Modules for Namespace Management

We start by showing how ML modules can be used to separate bindings into different namespaces. We then build on this material to cover the much more interesting and important topic of using modules to hide bindings and types.

To learn the basics of ML, pattern-matching, and functional programming, we have written small programs that are just a sequence of bindings. For larger programs, we want to organize our code with more structure. In ML, we can use structures to define modules that contain a collection of bindings. At its simplest, you can write structure Name = struct bindings end where Name is the name of your structure (you can pick anything; capitalization is a convention) and bindings is any list of bindings, containing values, functions, exceptions, datatypes, and type synonyms. Inside the structure you can use earlier bindings just like we have been doing “at top-level” (i.e., outside of any module). Outside the structure, you refer to a binding b
in Name by writing Name.b. We have already been using this notation to use functions like List.foldl; now you know how to define your own structures.

Though we will not do so in our examples, you can nest structures inside other structures to create a tree-shaped hierarchy. But in ML, modules are not expressions: you cannot define them inside of functions, store them in tuples, pass them as arguments, etc.

If in some scope you are using many bindings from another structure, it can be inconvenient to write SomeLongStructureName.foo many times. Of course, you can use a val-binding to avoid this, e.g.,

val foo = SomeLongStructureName.foo

but this technique is ineffective if we are using many different bindings from the structure (we would need a new variable for each) or for using constructor names from the structure in patterns. So ML allows you to write open SomeLongStructureName, which provides “direct” access (you can just write foo) to any bindings in the module that are mentioned in the module’s signature. The scope of an open is the rest of the enclosing structure (or the rest of the program at top-level).

A common use of open is to write succinct testing code for a module outside the module itself. Other uses of open are often frowned upon because it may introduce unexpected shadowing, especially since different modules may reuse binding names. For example, a list module and a tree module may both have functions named map.

Signatures

So far, structures are providing just namespace management, a way to avoid different bindings in different parts of the program from shadowing each other. Namespace management is very useful, but not very interesting. Much more interesting is giving structures signatures, which are types for modules. They let us provide strict interfaces that code outside the module must obey. ML has several ways to do this with subtly different syntax and semantics; we just show one way to write down an explicit signature for a module. Here is an example signature definition and structure definition that says the structure MyMathLib must have the signature MATHLIB:

signature MATHLIB =
  sig
  val fact : int -> int
  val half_pi : real
  val doubler : int -> int
  end

structure MyMathLib :> MATHLIB =
  struct
  fun fact x =
    if x=0
    then 1
    else x * fact (x - 1)
  val half_pi = Math.pi / 2.0
  fun doubler y = y + y
  end

Because of the :> MATHLIB, the structure MyMathLib will type-check only if it actually provides everything the signature MATHLIB claims it does and with the right types. Signatures can also contain datatype, exception,
and type bindings. Because we check the signature when we compile MyMathLib, we can use this information when we check any code that uses MyMathLib. In other words, we can just check clients assuming that the signature is correct.

**Hiding Things**

Before learning how to use ML modules to hide implementation details from clients, let’s remember that separating an interface from an implementation is probably the most important strategy for building correct, robust, reusable programs. Moreover, we can already use functions to hide implementations in various ways. For example, all 3 of these functions double their argument, and clients (i.e., callers) would have no way to tell if we replaced one of the functions with a different one:

```ml
fun double1 x = x + x
fun double2 x = x * 2
val y = 2
fun double3 x = x * y
```

Another feature we use for hiding implementations is defining functions locally inside other functions. We can later change, remove, or add locally defined functions knowing the old versions were not relied on by any other code. From an engineering perspective, this is a crucial separation of concerns. I can work on improving the implementation of a function and know that I am not breaking any clients. Conversely, nothing clients can do can break how the functions above work.

But what if you wanted to have two top-level functions that code in other modules could use and have both of them use the same hidden functions? There are ways to do this (e.g., create a record of functions), but it would be convenient to have some top-level functions that were “private” to the module. In ML, there is no “private” keyword like in other languages. Instead, you use signatures that simply mention less: anything not explicitly in a signature cannot be used from the outside. For example, if we change the signature above to:

```ml
signature MATHLIB =
  sig
  val fact : int -> int
  val half_pi : real
  end
```

then client code cannot call MyMathLib.doubler. The binding simply is not in scope, so no use of it will type-check. In general, the idea is that we can implement the module however we like and only bindings that are explicitly listed in the signature can be called directly by clients.

**Introducing our extended example**

The rest of our module-system study will use as an example a small module that implements rational numbers. While a real library would provide many more features, ours will just support creating fractions, adding two fractions, and converting fractions to strings. Our library intends to (1) prevent denominators of zero and (2) keep fractions in reduced form (3/2 instead of 9/6 and 4 instead of 4/1). While negative fractions are fine, internally the library never has a negative denominator (−3/2 instead of 3/−2 and 3/2 instead of −3/−2). The structure below implements all these ideas, using the helper function reduce, which itself uses gcd, for reducing a fraction.
Our module maintains invariants, as seen in the comments near the top of the code. These are properties of fractions that all the functions both assume to be true and guarantee to keep true. If one function violates the invariants, other functions might do the wrong thing. For example, the gcd function is incorrect for negative arguments, but because denominators are never negative, gcd is never called with a negative argument.

structure Rational1 =
struct
 (* Invariant 1: all denominators > 0
   Invariant 2: rationals kept in reduced form, including that
     a Frac never has a denominator of 1 *)
datatype rational = Whole of int | Frac of int*int
exception BadFrac

(* gcd and reduce help keep fractions reduced,
   but clients need not know about them *)
(* they _assume_ their inputs are not negative *)
fun gcd (x,y) =
  if x=y
  then x
  else if x < y
  then gcd(x,y-x)
  else gcd(y,x)

fun reduce r =
  case r of
    Whole _ => r
  | Frac(x,y) =>
    if x=0
    then Whole 0
    else let val d = gcd(abs x,y) in (* using invariant 1 *)
      if d=y
      then Whole(x div d)
      else Frac(x div d, y div d)
    end

