Lesson 8
POLYMORPHISM

June 8, 2023

1. Type Inference
2. Parametric Polymorphism
3. Parameterized Datatypes
4. Polymorphic Sorting

Last time, we discussed the **tree method** and how it can be used to solve more difficult recurrences, that make multiple recursive calls.

We also learned how to analyze complexity of a function on trees via depth as opposed to the number of nodes. We saw that this obtained different bounds, but through reasoning about the relationship between nodes and depth, ultimately came out to be the same .

We then explored sorting, and wrote a terse implementation of **merge sort**, and then analyzed its work and span.

1 - Type Inference

You may have noticed that we've been using `int`s all over the place.

We've so far dealt with lists of ints, trees of ints, and sorting ints. We've been focusing on things of type `int` quite a bit, but we haven't had to! We might be interested in data structures which hold things which are not just integers, for instance.

If we did that, however, we would need to redo every lecture that we've done up to this point, because everything we've discovered has only been for the type `int`! If only there were some way of doing things in a generic way, for any type.

Recall the `length` function we defined long ago:

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length xs
```

This only works on lists of type `int`, but there's no reason for it to! We could have easily written it as:

```
fun length ([] : int list) : int = 0
  | length (_::xs) = 1 + length xs
```

and never inspected any specific element of the list at all.

We have been type annotating our code the entire semester. In keeping with our Big Idea that **Types Guide Structure**, we want to make sure that the structure of our code is easily described via types.

This is often a lot of work, however! We know that SML is able to automatically determine whether an expression is ill-typed – that is, if there types are used in a contradictory way. Is SML also able to determine the type of programs automatically?

The answer: **It is!**

SML performs **type inference**, which is the automatic derivation of types for all expressions. When this process fails, the program is rejected as ill-typed.[1]

The type inference algorithm is basically a straightforward recursive algorithm on an expression. When we encounter a function that we know to have a certain type, we assume that it type-checks, and then recurse to see if there are any contradictions.

[1] You might say that the SML type system has type `expression -> typ option`.

Suppose we are inferring a type for the expression `2 + 3`.

Since this is just the application of a function `+ : int * int -> int`, we assume it has type `int`.

We then check whether `2 : int`, which it does, so no contradiction.

We then check whether `3 : int`, which it does, so no contradiction.

So `2 + 3` type-checks, and additionally, has type `int`.

It's more interesting when we get into examples with functions!

When a function accepts an argument, it has a yet-unknown type. When inferring the type of such a variable, the type inference algorithm must use the context of *how the variable is used* to determine the variable's type.

For instance, consider the expression `fn x => x + 1`.

Upon entering the body of the expression, we assign `x` a yet-undetermined type, and type-check the application of `+` to the tuple `(x, 1)`.

In this instance, `+ : int * int -> int`, so it must be the case that `x : int`! Thus, we have resolved a type for the variable `x`.

What about a function like `fn x => (x + 1, x ^ "1")`?

Here, we again assign `x` an unknown type, and then proceed to type-check the body of the function. We see the expression `x + 1`, and by the same reasoning as earlier, conclude that `x : int`.

However, we then type-check `x ^ "1"`, and conclude that, since `^ : string * string -> string`, it must be the case that `x : string`. This is a contradiction, because we previously concluded `x : int`! So type-checking fails.

The previous example showed an expression that was ill-typed, due to too much information being specified. We added enough constraints onto the type of `x` that we couldn't possibly satisfy them all!

What happens if we specify too **few** constraints?

What's the type of the expression `fn x => x`?

2 - Parametric Polymorphism

Before, we talked about the type inference algorithm as assigning an "unspecified type" to each variable, that could be later solved for.

We are now ready to give a name to that type!

Def A **type variable** is the type given to an expression whose type is not known. The role of the type inference algorithm is to solve for what that type variable's type should be.

A type variable is named such because it is a variable, **ranging over types**. In the same way that variables as introduced by lambda expressions are meant to be substituted with values, type variables as introduced by the type inference algorithm are meant to be substituted with types.

These type variables are denoted via a backtick before a letter, like `'a`, and called their Greek letter equivalent (in this case, **alpha**).

Before, we asked about what the type of `fn x => x` is. Here, we enforce **zero constraints** on the type of `x`, so its type remains the same as it was originally initialized to, which is a type variable, in this case `'a`.

So we say that this function has type `'a -> 'a` (or "alpha to alpha").

What's the point of having such a function? Usually, we want to make sure that our code has a single, specified type, so that we don't make mistakes! What gives?

The reason why this is useful is for **code reuse**.

Def We say that an expression has a **polymorphic type** if it has a type that contains type variables.

Another name for the function we just defined, `fn x => x`, is the **identity function**. It is a polymorphic function.

Suppose we are interested in using the identity function. If we previously defined it with type annotations, we would have something like:

```
fun identityInt (x : int) : int = x
```

and we would only be able to use it like `identityInt 150`, but `identityInt "hi"` would be ill-typed.

This is really really annoying, because now we have to go and define one for every single type that we're interested in.

