2011 Haskell January Test
Type Inference

This test comprises four parts and the maximum mark is 30. Parts I, II and III are worth 28 of the 30 marks available. The **2011 Haskell Programming Prize** will be awarded for the best attempt(s) at Part IV. One bonus mark is available should you manage to complete the final question in Part IV. This will be added to your total, although your final mark will be capped at 30.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You may therefore wish to define your own tests to complement the ones provided.

1 Introduction

You will know by now that if you omit a function type declaration in a Haskell program the compiler/interpreter will work out the function's type automatically. For example, given the Haskell definitions:

```
c     = False
f x   = 1 + x
g b   = if b then 0 else 1
h x b = if x > 0 then True else b
id x  = x
```

and assuming that 0 and 1 are taken as being of type `Int`, it is relatively easy to see that their *most general* types are:

```
c  :: Bool
f  :: Int -> Int
g  :: Bool -> Int
h  :: Int -> Bool -> Bool
id :: a -> a
```

The process of working out types automatically is called *type inference* and is the subject of this exercise.

2 A Simple Language

Your task is to develop a function for inferring the type of an *expression* in a simple stripped-down language similar to Haskell. The algorithm for doing this is the famous "Algorithm W" developed by Luis Damas and Robin Milner.

The stripped-down language supports just two basic types, integers and booleans, and a small suite of *primitive* functions over them, viz. `+`, `>`, `==` and `not`. These will be used in prefix form only, i.e. there are no infix operators, and they have meanings that are similar to the equivalent functions in Haskell. Their types can be written in Haskell syntax as follows:

```
+   :: Int -> Int -> Int
>   :: Int -> Int -> Bool
==  :: Int -> Int -> Bool
not :: Bool -> Bool
```

There is no overloading, so `+`, `>` and `==` are defined to operate only on integers. Like Haskell, functions are *curried* in the sense that they take one argument at a time. The type of `+`, for example, might be written `Int -> (Int -> Int)` in Haskell syntax and an application such as `+ x 1`, equivalent to `x + 1` or `(+) x 1` in Haskell, might be written `(+ x) 1`; the meaning is the same.

2.1 Expressions

Expressions are built from basic values, i.e. integer and boolean constants, variable identifiers, primitive functions, conditionals and function applications. User-defined functions are also supported (constructor `Fun` below), but these will only feature in the last part of the exercise and can be ignored until then. The following Haskell data type will be used to represent expressions, as defined:

```
-- Ignore the Fun constructor for Parts I-III
data Expr = Number Int |
            Boolean Bool |
            Id String  |
            Prim String |
            Cond Expr Expr Expr |
            App Expr Expr |
            Fun String Expr
          deriving (Eq, Ord, Show)
```

As an example, the expression

```
if not (> x y) then x else + y 1
```

will be represented in Haskell as follows:

```
Cond (App (Prim "not") (App (App (Prim ">") (Id "x")) (Id "y")))
     (Id "x")
     (App (App (Prim "+") (Id "y")) (Number 1))
```

2.2 Types

In order to infer the type of an expression we need a representation for types. The following Haskell data type suffices:

```
-- Ignore the TVar constructor for Parts I-II
data Type = TInt |
            TBool |
            TFun Type Type |
            TErr |
            TVar String
          deriving (Eq, Ord, Show)
```

`TInt` and `TBool` represent the integer and boolean types respectively. The `TFun` constructor encodes the equivalent of the 'arrow' (`->`) function type operator in Haskell. `TErr` represents a *type error*: an expression that is badly typed, e.g. the expression `not 1`, will have its type inferred as `TErr`. `TVar` will be used to represent type *variables*, e.g. `a` and `b` in the type `a -> b -> b`, for example. However, type variables will only be relevant when we consider polymorphic types (Parts III and IV) and so can be ignored for now.