(* when making a frac, we ban zero denominators *)
fun make_frac (x,y) =
  if y = 0
  then raise BadFrac
  else if y < 0
  then reduce(Frac(~x,~y))
  else reduce(Frac(x,y))

(* using math properties, both invariants hold of the result
   assuming they hold of the arguments *)
fun add (r1,r2) =
  case (r1,r2) of
    (Whole(i),Whole(j)) => Whole(i+j)
  | (Whole(i),Frac(j,k)) => Frac(j+k*i,k)
  | (Frac(j,k),Whole(i)) => Frac(j+k*i,k)
  | (Frac(a,b),Frac(c,d)) => reduce (Frac(a*d + b*c, b*d))
fun toString r = 
    case r of
    Whole i => Int.toString i
  | Frac(a,b) => (Int.toString a) ^ \\
    ^ "/" ^ (Int.toString b)
end

Signatures for Our Example

Let us now try to give our example module a signature such that clients can use it but not violate its invariants.

Since \texttt{reduce} and \texttt{gcd} are helper functions that we do not want clients to rely on or misuse, one natural signature would be as follows:

\begin{verbatim}
signature RATIONAL_A =  
sig
  datatype rational = Frac of int * int | Whole of int
  exception BadFrac
  val make_frac : int * int -> rational
  val add : rational * rational -> rational
  val toString : rational -> string
end
\end{verbatim}

To use this signature to hide \texttt{gcd} and \texttt{reduce}, we can just change the first line of the structure definition above to \texttt{structure Rational1 :> RATIONAL_A}.

While this approach ensures clients do not call \texttt{gcd} or \texttt{reduce} directly (since they “do not exist” outside the module), this is \textit{not} enough to ensure the bindings in the module are used correctly. What “correct” means for a module depends on the specification for the module (not the definition of the ML language), so let’s be more specific about some of the desired \textit{properties} of our library for rational numbers:

- Property: \texttt{toString} always returns a string representation in reduced form
- Property: No code goes into an infinite loop
- Property: No code divides by zero
- Property: There are no fractions with denominators of 0

The properties are \textit{externally visible}; they are what we promise clients. In contrast, the invariants are \textit{internal}; they are facts about the \textit{implementation} that help ensure the properties. The code above maintains the invariants and relies on them in certain places to ensure the properties, notably:

- \texttt{gcd} will violate the properties if called with an arguments \( \leq 0 \), but since we know denominators are \( > 0 \), \texttt{reduce} can pass denominators to \texttt{gcd} without concern.
- \texttt{toString} and most cases of \texttt{add} do not need to call \texttt{reduce} because they can assume their arguments are already in reduced form.
• **add** uses the property of mathematics that the product of two positive numbers is positive, so we know a non-positive denominator is not introduced.

Unfortunately, under signature **RATIONAL_A**, clients must still be trusted not to break the properties and invariants! Because the signature exposed the definition of the datatype binding, the ML type system will not prevent clients from using the constructors **Frac** and **Whole** directly, bypassing all our work to establish and preserve the invariants. Clients could make “bad” fractions like `Rational.Frac(1,0), Rational.Frac(3,~2),` or `Rational.Frac(9,6)`, any of which could then end up causing **gcd** or **toString** to misbehave according to our specification. While we may have intended for the client only to use **make_frac**, **add**, and **toString**, our signature allows more.

A natural reaction would be to hide the datatype binding by removing the line
datatype rational = Frac of int * int | Whole of int. While this is the right intuition, the resulting signature makes no sense and would be rejected: it repeatedly mentions a type **rational** that is not known to exist. What we want to say instead is that there is a type **rational** but clients cannot know anything about what the type is other than it exists. In a signature, we can do just that with an **abstract type**, as this signature shows:

```ml
signature RATIONAL_B =
  sig
  type rational (* type now abstract *)
  exception BadFrac
  val make_frac : int * int -> rational
  val add : rational * rational -> rational
  val toString : rational -> string
  end
```

(Of course, we also have to change the first line of the structure definition to use this signature instead. That is always true, so we will stop mentioning it.)

This new feature of abstract types, which makes sense only in signatures, is exactly what we want. It lets our module define operations over a type without revealing the implementation of that type. The syntax is just to give a type binding without a definition. The implementation of the module is unchanged; we are simply changing how much information clients have.

Now, how can clients make rationals? Well, the first one will have to be made with **make_frac**. After that, more rationals can be made with **make_frac** or **add**. There is no other way, so thanks to the way we wrote **make_frac** and **add**, all rationals will always be in reduced form with a positive denominator.

What **RATIONAL_B** took away from clients compared to **RATIONAL_A** is the constructors **Frac** and **Whole**. So clients cannot create rationals directly and they cannot pattern-match on rationals. They have no idea how they are represented internally. They do not even know **rational** is implemented as a datatype.

Abstract types are a Really Big Deal in programming.

**A Cute Twist: Expose the Whole function**

By making the **rational** type abstract, we took away from clients the **Frac** and **Whole** constructors. While this was crucial for ensuring clients could not create a fraction that was not reduced or had a non-positive denominator, only the **Frac** constructor was problematic. Since allowing clients to create whole numbers directly cannot violate our specification, we could add a function like:

```ml
fun make_whole x = Whole x
```
to our structure and val make_whole : int -> rational to our signature. But this is unnecessary function wrapping; a shorter implementation would be:

val make_whole = Whole

and of course clients cannot tell which implementation of make_whole we are using. But why create a new

binding make_whole that is just the same thing as Whole? Instead, we could just export the constructor as a function with this signature and no changes or additions to our structure:

signature RATIONAL_C =
  sig
  type rational (* type still abstract *)
  exception BadFrac
  val Whole : int -> rational (* client knows only that Whole is a function *)
  val make_frac : int * int -> rational
  val add : rational * rational -> rational
  val toString : rational -> string
  end

This signature tells clients there is a function bound to Whole that takes an int and produces a rational. That is correct: this binding is one of the things the datatype binding in the structure creates. So we are exposing part of what the datatype binding provides: that rational is a type and that Whole is bound to a function. We are still hiding the rest of what the datatype binding provides: the Frac constructor and pattern-matching with Frac and Whole.

