Haskell’s Type System

- Haskell’s type system is based on seminal work by (among others):
  - Haskell Curry and Robert Feys (1958)
  - Roger Hindley (1969)
  - Robin Milner (1978)
  - Luis Damas (1985)
  - Philip Wadler and Stephen Blott (1989)
  - …
Types in Haskell

Type Safety:

- If an expression $E$ has type $T$, then evaluating $E$ will produce a value of type $T$.

- “Well-typed programs do not go wrong” (Robin Milner)

- No need to check types of values at run-time (a performance benefit)
Flexibility:

- Polymorphism allows the definition of functions that work uniformly over many different types of value.

- Higher-order functions make it possible to capture common patterns of computation and/or custom control structures.
Type Inference:

- There is an algorithm that can be used to determine if a term/program is well-typed.
- Any well-typed expression has a most general (principal) type from which all other possible types can be obtained.
- Explicit types can be provided as useful documentation, but are (usually) not required.
... continued

 Ease of Implementation:

- Type checking algorithm is relatively straightforward to implement

- Polymorphic functions are relatively easy to implement

Time to look at some details ...
Type Inference and Polymorphism
Type Inference

How do you figure out the type of an expression?

Known functions and constants have known types:

- True, False :: Bool
- not :: Bool -> Bool
- (&&) :: Bool -> Bool -> Bool
- ...

Applications are type checked using the rule:
- If T and S are types,
- \( e_1 \) is an expression of type \( T \rightarrow S \),
- \( e_2 \) is an expression of type \( T \),
- Then \( e_1 \ e_2 \) is an expression of type \( S \)
What about function definitions or lambda expressions?

Example: What is the type of the following function?

\[ \text{subst } x \ y \ z = x \ z \ (y \ z) \]

And how would we expect GHC to figure it out?

Inspiration: In math, we use variables as placeholders for unknown values ...

- Example: \[ 6x + 8y = 48 \]
Typing subst

\[
\text{subst } x \ y \ z = x \ z \ (y \ z)
\]

In the same way, we can use \textit{type variables} as placeholders for unknown types ...

To start, pick three “fresh” type variables to represent the type of values in the three parameters

- \( x :: a \)
- \( y :: b \)
- \( z :: c \)

If there is any relationship between \( a, b, \) and \( c, \) we’ll discover that as we proceed.
... continued

Consider the expression $x \, z \, (y \, z)$:

- Because $y$ is applied to $z$, we can infer that $b$ must be a function type $b = c \rightarrow d$ for some type $d$.
- Similarly, $x$ is applied to $z$, so: $a = c \rightarrow e$ for some type $e$.
- Finally, $(x \, z)$ is applied to $(y \, z)$, so: $e = d \rightarrow f$ for some type $f$.

Thus $x \, z \, (y \, z) :: f$ where:

- $x :: a, y :: b, z :: c$
- $a = c \rightarrow e$
- $b = c \rightarrow d$
- $e = d \rightarrow f$
- $x :: c \rightarrow d \rightarrow f$
- $y :: c \rightarrow d$
- $z :: c$
If we can show $e :: t$ when we assume that $x :: s$, then the function $\lambda x \to e$ has type $s \to t$.

For our example:
- Assuming $x :: c \to d \to f$, $y :: c \to d$, and $z :: c$ ...
- ... we have shown that $x \ z \ (y \ z) :: f$

Hence:

$$(\lambda x \ y \ z \to x \ z \ (y \ z))$$

$\quad :: (c \to d \to f) \to (c \to d) \to c \to f$

Or, equivalently:

$\text{subst} :: (c \to d \to f) \to (c \to d) \to c \to f$
Generalization

- We made all this progress without assuming anything about types c, d, and f.

- So, if we picked any types X, Y, and Z, then subst could also be used as a value of type
  \[(X \to Y \to Z) \to (X \to Y) \to X \to Z\]

- In fact, for all choices of a, b, and c, we could use subst as a value of type
  \[(a \to b \to c) \to (a \to b) \to a \to c\]

- We’ve just made the argument that:
  \[\text{subst} \:: \forall a. \forall b. \forall c. (a \to b \to c) \to (a \to b) \to a \to c\]
Type Variables

A type variable begins with a lower case letter and represents an arbitrary type.