```
fun identityString (x : string) : string = x
fun identityBool   (x : bool)   : bool   = x
```

The key observation is that the actual **contents** of the function are the same. They all look like some form of `fn x => x`.

So why should we need to restate this definition for every single type? We should be able to use the actual code that we wrote, irrespective of type annotation, if the **meaning** of the code is agnostic to type.

The key insight to maintaining type safety is a variant of polymorphism called **parametric polymorphism**.

Def We say a type is **parametrically polymorphic** if it is parameterized by one or more type variables, which are instantiated at a later time by substitution.

So for instance, the type `'a -> 'a` is parametrically polymorphic, because it is parameterized by the type variable `'a`. This means that we can write:

```
fun identity (x : 'a) : 'a = x
val x : int = identity 150
val y : string = identity "hi"
```

This allows a notion of **generic** code, which is code that can be used generically in its type![2] Essentially, the type `'a -> 'a` means "for all types `a`, `a -> a`".

In the previous example, `identity` is bound to a generic type, meaning that it can be instantiated concretely as `int -> int` in the RHS of the binding to `x`, and as `string -> string` in the other case.

In this case, we would call the two calls to `identity` as different instances of the same function. They are concrete cases of a general function template, known as `identity`.

[2] Some languages actually do call this "generics".

The specific kind of polymorphism employed by SML is known as **let-polymorphism**.[3]

Def **Let-polymorphism** means that values can only be **generalized** as polymorphic after their declaration site.

This doesn't really come up, but this can happen if you're expecting a function to be polymorphic *while you're defining it!*

[3] It's not really important to know why it's called this.

For instance, this function fails to be polymorphic:

```
fun identity x =
  let
    val _ = identity 5
  in
    x
  end
```

Even though it seems like it "should" still be polymorphic, the use of `identity` concretely within its own definition causes it to be typed at `int -> int`. It is only after a function is defined, that it is able to be used generically.

The identity function is kind of a contrived example, however. We don't use it that often.[4]

As we alluded to earlier, the `length` function we wrote needed not be specified at any particular type, because we never actually look at any of the elements of the list! We could indeed write:

```
fun length [] = 0
  | length (x::xs) = 1 + xs
```

and obtain `length : 'a list -> int`.

[4] Yet.

The process by which we determine what the type of a function without annotations is, is an entirely predictable one.

We can try to do it ourselves. Suppose we have the following function:

```
fun foo (a, b, c, d) =
  if a then
    (b + 1, c)
  else
    d ()
```

How are we going to determine the type of this function? First, we start off with all of the types of our arguments as unknowns.

```
fun foo (a, b, c, d) =
  if a then
    (b + 1, c)
  else
    d ()
```

$$\texttt{a} \mapsto \texttt{'a}$$
$$\texttt{b} \mapsto \texttt{'b}$$
$$\texttt{c} \mapsto \texttt{'c}$$
$$\texttt{d} \mapsto \texttt{'d}$$

But, we notice that `a` is the subject of an `if`, meaning that it must be of type `bool`!

```
fun foo (a, b, c, d) =
  if a then
    (b + 1, c)
  else
    d ()
```

$$\texttt{a} \mapsto \texttt{bool}$$
$$\texttt{b} \mapsto \texttt{'b}$$
$$\texttt{c} \mapsto \texttt{'c}$$
$$\texttt{d} \mapsto \texttt{'d}$$

We then see that we use `b` by adding `1` to it, implying it must be of type `int`.

```
fun foo (a, b, c, d) =
  if a then
    (b + 1, c)
  else
    d ()
```

a ↦ bool

b ↦ int

c ↦ 'c

d ↦ 'd

We also apply the variable `d` as a function, given an input of type `unit`. We don't know the output type, though, so let's just make it another type variable `'e`.

