CS558 Programming Languages
Winter 2010
Lecture 14
Static typechecking offers the great advantage of catching errors early, and generally supports more efficient execution.

Why ever consider dynamic typechecking?

• Simplicity. For short or simple programs, it’s nice to avoid the need for declaring the types of identifiers.

• Flexibility. Static typechecking is inherently more conservative about what programs it admits. For example, suppose function \( f \) happens to always return \( \text{false} \). Then

\[
(+ (\text{if } (f 0) "a" 2) 2)
\]

will never cause a runtime type error, but it will still be rejected by a static type system.

Perhaps more usefully, dynamic typing allows container data structures, to contain mixtures of values of arbitrary types, like this list:

\[
(\text{list } 2 \ #t \ "abc")
\]
Some statically-typed languages, including ML and Haskell, offer alternative ways to approach these goals, via type inference and polymorphic typing.

**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).
- Requires that the types of operators and literals are known.
Pretend Scheme is statically typed with base type `int` and `bool` (with values `#t` and `#f`).

```scheme
(let ((f (lambda (x) (+ x 2))))
  (f y))
```

The type of `x` must be `int` because it is used as an arg to `+`. So the type of `f` must be `int -> int`, and `y` must be an `int`.

```scheme
(let ((f (lambda (x) (list x))))
  (f #t))
```

Suppose `x` has some type `t`. Then the type of `f` must be `t -> t list`. Since `f` is applied to a `bool`, we must have `t = bool`.

(For the moment, we’re assuming the `f` must be given a unique **monomorphic** type; we’ll improve on this later.)
A harder example:

```scheme
(let ((f (lambda (x) (if x p q))))
 (+ 1 (f r)))
```

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

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.

Extract constraints that 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.
We’ll need some extra inference rules:

\[
\begin{align*}
    TE + \{x \mapsto t_1\} & \vdash e : t_2 & \text{(Fn)} \\
    TE & \vdash (\lambda (x) \ e) : t_1 \rightarrow t_2
\end{align*}
\]

\[
\begin{align*}
    TE & \vdash e_1 : t_1 \rightarrow t_2 \quad TE & \vdash e_2 : t_1 \\
    TE & \vdash (e_1 e_2) : t_2 & \text{(Appl)}
\end{align*}
\]
Inference Example

\[ \text{LET}(f) \]

- \( fn(x) \)
- \( + \)
- \( \text{IF} \)
- \( x \)
- \( p \)
- \( q \)
- \( f \)
- \( r \)
# Solving Inference Constraints

<table>
<thead>
<tr>
<th>Node</th>
<th>Rule</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Let</td>
<td>$t_f = t_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$t_1 = t_7$</td>
</tr>
<tr>
<td>2</td>
<td>Fn</td>
<td>$t_2 = t_x \rightarrow t_3$</td>
</tr>
<tr>
<td>3</td>
<td>If</td>
<td>$t_4 = \text{bool}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$t_3 = t_5 = t_6$</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>

Solution: $t_1 = t_7 = t_8 = t_9 = t_3 = t_5 = t_p = t_6 = t_q = \text{int}$  
$t_4 = t_x = t_{11} = t_r = \text{bool}$  
$t_2 = t_f = t_{10} = \text{bool} \rightarrow \text{int}$
Consider this variant program:

```lisp
(let ((f (lambda (x) (if x p #f))))
  (+ 1 (f r)))
```

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

The corresponding extracted constraints will be \textbf{inconsistent}; no solution can be found. Can report this to the programmer.

But which is wrong, the definition of \( f \) or the use? Doesn’t really work to associate the error message with a single program point. (In general, may need to consider an arbitrarily long chain of program points.)
Consider

\[
(\text{let } ((\text{second } (\text{fun } (xs) (\text{first } (\text{rest } xs)))))
\]
\[
(\text{second } (\text{list } 1 2 3)))
\]

By extracting the constraints as above, and solving, we will conclude that
\text{second} has type \text{int list} \rightarrow \text{int}.

It is also perfectly sensible to write:

\[
(\text{let } ((\text{second } (\text{fun } (xs) (\text{first } (\text{rest } xs)))))
\]
\[
(\text{second } (\text{list } \#t \#f \#t)))
\]

giving \text{second} the type \text{bool list} \rightarrow \text{bool}. Note that the definition of
\text{second} hasn’t changed at all!

So reasonable to ask: why can’t we write something like:

\[
(\text{let } ((\text{second } (\text{fun } (xs) (\text{first } (\text{rest } xs)))))
\]
\[
(\text{if } (\text{second } (\text{list } \#t \#f \#t))
\]
\[
(\text{second } (\text{list } 1 2 3)) \ 42))
\]

We can do this if we treat the type of \text{second} as \text{polymorphic}. 
By default ML infers the most polymorphic possible type for every function. In this case, it would give `second` the type `’a list → ’a`, where `’a` (pronounced “alpha”) is implicitly universally quantified. Each use of `second` occurs at a particular `instance` of `’a` (first at `bool`, then at `int`).

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

In this model, the **behavior** of the polymorphic function is **independent** of the instantiating type. In fact, an ML compiler typically generates just one piece of object code for each polymorphic function, shared by all instances. (More later.) An alternative is to generate type-specific versions of the code for each different instance.
Most languages provide some form of **overloading**, where the same 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 reals is very common.

Aim is to do “what we expect;” rules can get quite complicated (especially when **coercions** are considered)!

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

In conventional languages, overloading is resolved **statically**; that is, the compiler selects the appropriate version of the operator once and for all at compiler time. (Different from object-oriented dynamic overriding; more later.)

Overloading is sometimes called “**ad-hoc polymorphism**”. It is fundamentally **different** from parametric polymorphism, because the implementation of the overloaded operator changes according to the underlying types.