As an example, the type of the 'greater than' function, `>`, can be represented as `TFun TInt (TFun TInt TBool)`, equivalent to `Int -> Int -> Bool` in Haskell syntax. For convenience, the types of each of the primitives have been packaged up in a table (type synonym `TypeTable`) that maps identifiers (`String`s) to types thus:

```
type TypeTable = [(String, Type)]

primTypes :: TypeTable
primTypes = [("+",   TFun TInt (TFun TInt TInt)),
             (">",   TFun TInt (TFun TInt TBool)),
             ("==",  TFun TInt (TFun TInt TBool)),
             ("not", TFun TBool TBool)]
```

Note: You may find it easier to display types in the more familiar Haskell style, e.g. for debugging purposes. A function `showT :: Type -> String` has therefore been defined in the template to do this. For example,

```
Main> showT (TFun TInt (TFun TInt TBool))
"(Int -> (Int -> Bool))"
```

Do NOT attempt to define `Type` as an explicit instance of class `Show`, e.g. using `showT` as the default `show` function, as this will break the autotester.

3 Monomorphic Type Inference

The objective of type inference is simply stated: given an expression in the form of an object of type `Expr`, compute its type as an object of type `Type`. In the absence of type variables and user-defined functions this can be done fairly straightforwardly using the following rules:

1. **Constants** (Constructors `Number`, `Boolean`) The types of the integer and boolean constants are trivially known. For example, the type of `Number 6` is `TInt` and the type of `Boolean False` is `TBool`.

2. **Identifiers** (Constructor `Id`) The type of an identifier is given by a supplied *type environment* which is simply a table that associates variable identifiers in expressions with types. Note that the type of this table is exactly the same as that of `TypeTable` above, hence the type synonym:

   ```
   type TEnv = TypeTable    -- i.e. [(String, Type)]
   ```

   Thus, for example, if the expression is `Id "x"` and the environment contains a binding `("x",TInt)` then the expression can be inferred to have type `TInt`. A simple table lookup is all that's required for this case.[1]

3. **Primitives** (Constructor `Prim`) The type of a primitive function is determined by looking up its identifier in the `primTypes` table above. For example, if the expression is `Prim "=="` the inferred type will be `TFun TInt (TFun TInt TBool)`.

4. **Conditionals** (Constructor `Cond`) For a conditional expression to be correctly typed the type of the condition, i.e. the first argument of `Cond`, must be a boolean (`TBool`) and the type of the 'then' and 'else' alternatives, i.e. the second and third arguments of `Cond` respectively, can be any type so long as they are the same. In the absence of type variables these properties can be established by simple equality tests on the inferred types of the three components, each of which can be computed recursively. If any of these properties is violated then the inferred type of the conditional is `TErr`; otherwise it is the inferred type of either the two alternatives, as their types will be the same.

5. **Applications** (Constructor `App`) In the absence of type variables and user-defined functions the type of a function application can be inferred straightforwardly from the types of its two components, i.e. the function and argument expressions, computed recursively. For the expression to be correctly typed the recursively computed function type must be of the form

[1] Note that it is only possible to introduce an identifier via a user-defined function (constructor `Fun` above). However, for testing purposes an expression will be allowed to contain any identifier provided the type of that identifier is given in the type environment.

	Subexpression	Inferred type	Comment
1	`Number 1`	`TInt`	By rule 1
2	`Id "x"`	`TInt`	By rule 2
3	`Id "y"`	`TInt`	By rule 2
4	`Prim "+"`	`TFun TInt (TFun TInt TInt)`	By rule 3
5	`App (Prim "+") (Id "y")`	`TFun TInt TInt`	See *
6	`App (App (Prim "+") (Id "y")) (Number 1)`	`TInt`	See **

* Because `Prim "+"` has inferred type `TFun TInt (TFun TInt TInt)` (line 4) and the inferred type of `(Id "y")` (line 3) matches the underlined domain type of `+`, i.e. `TInt`.
** Because `App (Prim "+") (Id "y")` has type `TFun TInt TInt` (line 5) and the inferred type of `Number 1` (line 1) matches the underlined domain type `TInt`.

Figure 1: Example of type inference.

`TFun t t'` and the recursively computed argument type must be `t`, i.e. the same as the domain type of the function. If these properties hold then the application as a whole has type `t'`. In all other cases the expression is incorrectly typed and the inferred type will be `TErr`.