Rules for Signature Matching

So far, our discussion of whether a structure “should type-check” given a particular signature has been rather informal. Let us now enumerate more precise rules for what it means for a structure to match a signature. (This terminology has nothing to do with pattern-matching.) If a structure does not match a signature assigned to it, then the module does not type-check. A structure Name matches a signature BLAH if:

- For every val-binding in BLAH, Name must have a binding with that type or a more general type (e.g., the implementation can be polymorphic even if the signature says it is not — see below for an example). This binding could be provided via a val-binding, a fun-binding, or a datatype-binding.
- For every non-abstract type-binding in BLAH, Name must have the same type binding.
- For every abstract type-binding in BLAH, Name must have some binding that creates that type (either a datatype binding or a type synonym).

Notice that Name can have any additional bindings that are not in the signature.

Equivalent Implementations

Given our property- and invariant-preserving signatures RATIONAL_B and RATIONAL_C, we know clients cannot rely on any helper functions or the actual representation of rationals as defined in the module. So we could replace the implementation with any equivalent implementation that had the same properties: as long as
any call to the `toString` binding in the module produced the same result, clients could never tell. This is another essential software-development task: improving/changing a library in a way that does not break clients. Knowing clients obey an abstraction boundary, as enforced by ML’s signatures, is invaluable.

As a simple example, we could make `gcd` a local function defined inside of `reduce` and know that no client will fail to work since they could not rely on `gcd`’s existence. More interestingly, let’s change one of the invariants of our structure. Let’s *not* keep rationals in reduced form. Instead, let’s just reduce a rational right before we convert it to a string. This simplifies `make_frac` and `add`, while complicating `toString`, which is now the only function that needs `reduce`. Here is the whole structure, which would still match signatures `RATIONAL_A`, `RATIONAL_B`, or `RATIONAL_C`:

```
structure Rational2 :> RATIONAL_A (* or B or C *) =
  struct
    datatype rational = Whole of int | Frac of int*int
    exception BadFrac

    fun make_frac (x,y) =
      if y = 0
        then raise BadFrac
      else if y < 0
        then Frac(~x,~y)
      else Frac(x,y)

    fun add (r1,r2) =
      case (r1,r2) of
        (Whole(i),Whole(j)) => Whole(i+j)
      | (Whole(i),Frac(j,k)) => Frac(j+k*i,k)
      | (Frac(j,k),Whole(i)) => Frac(j+k*i,k)
      | (Frac(a,b),Frac(c,d)) => Frac(a*d + b*c, b*d)

    fun toString r =
      let fun gcd (x,y) =
        if x=y
          then x
        else if x < y
          then gcd(x,y-x)
        else gcd(y,x)
      in
        case reduce r of
          Whole _ => r
        | Frac(x,y) =>
          if x=0
            then Whole 0
          else let val d = gcd(abs x,y) in
            if d=y
              then Whole(x div d)
            else Frac(x div d, y div d)
            end
          end
      end
  end
```
Whole i => Int.toString i
  | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)
end

eend

If we give Rational1 and Rational2 the signature RATIONAL_A, both will type-check, but clients can still distinguish them. For example, Rational1.toString(Rational1.Frac(21,3)) produces "21/3", but Rational2.toString(Rational2.Frac(21,3)) produces "7". But if we give Rational1 and Rational2 the signature RATIONAL_B or RATIONAL_C, then the structures are equivalent for any possible client. This is why it is important to use restrictive signatures like RATIONAL_B to begin with: so you can change the structure later without checking all the clients.

While our two structures so far maintain different invariants, they do use the same definition for the type rational. This is not necessary with signatures RATIONAL_B or RATIONAL_C; a different structure having these signatures could implement the type differently. For example, suppose we realize that special-casing whole-numbers internally is more trouble than it is worth. We could instead just use int*int and define this structure:

```
structure Rational3 :> RATIONAL_B (* or C *) =
struct
  type rational = int*int
  exception BadFrac

  fun make_frac (x,y) =
    if y = 0
      then raise BadFrac
    else if y < 0
      then (~x,~y)
    else (x,y)

  fun Whole i = (i,1)

  fun add ((a,b),(c,d)) = (a*d + c*b, b*d)

  fun toString (x,y) =
    if x=0
      then "0"
    else
      let fun gcd (x,y) =
        if x=y
          then x
        else if x < y
          then gcd(x,y-x)
        else gcd(y,x)
        val d = gcd (abs x,y)
        val num = x div d
        val denom = y div d
      in
        Int.toString num ^ (if denom=1
          then ""
        else "/" ^ (Int.toString denom))
      end

end
```
(This structure takes the Rational2 approach of having toString reduce fractions, but that issue is largely orthogonal from the definition of rational.)

Notice that this structure provides everything RATIONAL_B requires. The function make_frac is interesting in that it takes an int*int and return an int*int, but clients do not know the actual return type, only the abstract type rational. And while giving it an argument type of rational in the signature would match, it would make the module useless since clients would not be able to create a value of type rational. Nonetheless, clients cannot pass just any int*int to add or toString; they must pass something that they know has type rational. As with our other structures, that means rationals are created only by make_frac and add, which enforces all our invariants.

Our structure does not match RATIONAL_A since it does not provide rational as a datatype with constructors Frac and Whole.

Our structure does match signature RATIONAL_C because we explicitly added a function Whole of the right type. No client can distinguish our “real function” from the previous structures’ use of the Whole constructor as a function.

The fact that fun Whole i = (i,1) matches val Whole : int -> rational is interesting. The type of Whole in the module is actually polymorphic: 'a -> 'a * int. ML signature matching allows 'a -> 'a * int to match int -> rational because 'a -> 'a * int is more general than int -> int * int and int -> rational is a correct abstraction of int -> int * int. Less formally, the fact that Whole has a polymorphic type inside the module does not mean the signature has to give it a polymorphic type outside the module. And in fact it cannot while using the abstract type since Whole cannot have the type 'a -> int * int or 'a -> rational.

