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.
- (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 front end and typechecker are provided for you, along with an (trivial) interpreter. You do not have to change anything in any module other than TyInfer.hs (even for the bonus part).

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 $\alpha$-renaming of type variables.

Please consult the message board forum for Assignment 2 on the class web page, and send bug reports about the assignment spec to sjw.

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 letrecs)
- 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 will appear here soon.

2 Bonus Task

The bonus task is optional, and is worth a bonus 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:

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

3 Aggregate Data Types

This section covers the extensions to the language of the first assignment. In all other respects 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.

<table>
<thead>
<tr>
<th>Types</th>
<th>( \tau \rightarrow \tau_1 \times \tau_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>Unit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expressions</th>
<th>( \text{exp} \rightarrow (e_1, e_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>( \text{fst} \ e \</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>()</td>
</tr>
</tbody>
</table>

3.2 Sum Types

<table>
<thead>
<tr>
<th>Types</th>
<th>( \tau \rightarrow \tau + \tau )</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Expressions</th>
<th>( \text{exp} \rightarrow \text{Inl} \ e_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>( \text{Inr} \ e_2 )</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>( \text{case} \ e ) of</td>
</tr>
<tr>
<td></td>
<td>( \text{Inl} \ x \rightarrow e_1 ) ;</td>
</tr>
<tr>
<td></td>
<td>( \text{Inr} \ y \rightarrow e_2 ) ;</td>
</tr>
</tbody>
</table>

3.3 Polymorphism

The extensions to allow polymorphism are relatively simple, two new type form has been introduced, the TyVar \( t \) type, and the Forall \( t \ e \) type.
Types \( \tau \rightarrow \forall \tau \ldots \tau \ldots \)

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

```haskell
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 ());
```

3
4 Type Inference Rules

Constants and Variables

\[
\begin{align*}
\{\{\} \vdash n : \text{Int} & \quad \{\{\} \vdash \text{True} : \text{Bool} & \quad \{\{\} \vdash \text{False} : \text{Bool} & \quad \{\} \vdash x : \tau
\end{align*}
\]

Constructors

\[
\begin{align*}
TT \vdash e_1 : \tau_1 & \quad T' \mid TT \vdash e_2 : \tau_2 & \quad T' \mid TT \vdash e_1 : \tau_1 & \quad TT \vdash e_2 : \tau_2
\end{align*}
\]

\[
\begin{align*}
T' \mid TT \vdash \text{pair} (e_1, e_2) & : (T' \tau_1 \ast \tau_2) & \quad TT \vdash \text{Inl} (e_1) & : (\tau_1 + \tau_2) & \quad TT \vdash \text{Inr} (e_2) & : (\tau_1 + \tau_2)
\end{align*}
\]

Primops

\[
\begin{align*}
op & : \tau \in \Gamma & \quad T_1 \Gamma \vdash e_1 : \tau_1 & \quad \ldots & \quad T_n \Gamma \vdash e_n : \tau_n & \quad \tau \vdash (T_n \ldots T_2 \tau_1) \rightarrow \ldots (T_n \tau_{n-1}) \rightarrow \tau_n \rightarrow \alpha & \quad \alpha \ \text{fresh}
\end{align*}
\]

If-Then-Else

\[
\begin{align*}
T \Gamma \vdash e \vdash \tau & \quad T \Gamma \vdash e_1 \vdash \tau_1 & \quad T \Gamma \vdash e_2 \vdash \tau_2 & \quad U \Gamma \vdash \text{if} (e, e_1, e_2) \vdash U \tau_2
\end{align*}
\]

Case

\[
\begin{align*}
U \Gamma \vdash e \vdash \tau & \quad T \Gamma \vdash e \vdash \tau_1 & \quad T' \Gamma \vdash e_1 \vdash \tau_1 & \quad T' \Gamma \vdash e_2 \vdash \tau_2 & \quad T' \Gamma \vdash e \vdash \tau
\end{align*}
\]

\[
\begin{align*}
U \Gamma \vdash \text{case} (e, x, e_1, y, e_2) & : U \tau
\end{align*}
\]