Note that the types inferred at this point are called *monomorphic* because they do not contain type variables.

To illustrate the first three rules, Figure 1 shows a breakdown of how they apply to each subexpression in the 'then' and 'else' alternatives, i.e. the second and third arguments of the `Cond` constructor respectively, in the example expression in Section 2.1 above. In each case the type environment is assumed to be `[("x", TInt), ("y", TInt)]`, which asserts that the identifiers `"x"` and `"y"` are both integers.

To illustrate rule 4 consider now how to infer the type of the outer `Cond` expression in the example. The condition has inferred type `TBool`, as required, and the two alternatives have the *same* inferred type (`TInt` in this case, from lines 2 and 6), again as required. Thus, the expression as a whole has type `TInt`, i.e. the type inferred for both the 'then' and 'else' alternatives.

At this point you are now in a position to answer Parts I and II of the exercise. You may wish to tackle these before proceeding to the next Section.

4 Polymorphic Type Inference – enter the `TVars`

In order to infer the type of expressions involving user-defined functions (`Fun` constructor) we need, in general, to be able to handle types containing type *variables*. When we allow type variables the question as to whether two types are "the same" no longer boils down to a simple test for syntactic, i.e. structural, equality, as above. Instead we have to ask whether the two types are *unifiable*, i.e. whether they can be 'made the same' by a suitable substitution of the type variables they contain. As an example, suppose that when typing a conditional the 'then' and 'else' alternatives are inferred to have the (function) types

```
TFun TInt (TVar "a")
```

and

```
TFun (TVar "b") TBool
```

respectively. These are clearly not syntactically the same, but they are *unifiable*, i.e. the types of the two alternatives can be made the same by associating the type variable `"a"` with the type `TBool` and, similarly, `"b"` with `TInt`. In this sense the two alternatives are consistently typed. A set of type associations of this sort is called a type *substitution* and will be represented as a list of `(String, Type)` pairs, which again is another form of `TypeTable`[2]:

```
type Sub = TypeTable    -- i.e. [(String, Type)]
```

As an example, the substitution above might be represented by the list `[("a",TBool), ("b",TInt)]`, although the order of the elements in the list is not significant.

The result of the unification of two types will be an object of type `Sub`, if a unifying substitution can be found. If not then the unification *fails* – see below for details of how to handle this.

4.1 The Martelli-Montanari Unification Algorithm

In this exercise you are going to use the Martelli-Montanari unification algorithm to unify two types. The algorithm actually operates on a *list* of *pairs* of types of the form

```
[(t1, t1'), (t2, t2'), ..., (tn, tn')]
```

and a substitution (`Sub`) which is initially empty (`[]`) and which 'grows' as the algorithm proceeds. The order of the elements in the list of type pairs, and indeed the order of the two elements within each type pair, is unimportant. The algorithm generates as its result either a unifying substitution, or failure.

To unify two types `t` and `t'` we initialise the algorithm with the singleton list `[(t,t')]` and the empty substitution `[]`. The algorithm repeatedly examines the type pair at the head of the list and applies the following rules to the two types contained therein:

- Two `TInt`s unify successfully in which case the pair `(TInt,TInt)` is essentially 'discarded' and the algorithm recurses on the remaining type pairs in the list. The substitution is unmodified. A similar rule applies in the case of two `TBool`s.
- Two type variables `TVar v` and `TVar v'` unify successfully if `v == v'`, in which case the algorithm recurses on the remaining type pairs and an unmodified substitution. In all other cases where at least one of the types is a type variable, `TVar v`, say, and the other some type `t` $\neq$ `TVar v`, the binding `(v,t)` is added to the substitution list (e.g. using `:`) and the remaining pairs in the list of type pairs are *updated* by applying the *singleton* substitution `[(v,t)]` to *both* types within each pair. For example, if the head of the list is the pair `(TVar "a", TInt)` and the remaining elements, i.e. with the head removed, are

  ```
  [(TBool, TBool), (TFun (TVar "a") (TVar "b"), TVar "c"), (TInt, TVar "a")]
  ```

  then the algorithm recurses with the updated list

  ```
  [(TBool, TBool), (TFun TInt (TVar "b"), TVar "c"), (TInt, TInt)]
  ```

[2]Note that the structure of a `Sub` is the same as the previous `TEnv` except that the `String`s in the table represent type variable names rather than identifiers.

and with the substitution updated with `("a", TInt)`. Notice that all occurrence of `TVar "a"` have been replaced with `TInt` in the list of type pairs. An important exception is when the variable `v` appears *anywhere* in `t`, in which case the unification fails. This is the only exception. This test is sometimes called the *occurs check* and prevents infinite loops from occurring during type inference. An example of this is given later on.

