Dynamic Type Checking

- Static type checking offers the great advantage of catching errors early.

- And it generally supports more efficient execution.

- So why ever consider dynamic type checking?

- Simplicity. For short or simple programs, it’s nice to avoid the need to declare the types of identifiers.

- Flexibility. Static type checking is inherently more conservative about what programs it allows.
Conservative Typing

For example, suppose + is defined on both strings and numbers (but not mixtures of the two). Then

$$(\text{if } b \text{ then } "a" \text{ else } 2) + (\text{if } b \text{ then } "c" \text{ else } 3)$$

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

Dynamic typing allows container data structures to contain mixtures of values of arbitrary types, e.g.

```
List(2, true, 3.14)
```
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

\[
\begin{align*}
\text{(let } f \text{ (fun } (x) (+ x 2)) & \text{ \hfill (\@ f y))} \\
\text{The type of } x \text{ must be } \text{int} \text{ because it is used as an arg to } +. \text{ So the type of } f \text{ must be } \text{int} \to \text{int} \text{ (i.e. the type of functions that expect an int argument and return an int result), and } y \text{ must be an int.}
\end{align*}
\]

\[
\begin{align*}
\text{(let } f \text{ (fun } (x) \text{ (cons } x \text{ nil))} & \text{ \hfill (\@ f \text{ true}))} \\
\text{Suppose } x \text{ has some type } t. \text{ Then the type of } f \text{ must be } t \to \text{(list } t). \text{ Since } f \text{ is applied to a bool, we must have } t = \text{bool.}
\end{align*}
\]

For the moment, we assume that \( f \) must be given a unique \textbf{monomorphic} type; we will relax this later…
Systematic Inference

Here's a harder example:

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

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:

\[ \begin{align*}
    TE + \{ x \mapsto t_1 \} \vdash e : t_2 \\
    \quad \frac{}{TE \vdash \text{fun}(x) \ e : t_1 \to t_2} \quad \text{(Fn)}
\end{align*} \]

\[ \begin{align*}
    TE \vdash e_1 : t_1 \to t_2 \\
    \quad \frac{TE \vdash e_2 : t_1}{TE \vdash \text{@}(e_1 \ e_2) : t_2} \quad \text{(Appl)}
\end{align*} \]
Solution: \( t_1 = t_7 = t_8 = t_9 = t_3 = t_5 = t_p = t_6 = t_q = \text{int} \)
\( t_4 = t_x = t_{11} = t_r = \text{bool} \quad t_2 = t_f = t_{10} = \text{bool} \to \text{int} \)

**Node** | **Rule** | **Constraints**
--- | --- | ---
1 | Let | \( t_f = t_2 \quad t_1 = t_7 \)
2 | Fun | \( t_2 = t_x \to t_3 \)
3 | If | \( t_4 = \text{bool} \quad t_3 = t_5 = t_6 \)
4 | Var | \( t_4 = t_x \)
5 | Var | \( t_5 = t_p \)
6 | Var | \( t_5 = t_q \)
7 | Add | \( t_7 = t_8 = t_9 = \text{int} \)
8 | Int | \( t_8 = \text{int} \)
9 | Appl | \( t_{10} = t_{11} \to t_9 \)
10 | Var | \( t_{10} = t_f \)
11 | Var | \( t_{11} = t_r \)
Drawbacks of Inference

Consider this variant example:

(let f (fun (x) (if x p false))
 (+ 1 (@ f r)))

Now the body of $f$ returns type bool, but it is used in a context expecting an 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.
Polymorphism

Consider

\[
\text{let } \text{snd} = (\text{fun } l \rightarrow \text{head (tail } l))
\]

\[
\text{snd} : \text{(list int)} \rightarrow \text{int}
\]

By extracting constraints and solving, we will get

\[
\text{snd} : \text{(list int)} \rightarrow \text{int}
\]

Same definition!

We could also write

\[
\text{let } \text{snd} = (\text{fun } l \rightarrow \text{head (tail } l))
\]

\[
\text{snd} : \text{(list bool)} \rightarrow \text{bool}
\]

And get

\[
\text{snd} : \text{(list bool)} \rightarrow \text{bool}
\]
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.
We can think of these polymorphic types as being universally quantified over their type variables and instantiated at use sites:

```plaintext
let snd : \forall t. list t -> t = ...
in snd \{bool\} (true::false::nil);
snd \{int\} (1::2::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 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 *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 *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 *type classes* provide an unusual form of dynamic overloading with a static guarantee that a suitable version exists.