Application

\[
\begin{align*}
TT \vdash e_1 : \tau_1 & \quad T'T \vdash e_2 : \tau_2 & \quad T' \tau_1 \vdash \tau_2 \rightarrow \alpha & \quad \alpha \ \text{fresh}
\end{align*}
\]

Recursive Functions

\[
\begin{align*}
U \Gamma \vdash \text{letfun} (f, x, e) & : U (T \alpha \rightarrow \tau)
\end{align*}
\]

Let Bindings

\[
\begin{align*}
TT \vdash e_1 : \tau & \quad T' (TT \cup \{ x : \text{Generalise} (TT, \tau) \}) \vdash e_2 : \tau' & \quad TT \vdash e : \tau & \quad \Gamma \vdash \text{main} (e) : \text{Generalise} (TT, \tau)
\end{align*}
\]

where \( \text{Generalise} (\Gamma, \tau) = \forall (TV (\tau) \setminus TV (\Gamma)) \cdot \tau \)

\( \forall \text{-elimination} \)

\[
\begin{align*}
x : \forall a_1 \ldots \forall a_n, \tau \in \Gamma & \quad \{\} \vdash x : [\beta_1 / a_1] \ldots [\beta_n / a_n] \tau, & \quad \beta_i \ \text{fresh}
\end{align*}
\]

4
5 Unification

The unification algorithm is quite simple:

- **input:** two type terms $t_1$ and $t_2$, where for all 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 $t_2$.

5.1 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 $t_1 = (t_{11} * t_{12})$, $t_2 = (t_{21} * t_{22})$
   - compute the unifier $S$ of $t_{11}$ and $t_{21}$
   - compute the unifier $S'$ of $S t_{12}$ and $S t_{22}$
   - return $S' \cup S$

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

6 Errors

Unification is a partial function. We need a way to handle the error cases. The module Error.hs provides phasefail and pprPhaseFail which are useful for printing error messages, as well as the fragment of code that produced the error.

7 Internal representation of constructors

The abstract syntax of MinHS has been extended to represent the new data constructors for sum and product types. Previously, only the constructors True and False were allowed. Now, however, constructors may take arguments. E.g. InL 1 is the constructor for the “left” value of a sum type. (1,False) is the constructor for a 2-tuple, while () is the constructor for the 0-tuple, also known as “unit”. Constructors are distinguished via their tags.
8 Program structure

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

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

which can be elaborated to the following type:

```haskell
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 part 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 before and after your pass using --dump-rawparse and --inf.

9 Implementing Type Inference

You are required to implement the function `elaborate`. 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 `elaborate`, of the correct type. The stub code is provided only as a hint, you are free to ignore it.

10 Fresh Names

Type inference requires a mechanism to generate unique variable names. To do this we make use of 'monadic' programming to thread a unique name generator around the type inference pass. You can mostly ignore the details of the monad, except that you must program the `tyInf` function using do-notation.

To generate a fresh variable name, you can call the function `freshTyVar` in TcMonad.hs. This returns a fresh type variable different to those already in the program.

11 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 `TcMonad.hs`, and there are many examples throughout the compiler.

The `phasefail` and `pprPhaseFail` functions are useful if you detect any error conditions. These are defined in `Error.hs`.

Some useful functions defined over types are available in `Type.hs`, including:

```haskell
resultType :: Type -> Type
argTypes :: Type -> [Type]
```

these are good for extracting the components of function types.
12 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-rawinfer against the expected abstract syntax.
Your solution must be $\alpha$-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.

13 Building MinHS

To run the type inference pass and inspect its results:

```
$ ./minhs --inf foo.mhs
```

alternatively, without arguments MinHS will read from standard input:

```
$ ./minhs --inf
main = 1;
```

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.

14 Late Penalty

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

15 Plagiarism

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

References


http://www-2.cs.cmu.edu/~rwh/plbook/