Different modules define different types

While we have defined different structures (e.g., Rational1, Rational2, and Rational3) with the same signature (e.g., RATIONAL_B), that does not mean that the bindings from the different structures can be used with each other. For example, Rational1.toString(Rational2.make_frac(2,3)) will not type-check, which is a good thing since it would print an unreduced fraction. The reason it does not type-check is that Rational2.rational and Rational1.rational are different types. They were not created by the same datatype binding even though they happen to look identical. Moreover, outside the module we do not know they look identical. Indeed, Rational3.toString(Rational2.make_frac(2,3)) really needs not to type-check since Rational3.toString expects an int*int but Rational2.make_frac(2,3) returns a value made out of the Rational2.Frac constructor.

What is Type Inference?

While we have been using ML type inference for a while now, we have not studied it carefully. We will first carefully define what type inference is and then see via several examples how ML type inference works.

Java, C, and ML are all examples of statically typed languages, meaning every binding has a type that is determined “at compile-time,” i.e., before any part of the program is run. The type-checker is a compile-time procedure that either accepts or rejects a program. By contrast, Racket, Ruby, and Python are dynamically typed languages, meaning the type of a binding is not determined ahead of time and computations like binding 42 to x and then treating x as a string result in run-time errors. After we do some programming with Racket, we will compare the advantages and disadvantages of static versus dynamic typing as a significant course topic.
Unlike Java and C, ML is *implicitly typed*, meaning programmers rarely need to write down the types of bindings. This is often convenient (though some disagree as to whether it makes code easier or harder to read), but in no way changes the fact that ML is statically typed. Rather, the type-checker has to be more sophisticated because it must *infer* (i.e., figure out) what the *type annotations* “would have been” had programmers written all of them. In principle, type inference and type checking could be separate steps (the inferrer could do its part and the checker could see if the result should type-check), but in practice they are often merged into “the type-checker.” Note that a correct type-inferencer must find a solution to what all the types should be whenever such a solution exists, else it must reject the program.

Whether type inference for a particular programming language is easy, difficult, or impossible is often not obvious. It is *not* proportional to how permissive the type system is. For example, the “extreme” type systems that “accept everything” and “accept nothing” are both very easy to do inference for. When we say type inference may be impossible, we mean this in the technical sense of undecidability, like the famous halting problem. We mean there are type systems for which no computer program can implement type inference such that (1) the inference process always terminates, (2) the inference process always succeeds if inference is possible, and (3) the inference process always fails if inference is not possible.

Fortunately, ML was rather cleverly designed so that type inference can be performed by a fairly straightforward and elegant algorithm. While there are programs for which inference is intractably slow, programs people write in practice never cause such behavior. We will demonstrate key aspects of the algorithm for ML type inference with a few examples. This will give you a sense that type inference is not “magic.” In order to move on to other course topics, we will not describe the full algorithm or write code to implement it.

ML type inference ends up intertwined with parametric polymorphism — when the inferencer determines a function’s argument or result “could be anything” the resulting type uses `{a}, {b}, etc. But type inference and polymorphism are entirely separate concepts: a language could have one or the other. For example, Java has generics but no inference for method argument/result types.

**Overview of ML Type Inference**

Here is an overview of how ML type inference works (more examples to follow):

- It determines the types of bindings in order, using the types of earlier bindings to infer the types of later ones. This is why you cannot use later bindings in a file. (When you need to, you use mutual recursion and type inference determines the types of all the mutually recursive bindings together. Mutual recursion is covered later in this unit.)

- For each `val` or `fun` binding, it analyzes the binding to determine necessary facts about its type. For example, if we see the expression `x+1`, we conclude that `x` must have type `int`. We gather similar facts for function calls, pattern-matches, etc.

- Afterward, use *type variables* (e.g., `{a}) for any unconstrained types in function arguments or results.

- (Enforce the value restriction — only variables and values can have polymorphic types, as discussed later.)

The amazing fact about the ML type system is that “going in order” this way never causes us to reject a program that could type-check nor do we ever accept a program we should not. So explicit type annotations really are optional unless you use features like `{1}. (The problem with `{1} is that it does not give enough information for type inference to know what other fields the tuple/record should have, and the ML type system requires knowing the exact number of fields and all the fields’ names.)
Here is an initial, very simple example:

\[
\begin{align*}
\text{val } x &= 42 \\
\text{fun } f(y,z,w) &= \text{if } y \text{ then } z+x \text{ else } 0
\end{align*}
\]

Type inference first gives \(x\) type \texttt{int} since 42 has type \texttt{int}. Then it moves on to infer the type for \(f\). Next we will study, via other examples, a more step-by-step procedure, but here let us just list the key facts:

- \(y\) must have type \texttt{bool} because we test it in a conditional.
- \(z\) must have type \texttt{int} because we add it to something we already determined has type \texttt{int}.
- \(w\) can have any type because it is never used.
- \(f\) must return an \texttt{int} because its body is a conditional where both branches return an \texttt{int}. (If they disagreed, type-checking would fail.)

So the type of \(f\) must be \texttt{bool * int * 'a -> int}.

**More Thorough Examples of ML Type Inference**

We will now work through a few examples step-by-step, generating all the facts that the type-inference algorithm needs. Note that humans doing type inference “in their head” often take shortcuts just like humans doing arithmetic in their head, but the point is there is a general algorithm that methodically goes through the code gathering constraints and putting them together to get the answer.

