Type Inference

Some statically typed languages, like ML (and to a lesser extent Scala), offer alternative ways to regain the flexibility of dynamic typing, via type inference and polymorphism.

Type inference works like this:

- The types of identifiers are automatically inferred from the way they are used.
- The programmer is no longer required to declare the types of identifiers (although this is still permitted).
- Method requires that the types of operators and literals is known.
Inference Examples

(let f (fun (x) (+ x 2))
  (@ f y))

The type of \(x\) must be \texttt{int} because it is used as an arg to +. So the type of \(f\) must be \texttt{int \rightarrow int}\ (i.e. the type of functions that expect an \texttt{int} argument and return an \texttt{int} result), and \(y\) must be an \texttt{int}.

(let f (fun (x) (cons x nil))
  (@ f true))

Suppose \(x\) has some type \(t\). Then the type of \(f\) must be \(t \rightarrow \text{(list t)}\). Since \(f\) is applied to a \texttt{bool}, we must have \(t = \text{bool}\).

For the moment, we assume that \(f\) must be given a unique \textbf{monomorphic} type; we will relax this later…
Here’s a harder example:

\[\text{(let } f \text{ (fun } x \text{ (if } x \text{ p q)} \text{)} \\
\quad \text{(+ 1 (@ f r))}\]

Can only infer types by looking at both the function’s body and its application.

In general, we can solve the inference task by extracting a collection of typing constraints from the program’s AST, and then finding a simultaneous solution for the constraints using unification.

Extracted constraints tell us how types must be related if we are to be able to find a typing derivation. Each node generates one or more constraints.
Rules for First-class Functions

To handle this example, we’ll need some extra typing rules:

\[
\frac{TE + \{x \mapsto t_1\} \vdash e : t_2}{TE \vdash (\text{fun } (x) \ e) : t_1 \rightarrow t_2} \quad \text{(Fn)}
\]

\[
\frac{TE \vdash e_1 : t_1 \rightarrow t_2 \quad TE \vdash e_2 : t_1}{TE \vdash (@ e_1 \ e_2) : t_2} \quad \text{(Appl)}
\]
Inference Example

Solution: \( t_1 = t_7 = t_8 = t_9 = t_3 = t_5 = t_p = t_6 = t_q = \text{int} \)
\( t_4 = t_x = t_{11} = t_r = \text{bool} \)
\( t_2 = t_f = t_{10} = \text{bool} \rightarrow \text{int} \)

<table>
<thead>
<tr>
<th>Node</th>
<th>Rule</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Let</td>
<td>( t_f = t_2 ) ( t_1 = t_7 )</td>
</tr>
<tr>
<td>2</td>
<td>Fun</td>
<td>( t_2 = t_x \rightarrow t_3 )</td>
</tr>
<tr>
<td>3</td>
<td>If</td>
<td>( t_4 = \text{bool} ) ( t_3 = t_5 = t_6 )</td>
</tr>
<tr>
<td>4</td>
<td>Var</td>
<td>( t_4 = t_x )</td>
</tr>
<tr>
<td>5</td>
<td>Var</td>
<td>( t_5 = t_p )</td>
</tr>
<tr>
<td>6</td>
<td>Var</td>
<td>( t_5 = t_q )</td>
</tr>
<tr>
<td>7</td>
<td>Add</td>
<td>( t_7 = t_8 = t_9 = \text{int} )</td>
</tr>
<tr>
<td>8</td>
<td>Int</td>
<td>( t_8 = \text{int} )</td>
</tr>
<tr>
<td>9</td>
<td>Appl</td>
<td>( t_{10} = t_{11} \rightarrow t_9 )</td>
</tr>
<tr>
<td>10</td>
<td>Var</td>
<td>( t_{10} = t_f )</td>
</tr>
<tr>
<td>11</td>
<td>Var</td>
<td>( t_{11} = t_r )</td>
</tr>
</tbody>
</table>
Drawbacks of Inference

Consider this variant example:

\[
\text{let } f = \text{fun (x) (if x p false)}
\text{ (+ 1 (@ f r))}
\]

Now the body of \( f \) returns type \texttt{bool}, but it is used in a context expecting an \texttt{int}.

