Polymorphism

Consider

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

By extracting the constraints as above, and solving, we will conclude that
snd has type (list int) → int.

It is also perfectly sensible to write:

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

giving snd the type (list bool) → bool. Note that the definition of snd
hasn’t changed at all!

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

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

Can do this if we treat the type of snd as **polymorphic**.
Inferring Polymorphism

In fact, if we extract constraints and solve just for the definition (fun l (head (tail l))) without considering its uses, we will end up with the type (list t) -> t where t is an unresolved type variable.

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

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

This is called \textit{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 \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.