Types in Haskell

As we know...
▶ Haskell is *strongly typed*, so incorrectly typed programs are rejected at compile time
▶ Haskell is *polymorphically typed*, so types may be universally quantified over sets of type variables.
▶ The Haskell compiler is capable of *automatic type inference* to discover the types of functions.

For completeness we note:
▶ Haskell functions can also take sets of types to provide a kind of structured overloading. This facility is provided via *type classes*.
▶ We will ignore this today, but note that everything discussed today generalises easily to cover this.

The type inference system

Today we will introduce some of the ideas behind Haskell’s type inference mechanism. The Haskell type checker must be able to
1. Determine whether the program is well typed
2. If so, determine the type of any expression in the program.

The programmer (presumably) believes their program is well typed. They have an idea for the type of *each expression* in the program. They could, in fact, have labelled each expression with that type. In a sense, the job of the type checker is to recover the lost labels on the parts of the program.

An example

Take a simple expression:

\( \lambda x \ y \ z \rightarrow x \ z \ (y \ z) \)

which we might lay out in a tree as:

```
    @
   /|
  @ x
 /|
@ y
 /|
@ z
 /|
@ (y z)
```

We can decorate the tree with some labels, standing in for the types that we want to discover (we can call these *type variables*):

```
    @
   /|
  @ x
 /|
@ y
 /|
@ z
 /|
@ (y z)
```
Drawing the tree again

For convenience we will re-draw the tree in a more compact notation (this notation will also be a help, later on, in formalizing the rules).

\[
\begin{array}{c}
x :: T_0  \\
z :: T_1 \\
\hline
T_4  \\
y :: T_2  \\
z :: T_3 \\
\hline
T_5 \\
\hline
T_6 \\
\hline
T_7 \\
\end{array}
\]

\[\lambda x y z\]

Inferences

For any sub-part of the tree shaped like this:

\[
\begin{array}{c}
E_0 :: A  \\
E_1 :: B \\
\hline
C \\
\end{array}
\]

we know that we must conclude that \(C\) is an arrow type (because that is the meaning of application). So we produce the equation:

\[C = A \rightarrow B\]

So from our tree we get the set of equations:

\[
\begin{align*}
T_0 &= T_1 \rightarrow T_4 \\
T_2 &= T_3 \rightarrow T_5 \\
T_4 &= T_5 \rightarrow T_6
\end{align*}
\]

Inferences

Substituting these back into the tree:

\[
\begin{array}{c}
x :: T_1 \rightarrow T_4  \\
z :: T_1 \\
\hline
T_5  \\
y :: T_3 \rightarrow T_5  \\
z :: T_3 \\
\hline
T_5 \\
\hline
T_6  \\
\hline
T_7 \\
\end{array}
\]

\[\lambda x y z\]

What to do with \(z\)? There are two type variables, \(T_1\) and \(T_3\) associated with the two instances of \(z\).

Inference

We can either assume that \(T_1\) and \(T_3\) must be equal (in which case we can add these equations):

\[
\begin{align*}
T_1 &= T_3 \\
T_7 &= T_0 \rightarrow T_2 \rightarrow T_1 \rightarrow T_6
\end{align*}
\]

The second equation comes from our knowledge of the meaning of lambda.

\[
\begin{array}{c}
x :: A  \\
E :: B \\
\hline
A \rightarrow B \\
\hline
\lambda E.x \\
\end{array}
\]

(putting it more loosely, we are looking for sub-graphs of this shape:

\[
\begin{array}{c}
E_0 :: A \\
\vdots \\
\vdots \vdots \vdots \vdots \vdots \vdots \\
B \\
\hline
A \rightarrow B \\
\hline
\lambda E_0 \\
\end{array}
\]
Inference

Substituting back we would then get:

\[
\begin{array}{c}
\frac{x : T_1 \rightarrow T_4 \quad z : T_3 \quad y : T_3 \rightarrow T_5}{(T_1 \rightarrow T_5) \rightarrow (T_5 \rightarrow T_6) \rightarrow T_6} \\
\end{array}
\]

\[\lambda x \ y \ z \quad \] (\(T_1 \rightarrow T_5 \rightarrow T_6\)) \rightarrow (T_1 \rightarrow T_5) \rightarrow T_1 \rightarrow T_6

Note that this forces us to conclude that all occurrences of a lambda-bound variable have the same type. We must make this a requirement if we are to be sure that the lambda abstraction will be usable in all contexts.

