Thus far, we’ve been focusing a lot on fundamentals: writing parallel functional programs (key tool: recursion), analyzing work and span (key tools: recurrences, big-O), and proving correctness (key tools: induction, equivalence). These are the basic ingredients of functional programming. But, to a large extent, we haven't been taking advantage of the things that make ML fun and elegant to program in. Over the next few lectures, we’re going to introduce some new features of ML that will make code a lot more concise and pretty.

1 Type Constructors

At these point, we’ve seen a few different types of lists, but we’ve never actually discussed how or why we can have different types of lists. The idea of a list is not specific to integers. Here’s a list of strings:

val s : string list = "a" :: ("b" :: ("c" :: []))

We can also have lists of lists of integers, following exactly the same pattern:

val i2 : (int list) list = [1,2,3] :: [4,5,6] :: []

Note that (int list) list can also be written int list list.

In fact, there is a type T list for every type T. And we can reuse [] and :: for a list with any type of elements. This abstracts over having different nils and conses for different types of lists. We call list a type constructor because if we apply list to a type, we get another type: e.g. if we apply list to int, that gives us int list. You could also think of this as sort of like a function that takes a type and computes another type: if we plug in the argument int, then the list constructor gives us back int list, the type of lists of integers (type constructors use postfix notation, meaning that the argument appears on the left).

2 Polymorphism

Some functions work just as well for any kind of list:

*Based on notes by Brandon Bohrer, Mike Erdmann and others
fun length (l : int list) : int =
case l of
  [] => 0
| x :: xs => 1 + length xs

fun length (l : string list) : int =
case l of
  [] => 0
| x :: xs => 1 + length xs

What’s the difference between this code and the above? Nothing but the type annotation! You can express that a function is polymorphic (works for any type) by writing

fun length (l : 'a list) : int =
case l of
  [] => 0
| x :: xs => 1 + length xs

The type of length is

length : 'a list -> int

and it’s implicit in this that it means “for all 'a”. Here 'a (which is pronounced “alpha” and typeset as α) is a type variable that stands for any type 'a. You can apply length to lists of any type:

val 5 = length (1 :: (2 :: (3 :: (4 :: (5 :: []))))))
val 5 = length ("a" :: ("b" :: ("c" :: ("d" :: ("e" :: []))))))
val 2 = length ([1,2,3] :: [4,5,6] :: [])

Note that even though you can apply length to any type of list, you still can’t have a list where the elements are different types

(* NOT ALLOWED!! * )
val 2 = length (1 :: ("foo" :: []))

Even though the argument to length is allowed to have type 'a list for any type 'a, it still has to be a well-typed list for some type 'a! To type-check a function call to length, we first figure out the type of the argument to length, then make sure that type looks like 'a list for some 'a. If we try to type-check (1 :: ("foo" :: [])) it doesn’t have a type, so the expression length (1 :: ("foo" :: [])) doesn’t type-check, either.

Here’s another example, zip, a function that takes a pair of lists and turns it into a list of pairs.

fun zip (l : int list, r : string list) : (int * string) list =
case (l,r) of
  ([],_) => []
| (_,[]) => []
| (x::xs,y::ys) => (x,y)::zip(xs,ys)
Does it depend on the element types? No: it's just a structural transformation; it just shuffles elements around. So we can say

```haskell
fun zip (l : 'a list, r : 'b list) : ('a * 'b) list =
  case (l,r) of
    ([],_) => []
  | (_,[]) => []
  | (x::xs,y::ys) => (x,y)::zip(xs,ys)
```

instead. Note that both \(\alpha\) and \(\beta\) here are universally quantified, so they might stand for different types (like \texttt{int} and \texttt{string}) or they could also happen to be the same type (e.g. if we want to zip two \texttt{int} lists). On the other hand, it's important that the type variables \(\alpha\) and \(\beta\) in the return type are the same type variables as we have in the argument type, because the return value for \texttt{zip} is computed from the input lists.