The corresponding extracted constraints will be inconsistent; no solution can be found. Can report a type error to the programmer.

But which is wrong, the definition of \( f \) or the use? No good way to associate the error message with a single program point.
Consider

```
(let snd (fun (l) (head (tail l)))
  (@ snd (cons 1 (cons 2 (cons 3 nil)))))
```

By extracting constraints and solving, we will get

```
snd : (list int) → int
```

Same definition!

We could also write

```
(let snd (fun (l) (head (tail l)))
  (@ snd (cons true (cons false (cons true nil)))))
```

And get

```
snd : (list bool) → bool
```
Polymorphism (2)

- So why can’t we write something like this?

(let snd (fun (l) (head (tail l)))
  (block
    (@ snd (cons 1 (cons 2 (cons 3 nil))))
    (@ snd (cons true (cons false (cons true nil))))))

- We can, by treating the type of snd as **polymorphic**

  snd : (list t) → t

- Here t is an unconstrained **type variable**
Inferring Polymorphism

In fact, if we extract constraints and solve just for the definition \((\text{fun}\ l\ (\text{head}\ (\text{tail}\ l)))\) without considering its uses, we will end up with exactly the type \((\text{list}\ t) \rightarrow t\)

We can assign this *polymorphic* type to \(\text{snd}\) allowing it to be used multiple times, each with a different instance of \(t\) (e.g. with \(t = \text{bool}\) or \(t = \text{int}\)).

By default, languages like ML infer the most polymorphic possible type for every function

This is the natural result of the inference process we’ve described
Parametric Polymorphism

We can think of these polymorphic types as being universally quantified over their type variables and instantiated at use sites.

\[
\text{let } \text{snd} : \forall t. \text{list } t \rightarrow t = \ldots \\
\text{in } \text{snd} \ \{\text{bool}\} \ (\text{true}::\text{false}::\text{nil}); \\
\text{snd} \ \{\text{int}\} \ (1::2::\text{nil})
\]

This is called parametric polymorphism because the function definition is (implicitly) parameterized by the instantiating type.

In ML-like languages the quantification and instantiation don’t actually appear.
Explicit Parametric Polymorphism

Java generics and Scala type parameterization are also a form of a parametric polymorphism, in which type abstraction and instantiation are (mostly) explicit.

```scala
def snd[A](l: List[A]) : A = l.tail.head
val a = snd[Boolean] (List(true,false))
val b = snd(List(1,2))
```

In parametric polymorphism, the behavior of the polymorphic function is the same no matter what the instantiating type is.

In fact, an ML compiler typically generates just one piece of machine code for each polymorphic function, shared by all instances.
Overloading and Ad-hoc Polymorphism

- Most languages provide some form of overloading, where the same function name or operator symbol means different things depending on the types to which it is applied.
  - e.g. overloading of arithmetic operators to work on either integers or floats is very common.

- Some languages (e.g. Ada, C++) support user-defined overloading, especially useful for user-defined types (e.g. complex numbers).

- OO languages (e.g. C++, Java) often support method overloading based on argument types.

- Overloading is sometimes called ad-hoc polymorphism, because the implementation of the overloaded operator changes based on the argument types.
Static vs. Dynamic Overloading

- In most statically-typed languages, overloading is resolved \textit{statically}; i.e. the compiler selects the right version of the overloaded definition once and for all at compile time.

- Dynamically-typed languages also often overload operators (e.g. + on different kinds of numbers, strings, etc.)

  - Here the right version of the overloaded operator is picked at \textit{runtime} after checking the (runtime) types of the arguments.

  - Of course, the operator might fail altogether if there is no version suitable for the types discovered.

- Haskell \textit{type classes} provide an unusual form of dynamic overloading with a static guarantee that a suitable version exists.