- Given two function types, `TFun t1 t2` and `TFun t1' t2'`, say, the pairs `(t1, t1')` and `(t2, t2')` are added to the remaining elements in the list (e.g. using `:` or `++`) and the algorithm recurses. Note that the two pairs can be added anywhere in the list, although the head end will suffice. The substitution is unmodified.
- In all other cases the unification fails.

Termination: If the list of type pairs becomes empty (`[]`) when the substitution is `s` then the unification *succeeds* and the resulting substitution is `s`.

4.1.1 Coping with Failure

The unification process needs to distinguish between a successful unification, which results in a (possibly empty) substitution, and a unification failure. A nice way to handle this is to use Haskell's `Maybe` data type:

```
data Maybe a = Nothing | Just a
```

to distinguish the two. The result of a unification in this exercise will thus be an object of type `Maybe Sub`, where `Nothing` represents unification failure and `Just s` represents the result of a successful unification with the unifying substitution being `s`.

5 What to Do

There are four parts to this exercise. Most of the marks are allocated to the first three parts. The last part is extremely hard and should only be attempted if you have completed the first three.

5.1 Part I

1. Define two functions:

   (a) `lookUp :: Eq a => a -> [(a, b)] -> b` that will look up the binding of a given item in a list of (item, value) pairs. A precondition is that the search item is in the given list. For example, `lookUp "x" [("hello",1),("x",9),("dolly",1)]` should return `9` and `lookUp "not" primTypes` should return `TFun TBool TBool`.

   (b) `tryToLookUp :: Eq a => a -> b -> [(a, b)] -> b` that will look up the binding of a given item (of type `a`) in a given list of (item, value) pairs. If the item is not contained in the list the result is the *default* value (of type `b`) provided. For example, `tryToLookUp "k" 0 [("hello",1),("x",9),("dolly",1)]` should return `0`.

   Hint: you can optionally define `lookUp` in terms of `tryToLookUp` by passing the 'dummy' value `undefined` in place of the default value.

   [3 marks]

2. Define a function `reverseLookUp :: Eq b => b -> [(a, b)] -> [a]` that, given a value and a list of (item, value) pairs, will generate the list of all items with that value. For example, `reverseLookUp 1 [("hello",1),("x",9),("dolly",1)]` should return the list `["hello","dolly"]` and `reverseLookUp (TFun TInt (TFun TInt TBool)) primTypes` should return `[">","=="]`.

[2 marks]

3. Define a function `occurs :: String -> Type -> Bool` that will return `True` iff a given type variable identifier (`String`) occurs in a given type (`Type`). For example, `occurs "x" (TVar "y")` and `occurs "x" TBool` should both return `False` and `occurs "x" (TFun TBool (TVar "x"))` should return `True`. Note that a variable `v` occurs in `TFun t t'` if it occurs in either of `t` or `t'`.

[3 marks]

5.2 Part II

Remark: If you get stuck in this part of the exercise you might wish to take a break by tackling question 1 of Part III before continuing.

Define a function `inferType :: Expr -> TEnv -> Type` that, given an expression and a type environment, will infer the type of the given expression using the rules outlined in Section 3 above. For the case of identifiers, e.g. `Id i`, a precondition is that there will be a binding for `i` in the given type environment.

IMPORTANT: The expression is guaranteed not to contain any user-defined functions at this point, so you do *not* need a rule for the `Fun` constructor.

Hint: You might find it useful to define a helper function, `inferApp`, say, to infer the type of a function application, `App f a`, say, in terms of the inferred types of the function (`f`) and argument (`a`), computed recursively.