That is, we can abstract over the pattern of zipping together two lists, and do it for all element types at once! This saves you from having to write out \texttt{zip} every time you have two kinds of lists that you want to zip together, which would be bad, since:

- it's annoying to write that extra code, and
- it's hard to maintain: when you find bugs you have to make sure you fix it in all the copies.\(^1\)

It's worth pointing out that SML has what we call \textit{parametric polymorphism}. Polymorphic functions work the same on arguments of any type; their behavior cannot depend on the types of their arguments. What this means is that there's no operator in the language that lets us test the type of expressions. For those of you familiar with Java, this means there's no analog of Java's \texttt{instanceof} operator.

Note that \texttt{[]} and \texttt{::} are polymorphic, in that they work for lists built from any type. That is to say,

\[
\begin{align*}
  [] & : 'a list \\
  :: & : 'a * 'a list \rightarrow 'a list 
\end{align*}
\]

We'll talk about how to make polymorphic constructors in a little bit.

### 3 A shameless plug for later PL classes

In addition to cutting down on the amount of code we have to write and making our code more versatile, polymorphism also reduces the number of incorrect programs we can write by giving us more information about what a function does.

For example, there are a whole host of (total) functions we can write with type \texttt{int -> int}:

```haskell
fun (x : int) : int = x
fun (x : int) : int = ~x
fun (x : int) : int = x + 1
fun (x : int) : int = 42
```

\(^1\)This will happen to you, if it hasn’t already.
In the subset of SML we’ve learned so far, we can write precisely one total function (up to
extensional equivalence) of type \( 'a \rightarrow 'a \):

\[
\text{fun (x : 'a) : 'a = x}
\]

The intuition is that since we don’t know whether \( 'a \) is \texttt{int}, \texttt{string}, \texttt{int list}, \texttt{etc.}, we can’t
do anything clever by looking at it, and we can’t conjure something of that type out of thin air, so
all we can do is return the argument.

We can summarize this as a theorem:

**Theorem 1.** If \( f : 'a \rightarrow 'a \) and \( x : 'a \) and \( y : 'a \) are values, and

\[
f x \sim y
\]

then

\[
x \sim y
\]

The proof is way beyond the scope of the class, but if you’re interested, you should take 15-312
(which might go into this) and 15-814 (which will).

### 4 Type inference

Here’s another way to make your code easier to read: leave off unnecessary type annotations. (We’ll
talk about what’s necessary in a minute.) Then *type inference* will fill in the types for you.

For example:

\[
\text{fun length l = case l of } \begin{array}{ll}
  [] & \rightarrow 0 \\
  \mid x :: xs & \rightarrow 1 + \text{length} \; xs
\end{array}
\]

To figure out the type of this function, we

- annotate with type variables
- generate and solve constraints by analyzing the structure of the program

There are three ways this can end:

1. First, we can observe that we have polymorphic functions. Consider the example:

\[
\text{fun length (l : 'a1) : 'a2 = case l of } \begin{array}{ll}
  [] & \rightarrow 0 \\
  \mid x :: xs & \rightarrow 1 + \text{length} \; xs
\end{array}
\]

We can reason as follows:

- Because the \texttt{fun} keyword introduces a name for a function, we know that for some \( \alpha_1 \)
  and \( \alpha_2 \), \texttt{length} has type \( \alpha_1 \rightarrow \alpha_2 \).
Therefore, the argument \( l \) has some type \( \alpha_1 \) and the result is some type \( \alpha_2 \).

Because \( l \) gets case-analyzed with nil and cons patterns, it must be some kind of list, so we get the constraint \( \alpha_1 = \alpha \text{ list} \) for some type \( \alpha \).

Because 0 gets returned from the function, we get the constraint \( \alpha_2 = \text{int} \).