As a first example, consider inferring the type for this function:

\[
\begin{align*}
\text{fun } f \ x &= \\
\text{let } \text{val } (y,z) &= x \text{ in} \\
\quad (\text{abs } y) + z \\
\text{end}
\end{align*}
\]

Here is how we can infer the type:

- Looking at the first line, \(f\) must have type \(T_1\rightarrow T_2\) for some types \(T_1\) and \(T_2\) and in the function body \(f\) has this type and \(x\) has type \(T_1\).
- Looking at the \texttt{val}-binding, \(x\) must be a pair type (else the pattern-match makes no sense), so in fact \(T_1=T_3\times T_4\) for some \(T_3\) and \(T_4\), and \(y\) has type \(T_3\) and \(z\) has type \(T_4\).
- Looking at the addition expression, we know from the context that \texttt{abs} has type \texttt{int->int}, so \(y\) having type \(T_3\) means \(T_3=\text{int}\). Similarly, since \texttt{abs} \(y\) has type \texttt{int}, the other argument to + must have type \texttt{int}, so \(z\) having type \(T_4\) means \(T_4=\text{int}\).
- Since the type of the addition expression is \texttt{int}, the type of the let-expression is \texttt{int}. And since the type of the let-expression is \texttt{int}, the return type of \(f\) is \texttt{int}, i.e., \(T_2=\text{int}\).

Putting all these constraints together, \(T_1=\text{int*int}\) (since \(T_1=T_3\times T_4\)) and \(T_2=\text{int}\), so \(f\) has type \texttt{int*int->int}.

Next example:
fun sum xs = 
case xs of 
  [] => 0 
| x::xs' => x + (sum xs')

- From the first line, there exists types $T_1$ and $T_2$ such that $\text{sum}$ has type $T_1 \rightarrow T_2$ and $xs$ has type $T_1$.
- Looking at the case-expression, $xs$ must have a type that is compatible with all of the patterns. Looking at the patterns, both of them match any list, since they are built from list constructors (in the $x::xs'$ case the subpatterns match anything of any type). So since $xs$ has type $T_1$, in fact $T_1 = T_3 \text{ list}$ from some type $T_3$.
- Looking at the right-hand sides of the case branches, we know they must have the same type as each other and this type is $T_2$. Since 0 has type $\text{int}$, $T_2 = \text{int}$.
- Looking at the second case branch, we type-check it in a context where $x$ and $xs'$ are available. Since we are matching the pattern $x::xs'$ against a $T_3 \text{ list}$, it must be that $x$ has type $T_3$ and $xs'$ has type $T_3 \text{ list}$.
- Now looking at the right-hand side, we add $x$, so in fact $T_3 = \text{int}$. Moreover, the recursive call type-checks because $xs'$ has type $T_3 \text{ list}$ and $T_3 \text{ list} = T_1$ and $\text{sum}$ has type $T_1 \rightarrow T_2$. Finally, since $T_2 = \text{int}$, adding $\text{sum} \ xs'$ type-checks.

Putting everything together, we get $\text{sum}$ has type $\text{int list} \rightarrow \text{int}$.

Notice that before we got to $\text{sum} \ xs'$ we had already inferred everything, but we still have to check that types are used consistently and reject otherwise. For example, if we had written $\text{sum} \ x$, that cannot type-check — it is inconsistent with previous facts. Let us see this more thoroughly to see what happens:

fun broken_sum xs = 
case xs of 
  [] => 0 
| x::xs' => x + (broken_sum x)

- Type inference for $\text{broken_sum}$ proceeds largely the same as for $\text{sum}$. The first four bullets from the previous example all apply, giving $\text{broken_sum}$ type $T_3 \text{ list} \rightarrow \text{int}$, $x3$ type $T_3 \text{ list}$, $x$ type $T_3$, and $xs'$ type $T_3 \text{ list}$. Moreover, $T_3 = \text{int}$.
- We depart from the correct $\text{sum}$ implementation with the call $\text{broken_sum} \ x$. For this call to type-check, $x$ must have the same type as $\text{broken_sum}$’s parameter, or in other words, $T_1 = T_3$. However, we know that $T_1 = T_3 \text{ list}$, so this new constraint $T_1 = T_3$ actually generates a contradiction: $T_3 = T_3 \text{ list}$. If we want to be more concrete, we can use our knowledge that $T_3 = \text{int}$ to rewrite this as $\text{int} = \text{int list}$. Looking at the definition of $\text{broken_sum}$ it should be obvious that this is exactly the problem: we tried to use $x$ as an $\text{int}$ and as an $\text{int list}$.

When your ML program does not type-check, the type-checker reports the expression where it discovered a contradiction and what types were involved in that contradiction. While sometimes this information is helpful, other times the actual problem is with a different expression, but the type-checker did not reach a contradiction until later.
Examples with Polymorphic Types

Our remaining examples will infer polymorphic types. All we do is follow the same procedure we did above, but when we are done, we will have some parts of the function’s type that are still *unconstrained*. For each \( Ti \) that “can be anything” we use a type variable (‘a, ’b, etc.).

```haskell
fun length xs =
  case xs of
    [] => 0
  | x::xs' => 1 + (length xs')
```

Type inference proceeds much like with *sum*. We end up determining:

- \( \text{length} \) has type \( T1 \rightarrow T2 \).
- \( \text{x} \) has type \( T1 \).
- \( T1=T3 \ list \) (due to the pattern-match)
- \( T2=\text{int} \) because \( 0 \) can be the result of a call to \( \text{length} \).
- \( \text{x} \) has type \( T3 \) and \( \text{xs'} \) has type \( T3 \ list \).
- The recursive call \( \text{length} \ \text{xs'} \) type-checks because \( \text{xs'} \) has type \( T3 \ list \), which is \( T1 \), the argument type of \( \text{length} \). And we can add the result because \( T2=\text{int} \).

So we have all the same constraints as for *sum*, except we do not have \( T3=\text{int} \). In fact, \( T3 \) can be anything and \( \text{length} \) will type-check. So type inference recognizes that when it is all done, it has \( \text{length} \) with type \( T3 \ list \rightarrow \text{int} \) and \( T3 \) can be anything. So we end up with the type \( 'a \ list \rightarrow \text{int} \), as expected. Again the rule is simple: for each \( Ti \) in the final result that cannot be constrained, use a type variable.