```
fun foo (a, b, c, d) =
  if a then
    (b + 1, c)
  else
    d ()
```

$$\texttt{a} \mapsto \texttt{bool}$$
$$\texttt{b} \mapsto \texttt{int}$$
$$\texttt{c} \mapsto \texttt{'c}$$
$$\texttt{d} \mapsto \texttt{unit -> 'e}$$

Then, because we know that the types of two branches of an `if` must be the same, we conclude that `(b + 1, c)` must be the same type as the output type of `d`.

```
fun foo (a, b, c, d) =
  if a then
    (b + 1, c)
  else
    d ()
```

$$\texttt{a} \mapsto \texttt{bool}$$
$$\texttt{b} \mapsto \texttt{int}$$
$$\texttt{c} \mapsto \texttt{'c}$$
$$\texttt{d} \mapsto \texttt{unit -> int * 'c}$$

Finally, this is also the return type of the function, so the ultimate type of our function is `bool * int * 'c * (unit -> int * 'c) -> int * 'c`.[5]

[5] A very useful type.

3 - Parameterized Datatypes

We have seen that we can have types such as `'a -> 'a` and `'a list -> int`, which rely on type variables. These are polymorphic types that are "pre-existing", in the sense that we did not need to define the types of `list`, or the `->` type constructor.

We can also define our own types that are parameterized by other types!

Def A **parameterized datatype** is a datatype declared with a **type parameter**. This defines many types, which accept a type as an input.

Note The `list` and `option` datatypes as discussed earlier in the course are examples of parameterized datatypes!

This is the reason why lists and options can contain values of any type.

We can define lists and options like so:

```
datatype 'a list = [] | :: of 'a * 'a list

datatype 'a option = NONE | SOME of 'a
```

These datatype declarations are similar, in the sense that they define the `'a list` and `'a option` types, which are in a sense templates for a family of types. These can be instantiated concretely as `int list`, `string option`, and so on and so forth.

Note The two uses of `'a` in the type of `::` are the same! This means it is still invalid to write `2::[true]`, because `2 : int` and `[true] : bool list`.

Finally, we see the truth! This is what we have been tiptoeing around, when discussing `[]` and `::`.

Previously, we had to say things like:

Def For any type `t`, `[] : t list`, and `:: : t * t list -> t list`.

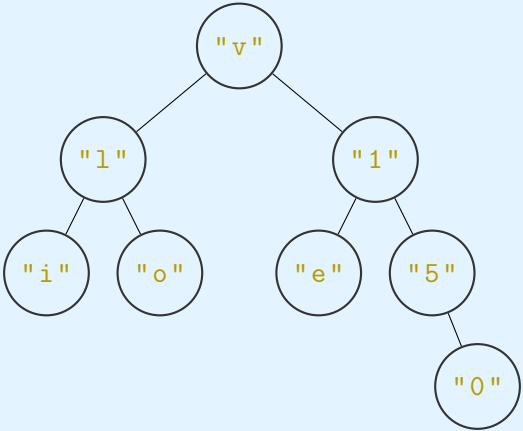
This quantification around the type of the list's elements, `t`, was all due to the type variable in the types of `[]` and `::`. In reality, what we could have just said was that `[] : 'a list`, and `:: : 'a * 'a list -> 'a list`.

We can also define polymorphic trees! Before, our trees only contained `int`s. But there is no reason for that.

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Now, we can have:

- `Empty : 'a tree`
- `Node (Empty, 5, Empty) : int tree`
- `Node (Empty, "hi", Empty) : string tree`
- `Node (Empty, Empty, Empty) : 'a tree tree`



Note What the future looks like, now that we can put strings in trees.

The fact that we can write these polymorphic functions on `list`s and `tree`s and the like is a *result* of the fact that they are polymorphic types!

We can write functions which respect the *structure* of the type outside of the polymorphic parts, and are still able to do useful work.

```
fun count Empty = 0
  | count (Node (L, x, R)) = count L + 1 + count R

fun safeValOf (default, NONE)   = default
  | safeValOf (default, SOME v) = v
```

Through these declarations, it is possible to achieve polymorphic values that are not functions. While it may seem like type variables are only introduced by function arguments we do not know the type of, they are really introduced by any ambiguity in a type.

For instance, what is the type of `[]`? We don't have enough information to constrain it either way, so the type is just `'a list`. This means that we could write the following code:

