Systematic Rules for Figuring Out Types

Prof. Susan Older

18 November 2019

Rule #1: Function Application

Suppose $\texttt{exp}_1$ and $\texttt{exp}_2$ are expressions with types as follows:

$$\begin{aligned} \texttt{exp}_1 &:: t \texttt{ -> } t' \\ \texttt{exp}_2 &:: t \end{aligned}$$

Then the expression $\texttt{exp}_1\ \texttt{exp}_2$ has type t'.

```
             h  :: Int -> Char
(length "abc")  :: Int
       isLower  :: Char -> Bool
             g  :: Char -> (Bool -> String)
```

```
          h (length "abc")  :: Char
isLower (h (length "abc"))  :: Bool
                  g 'A'     :: Bool -> String
       g 'A' (isLower '!')  :: String
```


Rule # 1b: Partial Application

Suppose we have the following:

$$\begin{aligned} \texttt{f} &:: \texttt{t}_1 \texttt{ -> } \texttt{t}_2 \texttt{ -> } \cdots \texttt{ -> } \texttt{t}_n \texttt{ -> t} \\ \texttt{e}_1 &:: \texttt{t}_1 \\ \texttt{e}_2 &:: \texttt{t}_2 \\ &\vdots \\ \texttt{e}_k &:: \texttt{t}_k \quad \text{(where } k < n\text{)} \end{aligned}$$

Then:

$$\texttt{f e}_1\ \texttt{e}_2 \cdots \texttt{e}_k :: \texttt{t}_{k+1} \texttt{ -> } \texttt{t}_{k+2} \texttt{ -> } \cdots \texttt{ -> } \texttt{t}_n \texttt{ -> t}$$

The textbook calls this the Rule of Cancellation (see page 244).

Rule #1b: Partial Application (A Generic Example)

Suppose $\texttt{fun}$, $\texttt{exp}_1$, $\texttt{exp}_2$, ..., $\texttt{exp}_4$ are expressions with types as follows:

$$\begin{aligned} \texttt{fun} &:: t_1 \texttt{ -> } t_2 \texttt{ -> } t_3 \texttt{ -> } t_4 \texttt{ -> } t \\ \texttt{exp}_1 &:: t_1 \\ \texttt{exp}_2 &:: t_2 \\ \texttt{exp}_3 &:: t_3 \\ \texttt{exp}_4 &:: t_4 \end{aligned}$$

Then:

$$\begin{aligned} \texttt{fun exp}_1 &:: t_2 \texttt{ -> } t_3 \texttt{ -> } t_4 \texttt{ -> } t \\ \texttt{fun exp}_1\ \texttt{exp}_2 &:: t_3 \texttt{ -> } t_4 \texttt{ -> } t \\ \texttt{fun exp}_1\ \texttt{exp}_2\ \texttt{exp}_3 &:: t_4 \texttt{ -> } t \\ \texttt{fun exp}_1\ \texttt{exp}_2\ \texttt{exp}_3\ \texttt{exp}_4 &:: t \end{aligned}$$

Suppose `g` has type `Int -> Char -> Bool -> Float -> String`.

What is the type of `g (length "cis 252") '#'`?

Rule #2: Tuples

Tuples provide a way to package together a fixed number of items. (The individual components of a tuple may have different types.)

Suppose $\texttt{exp}_1$, $\texttt{exp}_2$, ..., $\texttt{exp}_n$ are expressions with types as follows:

$$\begin{array}{lcl} \texttt{exp}_1 & :: & t_1 \\ \texttt{exp}_2 & :: & t_2 \\ \vdots & & \\ \texttt{exp}_n & :: & t_n \end{array}$$

Then $(\texttt{exp}_1, \texttt{exp}_2, \cdots, \texttt{exp}_n)$ has type $(t_1, t_2, \cdots, t_n)$.

```
          'A'  ::  Char
length "abc"   ::  Int
 isLower '!'   ::  Bool
```

Thus:
`('A', length "abc", isLower '!')`
has type `(Char,Int,Bool)`.

Rule #3: Lists

Lists provide a way to package together an arbitrary number of items, each of which has the same type.

Suppose $\texttt{exp}_1$, $\texttt{exp}_2$, ..., $\texttt{exp}_n$ are expressions with types as follows:

$$\begin{array}{lcl} \texttt{exp}_1 & :: & t \\ \texttt{exp}_2 & :: & t \\ \vdots & & \\ \texttt{exp}_n & :: & t \end{array}$$

Then the expression $[\texttt{exp}_1, \texttt{exp}_2, \cdots, \texttt{exp}_n]$ has type `[t]`.

```
           True  ::  Bool
length "abc" > 5 ::  Bool
     isLower '!' ::  Bool
```

Thus:
`[True, length "abc" > 5, isLower '!']`
has type `[Bool]`.

Rule #4: Polymorphism

Suppose `exp` is an expression whose type contains type variables (for example: `a`, `b` and `c`).

- You can plug any type t_1 in for all of the `a`s.
- You can plug any type t_2 in for all of the `b`s.
- You can plug any type t_3 in for all of the `c`s.
- `exp` will have the resulting type.

Suppose `blah` has type `a -> b -> (a,b)`:

```
blah  ::  Int -> Char -> (Int,Char)
blah  ::  Bool -> Float -> (Bool,Float)
blah  ::  [(Char,Int)] -> b -> ([(Char,Int)],b)
blah  ::  Int -> Int -> (Int,Int)
```


Polymorphism: More Examples

Recall these types:

```
fst  ::  (a,b) -> a      (:)  ::  a -> [a] -> [a]
```

What are the types of these expressions?

1. `fst ('A', False)` has type `Char`, because:

```
          'A'  ::  Char
        False  ::  Bool
('A', False)   ::  (Char,Bool)
          fst  ::  (Char,Bool) -> Char
```

2. `"hello!":[]` has type `[String]`, because:

```
"hello!"  ::  String
      []  ::  [a]
      []  ::  [String]
     (:)  ::  String -> [String] -> [String]
```