15-150 Fall 2023

Stephen Brookes

Lecture 10
Type-checking, polymorphism, and type inference

1 Outline

We'll say a lot more about:

- Types and typing rules
- Type variables and polymorphism
- Type inference and most general types
- Parameterized datatypes

We'll give examples to illustrate. And we'll summarize the most important technical details, hopefully in a way that doesn't confuse!

2 Introduction

We've started to build foundations for developing well-designed functional programs, with techniques for analyzing runtime behavior and discovering the potential for parallel evaluation (work and span; recurrences, big-O). We have emphasized the importance of specifying and proving correctness (forms of induction, evaluational reasoning, equational reasoning).

These are the basic ingredients of functional programming, and we hope that the examples we've discussed so far already make a strong case for the elegance and power of the functional style. But we haven't yet taken full advantage of some of the most elegant and powerful features of ML. In the next few lectures we will introduce you to some of these features, and we'll demonstrate their utility in a series of interesting examples.

Today's focus is on types, and on language features designed to enable you to "abstract" or "encapsulate" repeated patterns of code and re-use them in type-safe ways. We will introduce "polymorphic" types and parameterized datatype definitions, and we'll talk about type inference and "most general" types. We will also continue to emphasize the importance of specifications and proofs, and we'll show how you can make code design and debugging easier by implementing functions designed to check that your code behaves as intended. Although a few checks on test cases isn't enough to guarantee code correctness, you can often discover simple coding errors by checking.

3 Types

ML has a large repertoire of types, including primitive types like int, real and bool, and compound types built using type constructors, such as tuple or "Cartesian product" types (e.g. int * int, bool * int * real), function types (e.g. int -> int), and list types such as int list. For each type there is a set of "values" of that type. Values of type int correspond to mathematical integers. Values of type int list are (finite) lists of integers.

In ML you can only evaluate an expression if the expression is well-typed, i.e. has a type. Similarly, you can only use a declaration if it is well-typed, and when this happens the declaration introduces bindings that associate types (and values) to names throughout a syntactically determined region (the *scope* of the declaration). You can only use an expression, or a bound name, in a manner that "fits" with its type. The ML implementation rewards your attention to types with the following guarantee: *Well-typed expressions don't go wrong.* More precisely,

> If e has type t, and e evaluates to a value v,
> then v is a value of type t.

Consider the ML function

```
fun split (L:int list):int list * int list =
   case L of
      [ ]     => ([ ], [ ])
    | [x]     => ([x], [ ])
    | x::y::R => let val (A,B) = split R in (x::A, y::B) end
```

(This is the split function used in class.)

We can prove easily by induction on the length of L that for all integer lists L, split(L) terminates. It follows from the type guarantee that the *value* of split(L) must be a pair of integer lists.

We've already used this kind of type-sensitive reasoning when proving code correct in class. Go back and look at the lecture slides and notes prior to these, and identify places where we said things like "because the expression has this type, its value must have a specific form".

4 Typing rules

Type-checking involves figuring out if an expression has a specific type. Given an expression `e` and a type `t` it is pretty simple to check if `e` has type `t`, by appealing to some syntax-directed type rules. We can also use these rules for figuring out if an expression is "type-able" (or "well-typed"), i.e. if it has a type, and what type(s) are appropriate. This is called type inference.

We've already talked, mostly informally, about typing rules. For example:

- An integer numeral, such as `0` or `1`, or `~42`, has type `int`.
- `42.0` has type `real`.
- There are two infix arithmetic operators `+` and `*`, one for integers, one for reals:
    - `e1 + e2` has type `int` if `e1` and `e2` have type `int`.
    - `e1 + e2` has type `real` if `e1` and `e2` have type `real`.
    - `e1 * e2` has type `int` if `e1` and `e2` have type `int`.
    - `e1 * e2` has type `real` if `e1` and `e2` have type `real`.

  Moreover `e1 + e2` is not well typed *unless* both `e1` and `e2` have type `int`, or both have type `real`.
- Similarly the arithmetic comparison operators `<` and `<=` can be used either on two integers or two reals:
    - `e1 < e2` has type `bool` if `e1` and `e2` have type `int`.
    - `e1 < e2` has type `bool` if `e1` and `e2` have type `real`.
    - `e1 < e2` is not well-typed otherwise.

  (The type rules for `<=` are similar.) The `=` operator can be used on integers but not reals:

    - If `e1` and `e2` have type `int`, then `e1 = e2` has type `bool`.
    - `e1 = e2` is not well-typed if `e1` and/or `e2` has type `real`.

  (There are other types, as we've seen, on which equality makes sense and is allowed. Recall that these are the so-called *equality types*.)

- A conditional expression `if e1 then e2 else e3` has type `t`, if and only if `e1` has type `bool` and `e2` and `e3` both have type `t`.

  The rule for case expressions similarly requires that all branches have the same type.

- An application `e1 e2` is well-typed, with type `t2`, if and only if there is some type `t1` such that `e1` has type `t1 -> t2` and `e2` has type `t1`. In other words, a function can only be applied to an argument of the "correct" type.

- An (non-recursive) function expression `fn x => e` is well-typed, with type `t`, if and only if `t` has the form `t1 -> t2` for some types `t1` and `t2` such that, if we assume that `x` has type `t1`, it follows that `e` has type `t2`. In other words, the type of a function indicates what kind of arguments the function expects to be applied to, and what type the result will be.