```
val l : 'a list = []
val x : int list = 1 :: l
val y : string list = "hi" :: l
```

The two instances of `l` are instantiated at `int list` and `string list`, respectively!

It is now no longer true that each expression has only one type.

For certain expressions like `fn x => x`, it could be interpreted to have multiple types, such as `int -> int`, or `'a -> 'a`.

To explain away this ambiguity, we will revise our interest in types to be for the **most general type** for an expression.

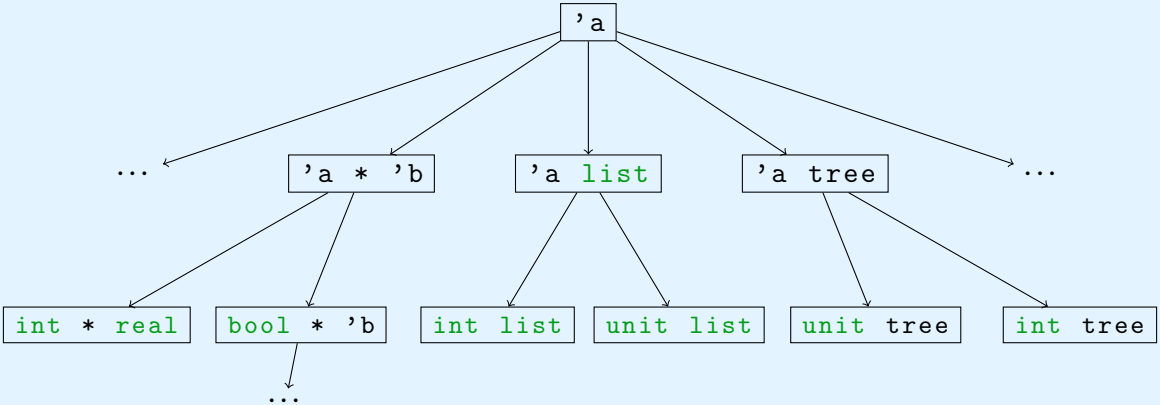
Def The **most general type** for an expression is the type that every other type for that expression is an instance of.

We spoke briefly on how different calls to `identity` were instances of the main declaration. We also define a concept of **instances** on types.

Def A type `t1` is an **instance** of type `t2` if `t1` can be obtained from `t2` by substituting for some type variables.

So here are some examples:

- `int` is an instance of `'a`
- `int list` is an instance of `'a list`
- `int * int` is an instance of `'a * 'b`
- `int * bool` is an instance of `'a * 'b`
- `'b * 'c` is an instance of `'a`
- `'a list list` is an instance of `'a list`



Note that only the type `bool * 'b` has further instances, because it's the only one with a remaining type variable.

We say that the rest of the types are **monomorphic**.

So we see that the MGT of `length` is `'a list -> int`, because while it can be typed as `int list -> int`, and `string list -> int`, among others, they are all just instances of the type `'a list -> int`.

All parametric polymorphism is, is choosing instances of a most general type!

Warning Be careful not to confuse being an **instance** of a type with a **most general type**. All types are instances of `'a`, but most expressions do not have MGT `'a`.

Beware the following common mistake on proofs!

Thm. `length (L @ R)` $\cong$ `length L + length R`

When proving this claim on `length : 'a list -> int`, you might be tempted to write the following:

We proceed by structural induction on `L : 'a list`.

BC `L = []`

...

This will get a point deduction! The quantification is incorrect.

There is only one value of type `'a list`, so how can we possibly induct on it? To add another element would be to escape the type entirely, because that would specify the type of the list!

The proper way to phrase the proof is:

Let `t` be a type. We proceed by structural induction on `L : t list`.

...

IH Case: `L = xs`, for some `xs : t list`. Assume ...
IS Case: `L = x::xs`, for some `x : t`. Let's show ...

When proving a claim on values of a parametric type, you are essentially writing a proof for many types. To do this properly, you need to parameterize your proof by an arbitary type, and then proceed with the proof. This which will end up proving the claim for the parametric type.

Another reason why the former proof doesn't work is that there are **no values of type** `'a`. This would entail a value which could be instantiated at any type, which obviously shouldn't exist!

The conceptual takeaway is that a polymorphic type is a **family of types**, parameterized by the input type. It is a **schema** that defines many other types.

A function with a polymorphic type is a **family of functions**, all with the same implementation, but each with a different type depending on its usage.

A proof on a function with a polymorphic type is a **family of proofs**, one for each input type that the function could be instantiated at. We essentially are proving something for each function in the family of functions.

4 - Polymorphic Sorting

We've now addressed our over-reliance on `int`s in the context of functions like `length` and `identity`. Let's turn our attention towards a more classic problem, though.

We previously discussed sorting, but only on integers! Let's abstract and try to create a **generic** sorting function. We'd like a function:

