CS 457/557: Functional Languages

Week 6: Haskell Type Checking

Mark P Jones and Andrew Tolmach
Portland State University

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)

… continued

- 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

… continued

- 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 -> S,
  - e_2 is an expression of type T,
  - Then e_1 e_2 is an expression of type S

… continued

- What about function definitions or lambda expressions?
- Example: What is the type of the following function? 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

subst x y z = x z (y z)

- In the same way, we can use 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

subst x y z = x z (y z)

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 -> d for some type d
- Similarly, x is applied to z, so:
  a = c -> e for some type e
- Finally, (x z) is applied to (y z), so:
  e = d -> f for some type f

Thus x z (y z) :: f where:

- x :: a, y :: b, z :: c
- a = c -> e
- b = c -> d
- e = d -> f

- x :: c -> d -> f
- y :: c -> d
- z :: c

… continued

subst x y z = x z (y z)

- If we can show e :: t when we assume that x :: s, then the function \x -> e has type s -> t

- For our example:
  - Assuming x :: c -> d -> f, y :: c -> d, and z :: c …
  - … we have shown that x z (y z) :: f

- Hence:

```
(\x y z -> x z (y z))
        :: (c -> d -> f) -> (c -> d) -> c -> f
```

Or, equivalently:

```
subst :: (c -> d -> f) -> (c -> d) -> c -> 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 -> Y -> Z) -> (X -> Y) -> X -> Z
- In fact, **for all** choices of a, b, and c, we could use subst as a value of type

  (a -> b -> c) -> (a -> b) -> a -> c
- We've just made the argument that:

  subst :: ∀a. ∀b. ∀c.
  (a -> b -> c) -> (a -> b) -> a -> c

Type Variables

- A <u>type variable</u> begins with a lower case letter and represents an arbitrary type
- A type expression that doesn’t include variables is sometimes called a <u>monotype</u>
- A type expression that includes type variables is sometimes called a <u>type scheme</u> 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 forall a . a –> 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 id :: ∀a. a -> a to obtain:
  - id :: Bool -> Bool
  - id :: Char -> Char
  - id :: (a,b) -> (a,b)
  - …

- Example: we can instantiate
  subst :: ∀a. ∀b. ∀c. (a -> b -> c) -> (a -> b) -> a -> c
  to obtain:
  - subst (&&) not True :: Bool
  - subst (+) (2*) 3 :: Int
  - subst (:) (\x -> [x,x]) id :: ?
  - subst map (\f -> f . f) True :: ?

Aside: Types are Logical

- Typing Function Application

$$\frac{f :: A \to B \qquad x :: A}{f\ x :: B}$$

- Typing Lambda Expressions

$$\frac{\text{Assuming } x :: A \qquad e :: B}{(\backslash x \to e) :: A \to B}$$

Aside: Types are Logical

- Typing Function Application

$$\frac{f :: A \text{ -> } B \qquad x :: A}{f\ x :: B}$$

- Typing Lambda Expressions

$$\frac{\text{Assuming } x :: A \qquad e :: B}{(\backslash x \text{ -> } e) :: A \text{ -> } B}$$

Aside: Types are Logical

- ~~Typing Function Application~~ Modus Ponens

$$\frac{f :: A \to B \qquad x :: A}{f\ x :: B}$$

- ~~Typing Lambda Expressions~~ Deduction Theorem

$$\frac{\text{Assuming } x :: A \qquad e :: B}{(\backslash x \to e) :: A \to B}$$

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:	\x -> f (g x) :: A -> C

Composition \f g x -> f (g x)

:: (B -> C) -> (A -> B) -> (A -> C)

Type Annotations

- Haskell allows us to add type signatures to function definitions

```
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:

```
(\x -> x) :: a -> a
```

- And on variables bound in patterns

```
(\(x::Int) -> x+1) :: Int -> Int
```

  but only if ScopedTypedVariables extension is enabled

… continued

- It's ok to declare any type that is an instance of the principal type:

```
id :: a -> a
id :: b -> b
id :: (a,b) -> (a,b)
id :: Int -> Int
id :: (Int, [b->Int]) -> (Int, [b -> Int])
id :: (a -> a) -> (a -> 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'
  - Bool does not match Char
- \x -> x x
  - "Occurs check: cannot construct the infinite type: a ~ a -> b"
  - if x :: a, then a = a -> b, for some b
  - Hence a = (a -> b) -> b = ((a -> b) -> b) -> b = (((a -> b) -> b) -> b) -> 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 = **let** mi z = z 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 \id -> (id 'a', id True) is not well-typed

Subtleties (1)

- Consider the following definition:

```
f x = let g y = [x, y]
       in  g x
```

- What is the type of f?
- What is the type of g?

Subtleties (2)

- Suppose that we define:

```
box   :: a -> [a]
box x  = [x]
```

- What is the type of:

  box (box True)?

- What is the type of:

  (\b -> b (bTrue)) box?

Subtleties (3)

- Haskell will not accept the following function definition:

  ```
  f xs  =  null xs  ||  f [xs]
  ```

- But it will accept the definition if we add a type signature:

  ```
  f :: [a] -> Bool
  ```

- What’s going on here?
- (“polymorphic recursion”!)

Pathologies

- Consider the following example:

```
h = f4 id
  where
    pair x y f = f x y
    f1 y = 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/…