A second example:

```haskell
fun compose (f,g) = fn x => f (g x)
```

- Since the argument to \( \text{compose} \) must be a pair (from the pattern used for its argument), \( \text{compose} \) has type \( T1*T2 \rightarrow T3 \), \( f \) has type \( T1 \) and \( g \) has type \( T2 \).
- Since \( \text{compose} \) returns a function, \( T3 \) is some \( T4 \rightarrow T5 \) where in that function’s body, \( x \) has type \( T4 \).
- So \( g \) must have type \( T4 \rightarrow T6 \) for some \( T6 \), i.e., \( T2=T4 \rightarrow T6 \).
- And \( f \) must have type \( T6 \rightarrow T7 \) for some \( T7 \), i.e., \( T1=T6 \rightarrow T7 \).
- But the result of \( f \) is the result of the function returned by \( \text{compose} \), so \( T7=T5 \) and so \( T1=T6 \rightarrow T5 \).

Putting together \( T1=T6 \rightarrow T5 \) and \( T2=T4 \rightarrow T6 \) and \( T3=T4 \rightarrow T5 \) we have a type for \( \text{compose} \) of \( (T6 \rightarrow T5)*(T4 \rightarrow T6) \rightarrow (T4 \rightarrow T5) \). There is nothing else to constrain the types \( T4 \), \( T5 \), and \( T6 \), so we replace them consistently to end up with \( ('a \rightarrow 'b)*(c \rightarrow 'a) \rightarrow ('c \rightarrow 'b) \) as expected (and the last set of parentheses are optional, but that is just syntax).

Here is a simpler example that also has multiple type variables:
fun f (x,y,z) = 
  if true
  then (x,y,z)
  else (y,x,z)

• The first line requires that \( f \) has type \( T_1*T_2*T_3 \to T_4 \), \( x \) has type \( T_1 \), \( y \) has type \( T_2 \), and \( z \) has type \( T_3 \).

• The two branches of the conditional must have the same type and this is the return type of the function \( T_4 \). Therefore, \( T_4=T_1*T_2*T_3 \) and \( T_4=T_2*T_1*T_3 \). This constraint requires \( T_1=T_2 \).

Putting together these constraints (and no others), \( f \) will type-check with type \( T_1*T_1*T_3 \to T_1*T_1*T_3 \) for any types \( T_1 \) and \( T_3 \). So replacing each type consistently with a type variable, we get \( 'a*'a*'b \to 'a*'a*'b \), which is correct: \( x \) and \( y \) must have the same type, but \( z \) can (but need not) have a different type. Notice that the type-checker always requires both branches of a conditional to type-check with the same type, even though here we know which branch will be evaluated.

Optional: The Value Restriction

As described so far in this unit, the ML type system is unsound, meaning that it would accept programs that when run could have values of the wrong types, such as putting an \( \text{int} \) where we expect a \( \text{string} \). The problem results from a combination of polymorphic types and mutable references, and the fix is a special restriction to the type system called \textit{the value restriction}.

This is an example program that demonstrates the problem:

```ml
val r = ref NONE (* 'a option ref *)
val _ = r := SOME "hi" (* instantiate 'a with string *)
val i = 1 + valOf(!r) (* instantiate 'a with int *)
```

Straightforward use of the rules for type checking/inference would accept this program even though we should not – we end up trying to add 1 to "hi". Yet everything seems to type-check given the types for the functions/operators \( \text{ref ('a -> 'a ref), := ('a ref * 'a -> unit), and ! ('a ref -> 'a)} \).

To restore soundness, we need a stricter type system that does not let this program type-check. The choice ML made is to prevent the first line from having a polymorphic type. Therefore, the second and third lines will not type-check because they will not be able to instantiate an \( 'a \) with \text{string} \) or \text{int}. In general, ML will give a variable in a val-binding a polymorphic type only if the expression in the val-binding is a value or a variable. This is called the value restriction. In our example, \( \text{ref NONE} \) is a call to the function \( \text{ref} \). Function calls are not variables or values. So we get a warning and \( r \) is given a type \( ?X1 \) \text{option ref} where \( ?X1 \) is a “dummy type,” not a type variable. This makes \( r \) not useful and the later lines do not type-check. It is not at all obvious that this restriction suffices to make the type system sound, but in fact it is sufficient.

For \( r \) above, we can use the expression \( \text{ref NONE} \), but we have to use a type annotation to give \( r \) a non-polymorphic type, such as \text{int option ref}. Whatever we pick, one of the next two lines will not type-check.

As we saw previously when studying partial application, the value restriction is occasionally burdensome even when it is not a problem because we are not using mutation. We saw that this binding falls victim to the value-restriction and is not made polymorphic:

```ml
val pairWithOne = List.map (fn x => (x,1))
```
We saw multiple workarounds. One is to use a function binding, even though without the value restriction it would be unnecessary function wrapping. This function has the desired type `'a list -> ('a * int) list:

```haskell
fun pairWithOne xs = List.map (fn x => (x,1)) xs
```

One might wonder why we cannot enforce the value restriction only for references (where we need it) and not for immutable types like lists. The answer is the ML type-checker cannot always know which types are really references and which are not. In the code below, we need to enforce the value restriction on the last line, because `'a foo` and `'a ref` are the same type.

```haskell
type `'a foo = `'a ref
val f : `'a -> `'a foo = ref
val r = f NONE
```

Because of ML’s module system, the type-checker does not always know the definition of type synonyms (recall this is a good thing). So to be safe, it enforces the value restriction for all types.

**Optional: Some Things that Make Type Inference More Difficult**

Now that we have seen how ML type inference works, we can make two interesting observations:

- Inference would be more difficult if ML had subtyping (e.g., if every triple could also be a pair) because we would not be able to conclude things like, “T3=T1*T2” since the equals would be overly restrictive. We would instead need constraints indicating that T3 is a tuple with at least two fields. Depending on various details, this can be done, but type inference is more difficult and the results are more difficult to understand.

- Inference would be more difficult if ML did not have parametric polymorphism since we would have to pick some type for functions like `length` and `compose` and that could depend on how they are used.

**Mutual Recursion**

We have seen many examples of recursive functions and many examples of functions using other functions as helper functions, but what if we need a function f to call g and g to call f? That can certainly be useful, but ML’s rule that bindings can only use earlier bindings makes it more difficult — which should come first, f or g?

