A Quick Introduction to Polymorphic Type Checking

J.N. Oliveira

U.Minho - March 2011

Polymorphic types

A type is said to be polymorphic if (a) it defines data structures holding values of other types (eg. lists of Booleans, trees of integers); (b) it encompasses operations which work independently of which particular values are held in the structure (eg. appending two lists, computing the depth of a tree).

Polymorphic functions are therefore *generic* in the sense that they are defined *once* for all its possible applications and instantiations. This is of great conceptual economy and saves a lot of programming effort. Moreover, every polymorphic function enjoys a *natural* or *free* property [2] which exhibits its type and is of great help in calculating programs.

However, we need rules enabling the inference of the most general (polymorphic) type of a given functional expression. The following rules apply to the pointfree combinators used in the algebra of programming.

Typing Rules

Each rule is of the form

$$\frac{a \ , \ b}{a \mathbin{\phi} b \quad \{e\}}$$

where a and b are polymorphic functional expressions, ϕ is a functional combinator and e is a set of type equalities required for expression $a \mathbin{\phi} b$ to be well-typed. Examples of typing rules follow:

– Composition:

$$\frac{B \xleftarrow{\ f\ } A \ \ , \ \ D \xleftarrow{\ g\ } C}{B \xleftarrow{\ f \cdot g\ } C \qquad \{A = D\}}$$

– Split:

$$\frac{B \xleftarrow{\ f\ } A \ \ , \ \ D \xleftarrow{\ g\ } C}{B \times D \xleftarrow{\ \langle f, g \rangle\ } C \qquad \{A = C\}}$$

– Product:

$$\frac{B \xleftarrow{\ f\ } A \ \ , \ \ D \xleftarrow{\ g\ } C}{B \times D \xleftarrow{\ f \times g\ } A \times C \qquad \{\}}$$

Exercise: add the typing rules of the other combinators not in the list.
□

Further to the above, the *equality rule* is implicit in typed functional equality:

$$\frac{B \xleftarrow{f} A \;=\; D \xleftarrow{g} C}{\{B = D, A = C\}}$$

Type checking

Type polymorphism raises the following question: what is the *most general* type which accommodates a given function or functional expression? Such a type (if it exists) is known as the expression's *principal type*, from which all other valid types is obtained by instantiation.

The more polymorphic the type of a function, the more applicable the function is. Thus the interest of the following algorithm.

Damas-Milner's algorithm

(Adapted from [1])

> (a) Start by typing all functions so that no type variable is shared by two different functions. (b) Apply typing rules as much as needed; (c) Collect all type unification equations and solve them.

If no finite solution can be found for the obtained system of type equations, the function will be ill-typed and cannot be trusted. (In Haskell, it won't compile.)

First example

Typing $\pi_1 \cdot \pi_1$:

$$\frac{A \xleftarrow{\pi_1} A \times B \;\;,\;\; C \xleftarrow{\pi_1} C \times D}{\{C = A \times B\} \qquad A \xleftarrow{\pi_1 \cdot \pi_1} C \times D}$$

Only the composition rule was applied, thus a single type unification and the final polymorphic type, obtained by substitution $c := A \times B$:

$$A \xleftarrow{\pi_1 \cdot \pi_1} (A \times B) \times D$$

Second example

Next, we want to type $f = \langle \pi_1 \cdot \pi_1, \pi_2 \times id \rangle$. As we already have the type of $\pi_1 \cdot \pi_1$, we focus on inferring the type of $\pi_2 \times id$,

$$\frac{F \xleftarrow{\pi_2} E \times F \;\;,\;\; G \xleftarrow{id} G}{F \times G \xleftarrow{\pi_2 \times id} (E \times F) \times G \qquad\qquad \{\}}$$

which raises the empty set of type constraints. Then we put both together:

$$\frac{A \xleftarrow{\pi_1 \cdot \pi_1} (A \times B) \times D \;\;,\;\; \dfrac{F \xleftarrow{\pi_2} E \times F \;\;,\;\; G \xleftarrow{id} G}{F \times G \xleftarrow{\pi_2 \times id} (E \times F) \times G \qquad \{\}}}{A \times (F \times G) \xleftarrow{f} (A \times B) \times D \qquad\qquad \{(A \times B) \times D = (E \times F) \times G\}}$$