Type environment

We represent the set of “known facts” about variables by a mapping, \(\Gamma\), from names to types, which we call the “type environment”. All of the globally defined names in the program have entries:

\[
\begin{align*}
1, 2, 3 & : Int \\
\text{map} & : (T_0 \rightarrow T_1) \rightarrow [T_0] \rightarrow [T_1]
\end{align*}
\]

When dealing with local definitions we use an augmented map. During compilation the map is updated when a functions type has been determined.

Type Rules

The standard type rules refer to this environment. Some key rules:

\[\Gamma \cup \{x : t\} \vdash x : t \quad \text{VAR}\]

\[\Gamma e : t' \rightarrow t \quad \Gamma e' : t' \quad \Gamma \vdash e e' : t \quad \text{APP}\]

\[\Gamma \cup \{x : t'\} \vdash e : t \quad \Gamma \vdash \lambda x . e : t' \rightarrow t \quad \text{ABS}\]

\[\Gamma \vdash p : \sigma \quad \Gamma \cup \{x : \sigma\} \vdash p e' : \tau \quad \Gamma \vdash p \ \text{let} \ x = e \ \text{in} \ e' : \tau \quad \text{LET}\]

Type environments

We often know some facts about names that are used in the expressions. For example, in the expression:

\[
\begin{align*}
\text{map} & : T_0 \\
f & : T_1 \\
1, 2, 3 & : T_3 \\
T_4 & : T_5
\end{align*}
\]

We actually know that (for example):

\[
\begin{align*}
T_6 & = \text{Int} \\
T_0 & = (T_7 \rightarrow T_8) \rightarrow [T_7] \rightarrow [T_8]
\end{align*}
\]
Graph representations of programs

From our examination of copying we recall that functional programs may be represented as graphs:

\[ x + \text{sum} \, \text{xs} \]

We can create a concrete implementation of this graph by creating cells for each node in the graph. Each cell contains a tag indicating what sort of node it represents, and may also contain pointers to other cells.

Normal forms

We use the following notion of a normal form:

- **Weak head normal form**: an expression \( F \, E_1 \, E_2 \ldots \, E_n \) is in WHNF iff:
  - \( n \geq 0 \)
  - \( F \) is a variable or data object or
  - \( F \) is a lambda abstraction and \( F \, E_1 \, E_2 \ldots \, E_m \) is not a redex for any \( m \leq n \)

Evaluation

- Once we have such a graph representing a program it is the job of the evaluator to reduce the graph to some normal form
- First the evaluator selects the next reducible expression (redex)
- Reduce it
- When there are no more redexes remaining the expression is in normal form
- When there is more than one redex the reduction might proceed by different routes.
- Normal order reduction:
  - Evaluate arguments on when their values are needed (not on application)
  - Only evaluate arguments once

Finding the top-level redex

For the expression \( f \, E_1 \, E_2 \ldots \, E_n \) we have the graph:

We find the next redex by unwinding the spine of the graph.
Do this by traversing the graph, building a stack

The stack top here is located at the bottom of the diagram

Reducing a Lambda expression

Substituting pointers to arguments

Reduces to:
To exploit sharing we overwrite the root of the redex with the result:

\[ \text{and} \ \
ot \hspace{0.5cm} \text{true} \]

Reduces to

\[ \text{not} \hspace{0.5cm} \text{true} \]

Construct an instance

When applying a lambda we must construct an instance of the body (make a copy):

\[ \text{not} \hspace{0.5cm} x \]

becomes

\[ \text{not} \hspace{0.5cm} \text{true} \]

With the original lambda intact (in case it is shared)
Reduce a built-in function

Can become:

Reduction summary

The node found at the tip of the spine may be:

- A data object (such as a constructor). In this case it should not be applied to any arguments
- A built-in function. If it is applied to too few arguments then it is in WHNF (so stop), otherwise evaluate the arguments, execute the built in function, and overwrite the root of the redex tree with the result
- A lambda abstraction. If there is no argument then the expression is in WHNF, otherwise instantiate the body and overwrite the root of the redex with the result.

The G Machine

- The G machine is a graph reduction implementation based on supercombinator compilation.
- A combinator is a lambda expression with no free variables
- A supercombinator is a lambda expression of the form: \( \lambda x_1. \lambda x_2 \ldots \lambda x_n. E \) where \( E \) is not a lambda abstraction and:
  - There are no free variables in the entire expression
  - Any lambda abstraction in \( E \) is itself a supercombinator
  - \( n \geq 0 \)
- We can turn any functional program into a collection of supercombinator definitions by a relatively straightforward transformation called lambda-lifting

Combinator language

After lambda lifting our program consists of a set of supercombinator definitions, each of which is a simple expression:

\[
\begin{align*}
E & \mid \text{constant} \\
& \mid \text{identifier} \\
& \mid E \ E \\
& \mid \text{let identifier} \ = \ E \ \text{in} \ E \\
& \mid \text{letrec identifier} \ = \ E \ \text{in} \ E \\
& \ldots \\
& \text{identifier} \ = \ E \\
& \text{in} \ E
\end{align*}
\]
At any given point in the execution the G-Machine has a notion of the current context.