It turns out ML has special support for mutual recursion using the keyword and putting the mutually recursive functions next to each other. Similarly, we can have mutually recursive `datatype` bindings. After showing these new constructs, we will show that you can actually work around a lack of support for mutually recursive functions by using higher-order functions, which is a useful trick in general and in particular in ML if you do not want your mutually recursive functions next to each other.

Our first example uses mutual recursion to process an `int list` and return a `bool`. It returns true if the list strictly alternates between 1 and 2 and ends with a 2. Of course there are many ways to implement such a function, but our approach does a nice job of having for each “state” (such as “a 1 must come next” or “a 2 must come next”) a function. In general, many problems in computer science can be modeled by such
finite state machines, and mutually recursive functions, one for each state, are an elegant way to implement finite state machines.

fun match xs = 
  let fun s_need_one xs = 
    case xs of 
      [] => true 
    | 1::xs' => s_need_two xs' 
    | _ => false 
  and s_need_two xs = 
    case xs of 
      [] => false 
    | 2::xs' => s_need_one xs' 
    | _ => false 
  in 
    s_need_one xs 
  end 

(The code uses integer constants in patterns, which is an occasionally convenient ML feature, but not essential to the example.)

In terms of syntax, we define mutually recursive functions by simply replacing the keyword fun for all functions except the first with and. The type-checker will type-check all the functions (two in the example above) together, allowing calls among them regardless of order.

Here is a second (silly) example that also uses two mutually recursive datatype bindings. The definition of types t1 and t2 refer to each other, which is allowed by using and in place of datatype for the second one. This defines two new datatypes, t1 and t2.

datatype t1 = Foo of int | Bar of t2 
and t2 = Baz of string | Quux of t1 

fun no_zeros_or_empty_strings_t1 x = 
  case x of 
    Foo i => i <> 0 
  | Bar y => no_zeros_or_empty_strings_t2 y 
and no_zeros_or_empty_strings_t2 x = 
  case x of 
    Baz s => size s > 0 
  | Quux y => no_zeros_or_empty_strings_t1 y 

Now suppose we wanted to implement the “no zeros or empty strings” functionality of the code above but for some reason we did not want to place the functions next to each other or we were in a language with no support for mutually recursive functions. We can write almost the same code by having the “later” function pass itself to a version of the “earlier” function that takes a function as an argument:

fun no_zeros_or_empty_strings_t1(f,x) = 
  case x of 
    Foo i => i <> 0 
  | Bar y => f y 

1Because all function calls are tail calls, the code runs in a small amount of space, just as one would expect for an implementation of a finite state machine.
fun no_zeros_or_empty_string_t2 x = case x of
  Baz s => size s > 0
| Quux y => no_zeros_or_empty_strings_t1(no_zeros_or_empty_string_t2,y)

This is yet-another powerful idiom allowed by functions taking functions.

Motivating and Defining Equivalence

The idea that one piece of code is “equivalent” to another piece of code is fundamental to programming and computer science. You are informally thinking about equivalence every time you simplify some code or say, “here’s another way to do the same thing.” This kind of reasoning comes up in several common scenarios:

- Code maintenance: Can you simplify, clean up, or reorganize code without changing how the rest of the program behaves?
- Backward compatibility: Can you add new features without changing how any of the existing features work?
- Optimization: Can you replace code with a faster or more space-efficient implementation?
- Abstraction: Can an external client tell if I make this change to my code?

Also notice that our use of restrictive signatures in the previous lecture was largely about equivalence: by using a stricter interface, we make more different implementations equivalent because clients cannot tell the difference.

We want a precise definition of equivalence so that we can decide whether certain forms of code maintenance or different implementations of signatures are actually okay. We do not want the definition to be so strict that we cannot make changes to improve code, but we do not want the definition to be so lenient that replacing one function with an “equivalent” one can lead to our program producing a different answer. Hopefully, studying the concepts and theory of equivalence will improve the way you look at software written in any language.

There are many different possible definitions that resolve this strict/lenient tension slightly differently. We will focus on one that is useful and commonly assumed by people who design and implement programming languages. We will also simplify the discussion by assuming that we have two implementations of a function and we want to know if they are equivalent.

The intuition behind our definition is as follows:

- A function \( f \) is equivalent to a function \( g \) (or similarly for other pieces of code) if they produce the same answer and have the same side-effects no matter where they are called in any program with any arguments.

- Equivalence does not require the same running time, the same use of internal data structures, the same helper functions, etc. All these things are considered “unobservable”, i.e., implementation details that do not affect equivalence.

As an example, consider two very different ways of sorting a list. Provided they both produce the same final answer for all inputs, they can still be equivalent no matter how they worked internally or whether one was
faster. However, if they behave differently for some lists, perhaps for lists that have repeated elements, then they would not be equivalent.

However, the discussion above was simplified by implicitly assuming the functions always return and have no other effect besides producing their answer. To be more precise, we need that the two functions when given the same argument in the same environment:

1. Produce the same result (if they produce a result)
2. Have the same (non)termination behavior; i.e., if one runs forever the other must run forever
3. Mutate the same (visible-to-clients) memory in the same way.
4. Do the same input/output
5. Raise the same exceptions

These requirements are all important for knowing that if we have two equivalent functions, we could replace one with the other and no use anywhere in the program will behave differently.

**Another Benefit of Side-Effect-Free Programming**

One easy way to make sure two functions have the same side effects (mutating references, doing input/output, etc.) is to have no side effects at all. This is exactly what functional languages like ML encourage. Yes, in ML you *could* have a function body mutate some global reference or something, but it is generally bad style. Other functional languages are *pure functional languages* meaning there really is no way to do mutation inside (most) functions.

*If* you “stay functional” by not doing mutation, printing, etc. in function bodies as a matter of policy, then callers can assume lots of equivalences they cannot otherwise. For example, can we replace \( (f \ x) + (f \ x) \) with \( (f \ x) \times 2 \)? In general, that can be a wrong thing to do since calling \( f \) might update some counter or print something. In ML, that’s also possible, but far less likely as a matter of style, so we tend to have more things be equivalent. In a purely functional language, we are guaranteed the replacement does not change anything. The general point is that mutation really gets in your way when you try to decide if two pieces of code are equivalent — it is a great reason to avoid mutation.