We finish by solving the type unification equation:

$$(A \times B) \times D = (E \times F) \times G \quad \equiv \quad A = E, B = F, D = G$$

The final, polymorphic type of $f = \langle \pi_1 \cdot \pi_1, \pi_2 \times id \rangle$ is, therefore:

$$A \times (B \times D) \xleftarrow{f} (A \times B) \times D$$

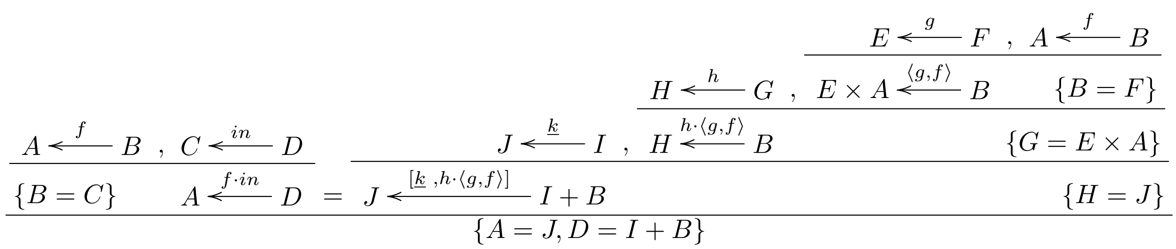
This example shows that one can proceed in a stepwise manner by inferring the types of sub-expressions separately and then merging the constraints.

Third example

What are the most general polymorphic types for the functions in function equality

$$f \cdot in = [\underline{k}\ , h \cdot \langle g, f \rangle] \quad ?$$

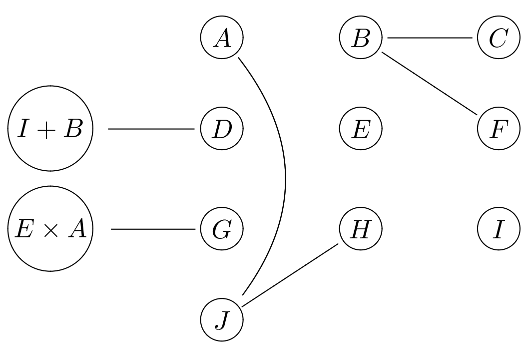
The whole type inference process is given below:



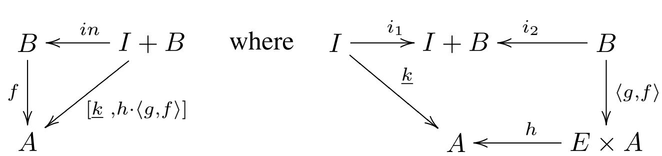
Collecting all type equations:

$$\begin{aligned} &A = J \\ &B = C = F \\ &D = I + B \\ &G = E \times A \\ &H = I \end{aligned}$$

Type unifications graphically:



Final type scheme:



Third example

Suppose we wish to define a new combinator of the algebra of programming as follows:

$$new(f, g) = \langle f, [g \ , f] \rangle$$

However, when submitting this definition to GHCi, we get an error message:

```
<interactive>:1:28:
    Occurs check: cannot construct the infinite type: b = Either a b
      Expected type: b -> c
      Inferred type: Either a b -> b1
    In the second argument of `either', namely `f'
    In the second argument of `split', namely `(either g f)'
*Cp>
```

Why? Just apply the typing rules of *split* and *either* so as to explain the type error message. You will infer type equation $B = A + B$ from such rules, which indeed has no finite (polymorphic) solution: $B = A + A + ...$ will, in general, be an infinite type!

References

1. Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.
2. P.L. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359, London, Sep. 1989. ACM.