The supplied template includes several examples of expressions, together with their types, that you can use for testing purposes. A sample type environment is also provided:

```
env :: TEnv
env = [("x",TInt),("y",TInt),("b",TBool),("c",TBool)]
```

The examples given in the template assume the type environment `env` and are as shown in Table 1. You can use these, and the examples in Section 3 above, in part to test your code.

[10 marks]

5.3 Part III

This part of the exercise requires you to handle type variables (constructor `TVar`) and to implement the type unification function as described in Section 4.

1. Using `tryToLookUp` defined earlier, define a function `applySub :: Sub -> Type -> Type` that will apply a given type substitution, `s` say, to a given type, `t` say. To do this you need to locate each type variable of the form `TVar v` in `t` and then replace it with the binding for `v` in `s`, if there is one; if not, the type variable should be left unmodified. For example, if the given substitution is `s = [("a",TBool),("b",TFun TBool TInt)]` then: `applySub s TInt` should return `TInt` – this is a base case, so no substitution is needed; `applySub s`

e	inferType e env
`Number 9`	`TInt`
`Boolean False`	`TBool`
`Prim "not"`	`TFun TBool TBool`
`App (Prim "not") (Boolean True)`	`TBool`
`App (Prim ">") (Number 0)`	`TFun TInt TBool`
`App (App (Prim "+") (Boolean True)) (Number 5)`	`TErr`
`Cond (Boolean True) (Boolean False) (Id "c")`	`TBool`
`Cond (App (Prim "==") (Number 4)) (Id "b") (Id "c")`	`TErr`

Table 1: Some example expressions and inferred types.

t	t'	unify t t'
`TFun (TVar "a") TInt`	`TVar "b"`	`Just [("b",TFun (TVar "a") TInt)]`
`TFun TBool TBool`	`TFun TBool TBool`	`Just []`
`TFun (TVar "a") TInt`	`TFun TBool TInt`	`Just [("a",TBool)]`
`TBool`	`TFun TInt TBool`	`Nothing`
`TFun (TVar "a") TInt`	`TFun TBool (TVar "b")`	`Just [("b",TInt),("a",TBool)]`
`TFun (TVar "a") (TVar "a")`	`TVar "a"`	`Nothing`

Table 2: Unification examples.

`(TFun TBool (TVar "a"))` should return `TFun TBool TBool`, i.e. with `TVar "a"` replaced with `TBool` by virtue of the binding for `"a"` in `s`; and `applySub s (TVar "c")` should return `TVar "c"`, as there is no binding for `"c"` in `s`.

[2 marks]

2. Define a function `unifyPairs :: [(Type, Type)] -> Sub -> Maybe Sub` that implements the Martelli-Montanari algorithm described in Section 4.1. Use the `applySub` function from 1. above to apply the (singleton) substitution as described in Section 4.1 (`TVar` case). Don't forget to include the *occurs check*, which you should implement using the `occurs` function from Part I.

   The top-level unification function (`unify`) for invoking the algorithm is provided for you in the template:

   ```
   unify :: Type -> Type -> Maybe Sub
   unify t t'
     = unifyPairs [(t, t')] []
   ```

   The template also contains a number of unification examples for illustration and testing purposes, as shown in Table 2. Note that the last example fails because of the occurs check; indeed it would fail even with a *single* occurrence of `TVar "a"` anywhere in `t`. Note also that the order of the elements in the resulting substitution is unimportant for correctness, but it will simplify testing if you stick to the order given.

   [8 marks]

5.4 Part IV

Using your unification function above, develop a new *polymorphic* type inference function that will infer the type of an arbitrary expression that may include user-defined functions. Anonymous functions are represented using the `Fun` constructor: the expression `Fun "x" e` is the representation of the function `\x -> e` in Haskell syntax, i.e. the unnamed function of `x` that computes the value of the expression `e`.