The other branch of the case-statement must also have type \( \text{int} \), because the branches of the case must agree. In that branch, we get to assume that \( x \) has type \( \alpha \) and \( \text{xs} \) has type \( \alpha \text{ list} \) because that’s our current best-guess for the type of \( l \).

\[
\begin{align*}
\alpha_1 & = \alpha \text{ list} \\
\alpha_2 & = \text{int}
\end{align*}
\]

Since we’ve run out of code to look at, this is the most information that we have, so we finish with

This system of equations is underconstrained: these equations do not constrain \( \alpha \). So \text{length} is polymorphic. This makes us very happy! We can use \text{length} on any type of list!

2. Second, we can get not-polymorphic but well-typed functions. Consider the example

\[
\text{fun sum \( l \) =} \quad \begin{align*}
\text{case } l \text{ of} \\
\text{[]} & \Rightarrow 0 \\
\text{x :: xs} & \Rightarrow x + \text{sum xs}
\end{align*}
\]

If we do the same structural reasoning, then we get the constraints

\[
\begin{align*}
\alpha_1 & = \alpha \text{ list} \\
\alpha_2 & = \text{int} \\
\alpha & = \text{int}
\end{align*}
\]

The last equation comes from the fact that in the second branch \( x \) has type \( \alpha \), and the \text{+} function is applied to \( x \). These equations have a unique solution, where \( \alpha_1 = \text{int list} \), so \text{sum} does not have a polymorphic type. This makes us less happy than having a polymorphic function, but we’re still a little bit happy - at least our code compiles! Non-polymorphic functions are good too.

3. Third, type inference can fail because the expression is ill-typed. If we’d messed up the base case, for example, and written

\[
\text{fun sum \( l \) =} \quad \begin{align*}
\text{case } l \text{ of} \\
\text{[]} & \Rightarrow "hi" \\
\text{x :: xs} & \Rightarrow x + \text{sum xs}
\end{align*}
\]
then we’d get the constraints

\[ \alpha_1 = \alpha \text{ list} \]
\[ \alpha_2 = \text{string} \]
\[ \alpha_2 = \text{int} \]
\[ \alpha = \text{int} \]

This system could only have a solution if \texttt{string = int} was true. But \texttt{string = int} is a contradiction, so there’s no solution. The code is ill-typed. This makes us sad, since our code doesn’t compile. And even though we’re a little bit sad, this is still very useful information!

SML type inference is actually guaranteed to find us the most general type, which means that if a function could possibly be well-typed, SML will find the type. And if a function can possibly be polymorphic, SML will find a polymorphic type. So if type-inference can’t find a type for our function, we know it really is ill-typed and we need to change the code.\(^2\)

So if you leave off all the type annotations, SML will happily fill them in for you. It’s even guaranteed to find the most general type possible, so you’ll often discover that code you’d written at one type has a much more general type than you thought. Does that mean that you should always leave them off? No. Type annotations serve three very important purposes:

1. Type annotations let you communicate with the type checker, which helps you understand your code for debugging purposes. If you believe an expression should have a certain type, you can annotate it with that type and the type checker will tell you whether it really has that type or not. This is really useful if you’re dealing with confusing code that has complicated types.

2. Type annotations let the type checker communicate with you, by giving you better error messages. Because the most general type of your code might be more general than you expected, an error message phrased in terms of the most general type can be very unintuitive. By adding typing annotations, you can get error messages which are phrased in terms of the type that you actually wanted, which are usually easier to understand.

3. Type annotations let you communicate with people reading your code. Type annotations are a form of machine-checked documentation! If your types are chosen well, you can understand a function pretty well just by reading the name and the type. The importance of types being machine-checked can’t be overstated: It is often said that documentation is obsolete the moment you finish writing it. But your type annotations can never be obsolete, because if they were then the program wouldn’t even compile! \(^3\)

So you can leave off really redundant annotations for \texttt{val} bindings and such, but you should keep them on function declarations. And remember that adding more annotations is a useful technique for debugging confusing code.