In addition to being able to remove repeated computations (like \( (f \ x) \) above) when maintaining side-effect-free programs, we can also reorder expressions much more freely. For example, in Java, C, etc.:

```c
int a = f(x);
int b = g(y);
return b - a;
```

might produce a different result from:

```c
return g(y) - f(x);
```

since \( f \) and \( g \) can get called in a different order. Again, this is possible in ML too, but if we avoid side-effects, it is much less likely to matter. (We might still have to worry about a different exception getting thrown and other details, however.)
### Standard Equivalences

Equivalence is subtle, especially when you are trying to decide if two functions are equivalent without knowing all the places they may be called. Yet this is common, such as when you are writing a library that unknown clients may use. We now consider several situations where equivalence is guaranteed in any situation, so these are good rules of thumb and are good reminders of how functions and closures work.

First, recall the various forms of syntactic sugar we have learned. We can always use or not use syntactic sugar in a function body and get an equivalent function. If we couldn’t, then the construct we are using is not actually syntactic sugar. For example, these definitions of \( f \) are equivalent regardless of what \( g \) is bound to:

\[
\begin{align*}
\text{fun } f \ x &= \text{fun } f \ x = \\
&\begin{cases} 
\text{if } x \ \text{andalso } g \ x \\
\text{true} \\
\end{cases} \\
\text{else } \text{false}
\end{align*}
\]

Notice though, that we could not necessarily replace \( x \ \text{andalso } g \ x \) with \( \text{if } g \ x \ \text{then } x \ \text{else } \text{false} \) if \( g \) could have side effects or not terminate.

Second, we can change the name of a local variable (or function parameter) provided we change all uses of it consistently. For example, these two definitions of \( f \) are equivalent:

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } f \ x &= x+y+x \\
\text{fun } f \ z &= x+y+z
\end{align*}
\]

But there is one rule: in choosing a new variable name, you cannot choose a variable that the function body is already using to refer to something else. For example, if we try to replace \( x \) with \( y \), we get \( \text{fun } y = y+y+y \), which is not the same as the function we started with. A previously-unused variable is never a problem.

Third, we can use or not use a helper function. For example, these two definitions of \( g \) are equivalent:

\[
\begin{align*}
\text{val } y &= 14 \\
\text{fun } g \ z &= (z+y+z)+z \\
\text{fun } f \ x &= x+y+x \\
\text{fun } g \ z &= (f \ z)+z
\end{align*}
\]

Again, we must take care not to change the meaning of a variable due to \( f \) and \( g \) having potentially different environments. For example, here the definitions of \( g \) are not equivalent:

\[
\begin{align*}
\text{val } y &= 14 \\
\text{val } y &= 7 \\
\text{fun } g \ z &= (z+y+z)+z \\
\text{val } y &= 7 \\
\text{fun } g \ z &= (f \ z)+z
\end{align*}
\]

Fourth, as we have explained before with anonymous functions, unnecessary function wrapping is poor style because there is a simpler equivalent way. For example, \( \text{fun } g \ y = f \ y \) and \( \text{val } g = f \) are always equivalent. Yet once again, there is a subtle complication. While this works when we have a variable like \( f \) bound to the function we are calling, in the more general case we might have an expression that evaluates to a function that we then call. Are \( \text{fun } g \ y = e \ y \) and \( \text{val } g = e \) always the same for any expression \( e \)? No.

As a silly example, consider \( \text{fun } h() \ (\text{print } \"hi\" ; \text{fn } x = x+x) \) and \( e \) is \( h() \). Then \( \text{fun } g \ y = (h()) \ y \) is a function that prints every time it is called. But \( \text{val } g = h() \) is a function that does not print — the
program will print "hi" once when creating the binding for \(g\). This should not be mysterious: we know that
val-bindings evaluate their right-hand sides “immediately” but function bodies are not evaluated until they are called.

A less silly example might be if \(h\) might raise an exception rather than returning a function.

Fifth, it is almost the case that let val \(p = e_1\) in \(e_2\) end can be sugar for \((fn\ p => e_2)\ e_1\). After all, for any expressions \(e_1\) and \(e_2\) and pattern \(p\), both pieces of code:

- Evaluate \(e_1\) to a value
- Match the value against the pattern \(p\)
- If it matches, evaluate \(e_2\) to a value in the environment extended by the pattern match
- Return the result of evaluating \(e_2\)

Since the two pieces of code “do” the exact same thing, they must be equivalent. In Racket, this will be the case (with different syntax). In ML, the only difference is the type-checker: The variables in \(p\) are allowed to have polymorphic types in the let-version, but not in the anonymous-function version.

For example, consider let val \(x = (fn\ y => y)\) in \((x\ 0, x\ true)\) end. This silly code type-checks and returns \((0, true)\) because \(x\) has type \('a->'a\). But \((fn\ x => (x\ 0, x\ true))\) \((fn\ y => y)\) does not type-check because there is no non-polymorphic type we can give to \(x\) and function-arguments cannot have polymorphic types. This is just how type-inference works in ML.

**Revisiting our Definition of Equivalence**

By design, our definition of equivalence ignores how much time or space a function takes to evaluate. So two functions that always returned the same answer could be equivalent even if one took a nanosecond and another took a million years. In some sense, this is a *good thing* since the definition would allow us to replace the million-year version with the nanosecond version.

But clearly other definitions matter too. Courses in data structures and algorithms study asymptotic complexity precisely so that they can distinguish some algorithms as “better” (which clearly implies some “difference”) even though the better algorithms are producing the same answers. Moreover, asymptotic complexity, by design, ignores “constant-factor overheads” that might matter in some programs so once again this stricter definition of equivalence may be too lenient: we might actually want to know that two implementations take “about the same amount of time.”

None of these definitions are superior. All of them are valuable perspectives computer scientists use all the time. Observable behavior (our definition), asymptotic complexity, and actual performance are all intellectual tools that are used almost every day by someone working on software.