A type expression that doesn’t include variables is sometimes called a monotype.

A type expression that includes type variables is sometimes called a type scheme because it represents a family of types.

E.g., (a -> a) represents a set of types that includes (Int->Int), (Bool->Bool), ([Int]->[Int]) and ((Int -> Bool) -> (Int -> Bool)) ... but not Int -> Bool.
Quantifier Notation

- We sometimes write type schemes using “forall” quantifiers: \( \forall a. \ a \rightarrow \ a \)
  - We can write this in actual code as \( \texttt{forall } a . \ a \rightarrow \ a \) if we use the ScopedTypedVariables extension in GHC

- This emphasizes the fact that this type works “for all” choices of the type \( a \).

- It is possible to use multiple quantifiers:
  \( \forall a. \ \forall b. \ a \rightarrow b \rightarrow a \)

- If \( e :: \forall a. \ T(a) \), then we can instantiate the quantified variable \( a \) with any other type \( t \), and use \( e \) as a value of type \( T(t) \)
Examples

Example: we can instantiate \( \text{id} :: \forall a. \ a \to a \) to obtain:
- \( \text{id} :: \text{Bool} \to \text{Bool} \)
- \( \text{id} :: \text{Char} \to \text{Char} \)
- \( \text{id} :: (a,b) \to (a,b) \)
- ...

Example: we can instantiate \( \text{subst} :: \forall a. \forall b. \forall c. (a \to b \to c) \to (a \to b) \to a \to c \) to obtain:
- \( \text{subst} (\&\&) \text{not} \text{True} :: \text{Bool} \)
- \( \text{subst} (+) (2*) 3 :: \text{Int} \)
- \( \text{subst} (:) (\lambda x \to [x,x]) \text{id} :: ? \)
- \( \text{subst} \text{map} (\lambda f \to f . f) \text{True} :: ? \)
Aside: Types are Logical

- **Typing Function Application**
  \[
  \begin{align*}
  f &:: A \to B \\
  x &:: A \\
  f \ x &:: B
  \end{align*}
  \]

- **Typing Lambda Expressions**
  Assuming \(x :: A\) \(e :: B\)
  \[
  (\lambda x \to e) :: A \to B
  \]
Aside: Types are Logical

- Typing Function Application

\[
\begin{align*}
f &:: A -> B \\
x &:: A \\
f \ x &:: B
\end{align*}
\]

- Typing Lambda Expressions

\[
\begin{align*}
\text{Assuming } x &:: A \quad e &:: B \\
(\lambda x -> e) &:: A -> B
\end{align*}
\]
Aside: Types are Logical

- **Typing Function Application**
  
  \[ f :: A \to B \quad x :: A \]
  
  \[ f \ x :: B \]

- **Typing Lambda Expressions**
  
  **Deduction Theorem**
  
  Assuming \( x :: A \quad e :: B \)
  
  \[ (\lambda x \to e) :: A \to B \]
Aside: Types are Logical

Hypothetical Syllogism:
if A -> B and B -> C, then A -> C

Proof: Let g :: A -> B and f :: B -> C

- Assume x :: A
- Apply g: g x :: B
- Apply f: f (g x) :: C
- Discharge assumption: \( \lambda x \to f (g x) :: A \to C \)

Composition \( \lambda f g x \to f (g x) :: (B \to C) \to (A \to B) \to (A \to C) \)
Type Annotations

- Haskell allows us to add type signatures to function definitions
  
  ```haskell
  id :: a -> a
  id x = x
  ```

- Type variables on the right of a `::` are assumed to be implicitly bound by a `∀`

- Haskell also allows type annotations on expressions:
  
