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, typechecker, and an interpreter backend are provided for you, although the interpreter doesn’t evaluate unit, pair and sum types. 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 course message board for Assignment 2 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 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 \ast \tau_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \mid ) Unit</td>
</tr>
</tbody>
</table>

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

3.2 Sum Types

| Types       | \( \tau \rightarrow \tau + \tau \) |

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

```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 ());
```
4 Type Inference Rules

Constants and Variables

\[ \Gamma \vdash n : \text{Int} \quad \Gamma \vdash \text{True} : \text{Bool} \quad \Gamma \vdash \text{False} : \text{Bool} \quad \Gamma \vdash x : \tau \]

Constructors

\[ TT \vdash e_1 : \tau_1 \quad T'\Gamma \vdash e_2 : \tau_2 \]
\[ T'\Gamma \vdash \text{pair}(e_1, e_2) : (T'\tau_1 + \tau_2) \]
\[ TT \vdash \text{Inl}(e_1) : (\tau_1 + \alpha) \]
\[ TT \vdash \text{Inr}(e_2) : (\alpha + \tau_2) \quad \alpha \text{ fresh} \]

Primops

\[ \Gamma \vdash \text{op} : \tau \quad T_1 \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad T_n \Gamma \vdash e_n : \tau_n \]
\[ \frac{\quad UT_1 \ldots T_n \Gamma \vdash \text{op}(e_1, \ldots, e_n) : U\alpha}{UT_n \ldots T_1 \Gamma \vdash \text{op}(e_1, \ldots, e_n) : U\alpha} \quad \alpha \text{ fresh} \]

If-Then-Else

\[ TT \vdash e : \tau \quad \tau \vdash \text{Bool} \quad T_1UT \vdash e_1 : \tau_1 \quad T_2T_1UT \vdash e_2 : \tau_2 \quad T_2\tau_1 \vdash \tau_2 \]
\[ UT_2T_1UT' \vdash \text{if}(e, e_1, e_2) : U'\tau_2 \]

Case

\[ TT \vdash e : \tau \quad T_1T(\Gamma \cup \{x : \alpha_1\}) \vdash e_1 : \tau_1 \quad T_2T_1T(\Gamma \cup \{y : \alpha_r\}) \vdash e_2 : \tau_r \quad T_2T_1T(\alpha_1 + \alpha_r) \vdash \tau_1 \quad T_2T_1\tau \vdash \tau_2 \quad U'T_2T_1UT' \vdash \text{case}(e, x, e_1, y, e_2) : U'U'\tau_r \quad \alpha_1, \alpha_r \text{ fresh} \]

Application

\[ TT \vdash e_1 : \tau_1 \quad T'\Gamma \vdash e_2 : \tau_2 \quad T'\tau_1 \vdash \tau_2 \rightarrow \alpha \quad \alpha \text{ fresh} \]
\[ UT'TT \vdash \text{apply}(e_1, e_2) : U\alpha \]

Recursive Functions

\[ T(\Gamma \cup \{x : \alpha_1\} \cup \{f : \alpha_2\}) \vdash e : \tau \quad T\alpha_2 \vdash T\alpha_1 \rightarrow \tau \]
\[ UT \vdash \text{letfun}(f, x, e) : U(T\alpha_1 \rightarrow \tau) \quad \alpha_1, \alpha_2 \text{ fresh} \]

Let Bindings

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

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

\( \forall \)-elimination

\[ x : \forall a_1 \ldots \forall a_n : \tau \in \Gamma \]
\[ \Gamma \vdash x : [\beta_1 / a_1] \ldots [\beta_n / a_n] \tau, \quad \beta_i \text{ fresh} \]
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 as output. Consider, for example, the pair rule. The environment $\Gamma$ and the expression $\text{pair}(e_1, e_2)$ are the arguments to the type inference. To construct the type of the pair, first the type of $e_1$ is constructed. This corresponds to a recursive call to the type inference function, now with $\Gamma$ and $e_1$ as input. It returns $\tau_1$ and the substitution $T$ as result. We then construct the type of $e_2$, with the environment $TT$ as input, and substitution $T'$ and type $\tau_2$ as result. The result of typing the pair-expression is then the substitution $T'T$ and the type $T\tau_1 * \tau_2$. There is, however, a problem if we implement it exactly like that for the type inference algorithm in this assignment: if we add type annotations to the expressions on the way (say, when typing $e_1$), those type annotations wouldn’t contain the information we get from typing successive expressions (e.g. $e_2$). Consider the following example:

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

When typing $\text{let z = x in x}$, we only know that $x$ is of type $a$, so we would add the type annotation $\text{let z :: a = x in x}$. Only when we type the second expression, $x+1$, do we know that $a$ has to be $\text{Int}$ (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 (this is what the functions stubs substBnd, substExp, and substAlt are for).

5 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 $t_2$.

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

\[
\text{main} = (1, \text{InL True, False});
\]

which can be elaborated to the following type:

\[
\text{main :: forall } t. \text{ (Int * ((Bool + t) * Bool))} = (1, \text{InL True, False});
\]

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 after your pass using --dump-infer or --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. The function `getFreshTyVar`, which is already defined in `TyInfer.hs`, calls this function and converts the resulting fresh variable into a type variable.

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 α-equivalent to the expected solution. It is up to you to write your own tests for your submission.

If you want to look at the type annotations during testing, the option `--inf` is more useful than `--dump-rawinfer`, as it pretty prints the code with the type annotations.

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:

```bash
$ ./minhs --inf foo.mhs
```
alternatively, without arguments MinHS will read from standard input:

```bash
$ ./minhs --inf
main = 1;
^D
```

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

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.

15 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 following very useful guide by the Learning Centre How Not To Plagiarise http://www.lc.unsw.edu.au/plagiarism/ and the CSE Plagiarism policy http://www.cse.unsw.edu.au/~chak/plagiarism/plagiarism-guide.html.

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 soley 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