```
sort : 'a list -> 'a list
```
REQUIRES: `true`
ENSURES: `sort L` is a sorted permutation of `L`

...But is that even a well-specified problem? What would it mean to have a function of type `'a list -> 'a list`?

Recall that if we impose any constraints on what the type of the elements of the list are, then it will no longer generalize polymorphically. We have to somehow sort elements of a list without ever looking at what any given element is.

It turns out, this function is impossible to write. Not in the least for the reason that "sorted" is a relative concept. We have an idea of a canonical sorting for integers – by magnitude. But what if we wanted to sort integers in reverse? Or modulo 12?

All these notions involve what is called a **comparison function**.

Def We say that a function `f : t * t -> order` is a **comparison function** if it is total, and defines a **total order**

Def A **total order** is a binary relation that essentially relates a bunch of things which can be ordered on a line. So a total order `f` should not have any cycles.

So we might define these comparison functions as:

```
val revIntCompare =
  fn (x, y) => case Int.compare (x, y) of
                 LESS => GREATER
               | GREATER => LESS
               | EQUAL => EQUAL
val mod12Compare =
  fn (x, y) => Int.compare (x mod 12, y mod 12)
```

We can, relative to a comparison function `cmp : t * t -> order`, recursively define what it means for a list to be sorted with respect to that comparison function.

Def We say that a list `L : t list` is `cmp`**-sorted** if it has the following properties:

- `L` ≅ `[]`
- `L` ≅ `[x]`
- `L` ≅ `x::y::xs`, where `cmp (x, y)` ≅ `LESS` or `EQUAL`, and `y::xs` is `cmp`-sorted.

To be able to sort relatively, we will adjust the type of our `sort` function, slightly.

Now, instead of being of type `'a list -> 'a list`, we will add a parameter to our `sort` function, which is a comparison function by which to sort.

But wait, why can we take in a comparison function? Can functions be passed in as arguments?

This might come as a surprise[6], but functions are values.

We've seen it, by binding lambda expressions in `val` declarations, which is an equivalent form to the `fun` declarations we've been using so far.

Another consequence of this is the fact that functions are **first-class citizens**. This means that they can be bound to values, passed into functions, and returned from functions, just like any other value.

[6] The same way that an annual tuition increase is a surprise.

We will discuss this more in-depth next lecture, but for this lecture, we will exploit the fact that functions can be passed in as arguments, to write our `sort` function in a way that it is `parameterized` by a comparison function.

Essentially, what the `sort` function does, depends on what the comparison function is!

```
sort : ('a * 'a -> order) * 'a list -> 'a list
```
REQUIRES: `true`
ENSURES: `sort (cmp, L)` is a `cmp`-sorted permutation of `L`

Let's implement this via our old definition of `msort`.

```
fun split [] = []
  | split [x] = [x]
  | split (x::y::xs) =
      let
        val (A, B) = split xs
      in
        (x::A, y::B)
      end

fun merge ([], R) = R
  | merge (L, []) = L
  | merge (x::xs, y::ys) =
      if x < y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)

fun msort [] = []
  | msort [x] = [x]
  | msort L =
      let
        val (A, B) = split L
      in
        merge (msort A, msort B)
      end
```

```
fun split [] = []
  | split [x] = [x]
  | split (x::y::xs) =
      let
        val (A, B) = split xs
      in
        (x::A, y::B)
      end

fun merge (cmp, ([], R)) = R
  | merge (cmp, (L, [])) = L
  | merge (cmp, (x::xs, y::ys)) =
      case cmp (x, y) of
        LESS => x :: merge (cmp, (xs, y::ys))
      | _ => y :: merge (cmp, (x::xs, ys))

fun msort (cmp, []) = []
  | msort (cmp, [x]) = [x]
  | msort (cmp, L) =
      let
        val (A, B) = split L
      in
        merge (cmp, (msort (cmp, A), msort (cmp, B)))
      end
```

All we did was pass the `cmp` function through, and then change a single place (where we used to do the integer comparison), instead using the provided comparison function. The code easily generalizes!

Now, we can define our `sort` function simply as:

```
fun sort (cmp : 'a * 'a -> order, L : 'a list) : 'a list =
   msort (cmp, L)
```

Now we can sort generically! For instance, we have:

- `sort (Int.compare, [2, 3, 1])` ↪ `[1, 2, 3]`
- `sort (String.compare, ["a", "ab", "b"])` ↪ `["a", "b", "ab"]`
- `sort revIntCompare [1, 2, 3]` ↪ `[3, 2, 1]`

Thank you!