  ```haskell
  (\x -> x) :: a -> a
  ```

- And on variables bound in patterns
  
  ```haskell
  (\(x::Int) -> x+1) :: Int -> Int
  ```
  
  but only if ScopedTypedVariables extension is enabled
It’s ok to declare any type that is an instance of the principal type:

\[ \text{id} :: a \to a \]
\[ \text{id} :: b \to b \]
\[ \text{id} :: (a,b) \to (a,b) \]
\[ \text{id} :: \text{Int} \to \text{Int} \]
\[ \text{id} :: (\text{Int}, [b\to\text{Int}]) \to (\text{Int}, [b \to \text{Int}]) \]
\[ \text{id} :: (a \to a) \to (a \to a) \]

Uses of the function will be restricted to the declared type.
... continued

It is an error to declare a type that is not an instance of the principal type:

- `id :: Int -> Bool`
- `id :: Bool -> [Bool]`
- `id :: a -> b`

None of these types will be accepted

None of these types is consistent with the behavior of the `id` function
... continued

- It is often useful to write types in code as a form of documentation

- But the types can be inferred automatically if they are omitted

- The Haskell typechecker will always choose the most general type possible
Type Errors

Type errors occur when the constraints that we obtain cannot be solved:

- **if** True **then** False **else** 'a'
  - **a** does not match **Char**

- \( \forall x \rightarrow x \times x \)
  - “Occurs check: cannot construct the infinite type: a ~ a \rightarrow b”
  - if x :: a, then a = a \rightarrow b, for some b
  - Hence a = (a \rightarrow b) \rightarrow b = ((a \rightarrow b) \rightarrow b) \rightarrow b = (((a \rightarrow b) \rightarrow b) \rightarrow b) \rightarrow b = ...
“Let Polymorphism”

- Haskell will infer polymorphic types for functions defined at the top-level

- and also in local definitions (i.e., in a let or where clause)

- Example: What is the type of this function?
  
  \[
  f \ x \ y = \text{let} \ mi \ z = z \ \text{in} \ (mi \ x, \ mi \ y)
  \]
“Lambda-bound Variables”

A limitation of the Haskell type system:
- Polymorphic values cannot be passed as function arguments

Example:
- (id 'a', id True) :: (Char, Bool)
- But \( \text{id} \rightarrow (\text{id} \ 'a', \text{id} \ True) \) is not well-typed
Subtleties (1)

Consider the following definition:

\[
f(x) = \text{let } g(y) = [x, y] \text{ in } g(x)
\]

What is the type of \( f \)?

What is the type of \( g \)?
Subtleties (2)

Suppose that we define:

\[
\text{box} :: \text{a} \rightarrow \text{[a]}
\]
\[
\text{box } x = \text{[x]}
\]

What is the type of:

\[
\text{box } (\text{box } \text{True})?
\]

What is the type of:

\[
(\lambda b \rightarrow b \ (\text{bTrue})) \ \text{box}?
\]
Subtleties (3)

- Haskell will not accept the following function definition:
  \[ f \, xs = \text{null} \, xs \; || \; f \, [xs] \]

- But it will accept the definition if we add a type signature:
  \[ f :: [a] \rightarrow \text{Bool} \]

- What’s going on here?
- (“polymorphic recursion”!)
Consider the following example:

\[ h = f4 \text{id} \]

where

\[ \text{pair } x \; y \; f = f \times y \]

\[ f1 \; y = \text{pair } y \; y \]

\[ f2 \; y = f1 \; (f1 \; y) \]

\[ f3 \; y = f2 \; (f2 \; y) \]

\[ f4 \; y = f3 \; (f3 \; y) \]

What is the type of \( h \)?

What happens if we extend the pattern to \( f5 \)?
Summary

The Haskell/Hindley-Milner type system hits a sweet spot providing safety, flexibility, type inference and ease of implementation.

Every well-typed term has a most general type that can be inferred automatically.

There are some subtleties and pathological bad behavior ... but, overall:

- The type system works well in practice
- It is fairly intuitive and flexible
- It is hard to live without when you go back to C/Java/C#/PHP/...