\(^2\)These three possibilities: polymorphic, not-polymorphic, and ill-typed, are reminiscent of linear algebra. If a system is underconstrained, we have infinitely many solutions (one for each type \texttt{‘a}). If a system is perfectly constrained, we get a unique solution for the type. If a system is over-constrained, then the constraints contradict each other, so there are no solutions.

\(^3\)Documentation is important. A good memory is no match for a bad pen.
4.1 Debugging with types

The type information produced by the SML implementation can help to debug code! Consider the following erroneous sorting function, obtained from the above function by omitting the clause for singleton lists.

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

(This function is only useful for sorting the empty list! It doesn’t terminate when applied to a nonempty list! Nonetheless, it is well-typed. The surprise is that this type isn’t what we probably expected!).

SML tells us

```
val msort = fn - : 'a list -> int list
```

To check that this `msort` has type `'a list -> int list`, let’s check that each clause fits with this type.

(i) Clearly the first clause is fine: even if we give the argument ([ ] ) the type `'a list` the right-hand-side ([ ] ) can be used at the required result type (int list).

(ii) For the other clause, suppose we give L the type `'a list`. We need to show that the right-hand-side of the clause gets type int list.
    We know that `split` can be used at type `'a list -> 'a list * 'a list`. Then `split L` has type `'a list * 'a list`. The `val`-declaration will give both A and B the type `'a list`, and these type bindings are in scope when we figure out a type for the let-body. Assuming the type `'a list -> int list` for the recursive calls to `msort`, the expression `(msort A, msort B)` has type `int list * int list`. So `merge(msort A, msort B)` has type `int list`. This is the required result type, so the second clause is fine too.

OK, so SML says our `msort` has type `'a list -> int list`. Isn’t that a problem? Surely if we apply `sort` to a list of strings we can’t expect it to produce a list of integers??? Well, of course not. But there is no contradiction here. The SML type guarantee is that we can use `sort` in any way consistent with an instance of this type. So let L be a string list. If `sort(L)` terminates the guarantee is that its value will be an integer list. The only situation where this happens is for L=[ ], and the empty list is also a perfectly valid value of type `int list`.

The lesson: if SML tells you your function has a type that you didn’t expect, your code is likely to be wrong. (Or your expectations about types are mistaken.)

**Exercise.** Work through how SML figures out that the most general type of the function `trev` defined by

```
fun trev ([], acc) = acc
| trev (x::L, acc) = trev (L, x::acc)
```

is `'a list * 'a list -> 'a list`.

7
Exercise. Now find the most general type of \texttt{trev} given this definition:

\begin{verbatim}
fun trev ([], acc) = []
  | trev (x::L, acc) = trev(L, x::acc)

\end{verbatim}

(answer: \texttt{'a list * 'a list -> 'b list})

5 Equality and equality types

SML uses a special \textit{double-quoted} notation for type variables that range over types on which there is a sensible way to implement equality, and for which you can safely use = to test for equal values. For example, \texttt{'a}, \texttt{'b}, \texttt{'c} and so on are (ordinary) type variables, and can be instantiated with any type. And \texttt{''a}, \texttt{''b}, \texttt{''c} and so on are “equality type variables” and can only be instantiated with an equality type. The types \texttt{int} and \texttt{bool} are equality types, but \texttt{real} is not. When \texttt{t} is an equality type, so is \texttt{t list}. When \texttt{t_1,...,t_k} are equality types so is \texttt{t_1*...*t_k}.

For example:

\begin{verbatim}
fun mem(x:'a, L:'a list):bool =
  case L of
    [ ] => false
  | y::R => (x=y) orelse mem(x, R)

\end{verbatim}

introduces \texttt{mem :''a * ''a list -> bool}.

