaces all quantifiers with fresh variables (very useful for inference cases for variables, constructors, and primops):

```
unquantify :: QType -> TC Type
unquantify (Ty t)       = return t
unquantify (Forall x t) = do x' <- fresh
                             unquantify (substQType (x =:x') t)
```

To run a `TC` action, in your top level `infer` function, the `runTC` function can be used:

```
runTC :: TC a -> Either TypeError a
```

Please note: This function runs the TC action with the same source of fresh names each time! Using it more than once in your program is not likely to give correct results.

8 Program structure

A program in MinHS may evaluate to any non-function type, including an aggregate type. This is a valid MinHS program:

```
main = (1,(Inl True, False));
```

which can be elaborated to the following type:

```
main :: forall t. (Int * ((Bool + t) * Bool))
     = (1,(InL True,False));
```

8.1 Type information

The most significant change to the language of assignment 1 is that the parser now accepts programs missing some or all of their type information. Type declarations are no longer compulsory! Unless you are attempting the bonus parts of the assignment, you can assume that *no* type information will be provided in the program. You must reconstruct it all.

You can view the type information after your pass using `--dump type-infer`.

9 Implementing Type Inference

You are required to implement the function `infer`. Some stub code has been provided for you, along with some type declarations, and the type signatures of useful functions you may wish to implement. You may change any part of `TyInfer.hs` you wish, as long as it still provides the function `infer`, of the correct type. The stub code is provided only as a hint, you are free to ignore it.

10 Useful interfaces

You need to use environments to a limited extent. This follows the same interface for environments you used in assignment 1, it is defined in `Env.hs`, and there are many examples throughout the program.

11 Testing

Your assignments will be autotested rigorously. You are encouraged to autotest yourself. `minhs` comes with a regress tester script, and you should add your own tests to this. Your assignment will be tested by comparing the output of `minhs --dump type-infer` against the expected abstract syntax. Your solution must be α-equivalent to the expected solution. It it up to you to write your own tests for your submission.

In this assignment we make no use of the later phases of the compiler.

12 Building MinHS

Building MinHS is exactly the same as in Assignment 1.

To run the type inference pass and inspect its results:

```
$ ./dist/build/minhs-2/minhs-2 --dump type-infer 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. Many `--dump` flags are provided, which let you see the abstract syntax at various stages in the compiler.

Similarly, if you download the Haskell for Mac version, also follow the instructions for Assignment 1.

13 Late Penalty

Unless otherwise stated if you wish to submit an assignment late, you may do so, but a late penalty reducing the maximum available mark applies to every late assignment. The maximum available mark is reduced by 10% if the assignment is one day late, by 25% if it is 2 days late and by 50% if it is 3 days late. Assignments that are late 4 days or more will be awarded zero marks. So if your assignment is worth 88% and you submit it one day late you still get 88%, but if you submit it two days late you get 75%, three days late 50%, and four days late zero.

Assignment extensions are only awarded for serious and unforeseeable events. Having the flu for a few days, deleting your assignment by mistake, going on holiday, work commitments, etc do not qualify. Therefore aim to complete your assignments well before the due date in case of last minute illness, and make regular backups of your work.

14 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. Please also consult the information on the Course Outline.

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 penalised, 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] *Haskell 2010 Language Report*, editor Simon Marlow, (2010) `http://www.haskell.org/onlinereport/Haskell2010`

[2] Robert Harper, *Programming Languages: Theory and Practice*, (Draft of Jan 2003), `http://www-2.cs.cmu.edu/~rwh/plbook/`.