When attempting this question do NOT modify the `inferType` function in any way. Instead define a new function `inferPolyType :: Expr -> Type` that returns the type of a given expression. To do this, you should consider using a helper function with a type such as:

```
inferPolyType' :: Expr -> TEnv -> Int -> (Sub, Type, Int)
```

or

```
inferPolyType' :: Expr -> TEnv -> [String] -> (Sub, Type, [String])
```

Both are defined in the template. The first two arguments should be self-explanatory. The result includes not only the inferred expression type, but also the substitution for any type variables in the given type environment that leads to the inferred type. We'll discuss the additional argument and result component shortly.

The rules for the `Number`, `Boolean` and `Prim` cases are straightforward translations of those in `inferType`. You should first attempt to implement the rule for `Fun`, which can be tested independently; then have a go at `App`. You are NOT required to implement the rule for `Cond` for credit, but you are welcome to try, especially if you are competing for the prize.

You need to understand the role of the substitution (`Sub`) in the result. Its job is to 'add information' about the type variables in the type environment. For example, if the type environment includes the binding `("x", TVar "a")`, which just says that `x` has some type `a`, and we use this to type the expression `App (Prim "+") (Id "x")` then the substitution that's returned by the `App` rule will include the binding `("a",TInt)` – see below for details of how. This is *new* information that we didn't have before and is computed in part by unification. If at any point we need to call `inferPolyType'` again we do so with a modified type environment that's obtained by applying this substitution to each type in the given environment. A function `updateTEnv :: TEnv -> Sub -> TEnv` has been defined in the template for this purpose. In the example, the call to `updateTEnv` would replace the binding `("x", TVar "a")` with `("x",TInt)`, which says that we now know that `"x"` has type `TInt`.

1. Implement the rules for the base cases (no credit, but needed for the other cases) and the case for the `Fun` constructor (2 marks). To implement the `Fun x e` case you need to add a new binding `(x, TVar a)` to the given type environment, where `a` is a *unique* type variable name (`String`), and then use that to infer the type of `e`. The result type will be of the form `TFun <something> te`, where `te` is the type of `e`. The `<something>` would be `TVar a` but, crucially, the substitution that's returned from the recursive call may include a substitution for `a`. You therefore need to apply the substitution to `TVar a` before returning it – use your `applySub` function defined earlier. Of course, if the type of `te` is `TErr` then the result should be `TErr`, *not* `TFun (TVar "a1") TErr`, for example.

   You may now see the role of the extra `Int` (or `[String]`) argument: the type variable names have to be *unique* so the suggestion is that you use the names `a1`, `a2`, `a3`..., or similar. This requires you to pass in, and hence out, the next available type variable identifier – this might be either in the form of an integer (the `Int` argument) that you can use to build a name, or a list (possibly infinite?) of pre-made names (the `[String]` argument). Some additional test

expressions are defined in the template file that you can use for testing. Note: the identifiers you choose for your type variables may not align with those given in the examples, but this is not important so long as the results agree up to an arbitrary consistent renaming.

[2 marks] (`Fun` rule)

2. Now have a go at the `App` rule. This requires making two recursive calls to `inferPolyType'`. For the second call you need to update the given type environment using the substitution returned from the first call, using `updateTEnv`.

   Unification is essentially used in place of equality in the `App` rule in `inferType`. Given `App f e` you first need to infer the type of `f` and `e` and then *unify* the type inferred for `f` with the type `TFun te (TVar a)`. Here, `te` is the inferred type of `e` and `a` is another unique type variable identifier. Remember that you need to update the type environment (`updateTEnv`) for the second call to `inferPolyType'`, as described above. The substitution you need to return in the result is derived by *combining* the substitutions that come from the two recursive calls to `inferPolyType'` *and* the unifiying substitution. A function `combineSubs` to do this has been defined for you in the template. Make sure you combine the substitutions in the right order (see the comment in the code). The result type is the result of applying the unifying substitution to the type variable `TVar a` that you chose earlier. Note that if the unification fails then the result type should be `TErr`. The test expressions `ex11`...`ex14` can be used in part to test your `App` rule.

   [1 BONUS mark] (`App` rule) that will be added to your total

3. If you get this far, have a go at the `Cond` case. There are no hints, so you'll have to work it out from what you've learnt so far. There is no credit for this question — it's for prize candidates only. Good luck!