If we try to use an even more general type, look what happens:

\begin{verbatim}
fun mem(x:'a,L:'a list):bool =
  = case L of [ ] => false | y::R => (x=y) orelse mem(x,R);

\end{verbatim}

\texttt{stdIn:22.35-22.38 Error: operator and operand don’t agree [UBOUND match]}

\texttt{operator domain: ''Z * ''Z}

\texttt{operand: 'a * 'Y}

\texttt{in expression:}

\texttt{x = y}

The error message is trying to say that \texttt{'a} needs to be an equality type (of the form \texttt{''Z}).

For a type containing equality type variables, such as

\texttt{''a * ''a list -> bool}

we are only allowed to instantiate the equality type variables with equality types. So, for example, \texttt{int * int list -> bool} is OK as an instance of this type, but not \texttt{real * real list -> bool}.

6 Parametrized Datatypes

Earlier we introduced datatype definitions, and we considered a datatype of integer trees, given by

\begin{verbatim}
datatype inttree = Empty | Node of inttree * int * inttree
\end{verbatim}
This datatype definition introduces a type name `inttree`, a value `Empty` of type `inttree`, and a function `Node` of type `inttree * int * inttree -> inttree`.

However, the notion of tree is much more general, and it makes perfect sense to consider trees with values of a different type at the nodes, for example trees containing Booleans or trees containing integer lists. We can use type variables in datatype definitions to obtain parameterized types, like `'a tree`. We’ve already seen types like `'a list`, so this shouldn’t come as a surprise.

The following parameterized datatype definition

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

introduces a type constructor `tree` (used postfix), a value `Empty` of type `'a tree`, and a function `Node : 'a tree * 'a * 'a tree -> 'a tree`.

So for any type `t` we can work with values of type `t tree`, which are either `Empty`, or nonempty and of the form `Node(l,v,r)`, where `l` and `r` are values of type `t tree` and `v` is a value of type `t`.

As before, we can use the value constructors `Empty` and `Node` in patterns, and we can design functions that use pattern-matching on tree values.

To get the lists we know and love, and now know to be polymorphic, we could have typed in something like

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

### 7 Domain Specific Types

One really beautiful use of SML's ability to make new types is using them to precisely represent answers to specific questions. Anytime you use `datatype` you get case analysis and recursion (if the type you defined is recursive) for free, which makes `datatypes` a powerful feature.

#### 7.1 Order

We saw one example of this above, with `String.compare`. It has type `string * string -> order`, where order is built in and defined like

```sml
datatype order = LESS | EQUAL | GREATER
```

This type precisely represents answers to questions like “how does `x` compare to `y`?”. There are lots of comparison functions for built in SML types, like `Int.compare` and `Char.compare`, and it’s easy to write them as you create new types from old.

---

Note that this is already defined for you at the top level, so you can’t actually do this. `::` and `[]` are reserved words. The toplevel also implicitly infixes `::` so that you can write things like `1 :: []` rather than `::(1,[])`. You can infix any constructor on two arguments you like by using the keyword `infix` after it’s been introduced, as in `infix ::`. 

---

4Note that this is already defined for you at the top level, so you can’t actually do this. `::` and `[]` are reserved words. The toplevel also implicitly infixes `::` so that you can write things like `1 :: []` rather than `::(1,[])`. You can infix any constructor on two arguments you like by using the keyword `infix` after it’s been introduced, as in `infix ::`. 

9
7.2 Grades

We can make a type for a much more specific problem, though. Specifically, strings are a really bad representation of letter grades. There are only 5 letter grades that we can actually assign, so it’s bad if our type allows us to write other strings as grades. Certainly, you would be upset if you checked SIO and found that your grade for the class was “acorn”.

Defining our own datatypes lets us solve this problem. We can say that a letter grade is only allowed to be an actual grade, like A, B, C, D or R:

```plaintext
datatype letter = A | B | C | D | R
```

\[5\text{Though if this got misinterpreted as “A”, maybe you won’t mind.}\]