If \( f \) is a function of two arguments we can rearrange the stack thus:

This context is maintained as:

- A function, denoted \( \rho \) which finds the offset from the base of the context to a particular argument
- \( d \), the depth of the current context (less one)

We will write the various parts of the algorithm as a series of compilation schemes which describe the conversion of parts of a supercombinator to G-Machine instructions (which themselves will be easily implemented on stock hardware). The main compilation scheme we denote \( F \):

\[
F[f \ x_1 \ x_2 \ldots \ x_n] = \text{GLOBSTART } f \ n \\
R[E][x_1 = n, x_2 = n - 1, \ldots, x_n = 1] \ n
\]

(we will use a number of primitive operations denoted by capitals, corresponding to machine instructions. GLOBSTART is the instruction that begins the program execution)
Compilation scheme R

The scheme R is responsible for

- Constructing the supercombinator instance
- Updating the root of the redex with the result
- Cleaning up the stack
- Continuing the reduction

\[
R[E] = C[E]_{\rho d}
\]

- UPDATE \(d + 1\) replaces the \(n+1\)th stack item with a pointer to the top of the stack.
- POP \(n\) removes the top \(n\) stack items
- UNWIND is a complex instruction corresponding to the outer loop of the evaluator; it either finds the next redex (when there is an application note on top of the stack), or terminates the evaluation.

Compilation scheme C

Scheme C has specialised definitions for each kind of expression.

- When the expression is a constant:

\[
C[i]_{\rho d} = \text{PUSHINT } i
\]

(and so on for other types of constant)

- When the expression is a function (either built-in or a named supercombinator):

\[
C[f]_{\rho d} = \text{PUSHGLOBAL } f
\]

- When the expression is a variable:

\[
C[x]_{\rho d} = \text{PUSH } (d - \rho x)
\]

- PUSHINT places an int in the heap (and a pointer to it on top of the stack)
- PUSHGLOBAL places a pointer to a heap node for an existing global symbol on top of the stack
- PUSH \(n\) takes a copy of the \(n\)th argument to a function on top of the stack
Compilation scheme C

When the expression is an application the situation is more complex, since we must construct the instance of each expression, and keep the pointers correct

\[ \text{C}[E_1 \ E_2] \rho \ d = \text{C}[E_1] \rho \ d \]
\[ \text{C}[E_2] \rho \ (d + 1) \]
MKAP

PKAP

MKAP creates an application node using the top two address on the stack.

Compilation Scheme C

A Let expression requires

- Construct the body of definition
- Augment \( \rho \)
- Construct the body of the let
- slide out the extra elements on the stack that points to the body of the definition.

\[ \text{C}[\text{let } x = E_x \text{ in } E_b] \rho \ d = \text{C}[E_x] \rho d \]
\[ \text{C}[E_b] \rho[x = d + 1]d + 1 \]
SLIDE1

- SLIDE \( n \) rearranges the stack by popping the \( n \) items below the top of the stack.

Compilation Scheme C

For a letrec the construction requires some auxiliary definitions:

\[
\begin{align*}
\text{C}[\text{letrec } D \text{ in } E_b] & = \text{CLetRec}[D]_{\rho'} d' \\
\text{C}[E_b] & = \text{C}[E_b]_{\rho'} d' \\
\text{SLIDE}(d' - d) \\
\text{where}(\rho', d') & = Xr[D]_{\rho d}
\end{align*}
\]

\[
\begin{align*}
\text{Xn}[x_n = E_n] & = \text{ALLOC}_n \\
\text{C}[E_n] & = \text{C}[E_n]_{\rho d}; \text{UPDATE } n \\
\text{C}[E_n + 1] & = \text{C}[E_n + 1]_{\rho d}; \text{UPDATE } n - 1 \\
\text{...} \\
\text{C}[E_1] & = \text{C}[E_1]_{\rho d}; \text{UPDATE } 1
\end{align*}
\]

\[
\begin{align*}
\text{Xn}[x_n = E_n] & = (\rho[x_n = d + n], d + n)
\end{align*}
\]