- A recursive function definition `fun f(x) = e` is well-typed, and introduces the name `f` with type `t1 -> t2`, if and only if `e` has type `t2` when we assume that `x` has type `t1` and `f` has type `t1 -> t2`. This sounds almost circular, but it isn't: `e` may contain occurrences of `x` and `f`, and we're assuming that the recursive calls to `f` have the "correct" type and requiring this to imply that the function body is well-typed.

For example, the declaration

```
fun fact(n:int):int = if n=0 then 1 else n * fact(n-1)
```

is well-typed, and introduces the name `fact` of type `int -> int`. This is because, if we assume that `n:int` and `f:int->int`, the expression

```
if n=0 then 1 else n * fact(n-1)
```

is well-typed and has type `int`.

Exercise: verify this assertion about types, using the rules from above.

Typing and patterns

In the above discussion we deliberately simplified the story by not dealing with patterns and pattern matching. It is quite straightforward to deal with pattern matching and types in a similar manner. The basic idea is that matching a pattern `p` at a type `t` either *succeeds* and introduces type bindings for the variables in the pattern, or *fails*. Here are some of the rules, to give the main ideas. Recall that patterns in ML must obey the syntactic constraint that no variable occurs more than once in the same pattern. That means when a compound pattern like `p1::p2` is used, there's no ambiguity whe n we combine bindings from `p1` and `p2`.

- Matching a numeral pattern (e.g. `0`) at type `t` succeeds if and only if `t` is `int`. No bindings.
- Matching the wildcard pattern `_` at type `t` succeeds, for any type `t`. No bindings.
- Matching a variable pattern `x` at type `t` succeeds, for any type `t`, and binds `x` to type `t`.
- Matching a pair pattern `(p1, p2)` at type `t` succeeds if and only if `t` is a pair type of form `t1 * t2`, matching `p1` at `t1` succeeds, and matching `p2` at `t2` succeeds; on success we combine the bindings from the two component matches. [Patterns in ML are syntactically constrained so that you can't use the same variable twice in one pattern; because of this there is no ambiguity here.]
- Matching the list pattern `[ ]` at type `t` succeeds (with no bindings) if and only if `t` is a list type of form `t1 list` for some type `t1`.
- Matching the list pattern `p1::p2` at type `t` succeeds if and only if `t` is a list type `t1 list`, and matching `p1` at `t1` succeeds, and matching `p2` at `t1 list` succeeds; on success, the match combines the bindings from the component matches.
- Exercise: give a rule for matching a list pattern of form `[p1,p2,...,pk]` (for a given integer `k`).

Notice that the only patterns that can be used for matching at type `real` are wildcard and variable patterns, and this is also the case for function types.

Type bindings and scope

The type of an expression, such as `x+x`, containing (free occurrences of) variables, depends on the types of those variables. When we evaluate an expression with free variables, there will be a set of type bindings currently in scope, and that's where we will find types to use in figuring out if the expression is well-typed. We talk about finding the type of an expression using assumptions about the types of its free variables; or finding the type of an expression using a given collection of type bindings or assumptions about the types of its free variables. We may refer to a list of type bindings (for distinct variables) like $x_1 : \mathtt{t}_1, \ldots, x_k : \mathtt{t}_k$ as a *type environment.* And we will develop ways to make judgements of the form "expression e has type `t`, assuming that variables x_1 through x_k have types $\mathtt{t}_1$ through $\mathtt{t}_k$". More succinctly, such a judgement can be expressed as "in type environment $x_1 : \mathtt{t}_1, \ldots, x_k : \mathtt{t}_k$, expression e has type `t`". For example, we expect that:

- Assuming `x:int`, `x+x` has type `int`.
- Assuming `x:real`, `x+x` has type `real`.
- When `x:t` for a type other than `int` or `real`, `x+x` is not well-typed.

Our use of terms like "assuming" may be misleading here; it would be just as valid to say that "when `x` has type `int`, the expression `x+x` has type `int`.

Here is an ML snapshot that illustrates these points.

```
- val x:int = 42;
val x = 42 : int
- x+x;
val it = 84 : int
- val x:real = 1.0;
val x = 1.0 : real
- x+x;
val it = 2.0 : real
- val x:bool = true;
val x = true : bool
- x+x;
stdIn:17.2 (Type Error)
```

5 Type variables and polymorphic types

We introduced in class a function `rev:int list -> int list` for reversing a list of integers:

```
fun rev(L:int list):int list =
  case L of
     [ ] => [ ]
   | (x::R) => (rev R) @ [x];
```

For example, `rev [1,2,3] = [3,2,1]`. This declaration introduces the name `rev` with type `int list -> int list`. The type annotation included in the function definition echoes this, because we wrote that the argument type (the type of `L`) would be `int list` and the result type (the type of the value returned by the function body) would be `int list` also.

But it should be obvious that the notion of "reversing a list" makes sense for *all* list types, not just for the type `int list`. If we want to reverse a list of strings, i.e. a value of type `string list`, we can't use `rev` as defined here.

```
- fun rev(L:int list):int list =
     case L of [ ] => [ ] | (x::R) => (rev R)@[x];
val rev = fn : int list -> int list
- rev ["f", "oo"];
stdIn:18.1-18.16 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int list
  operand:         string list
  in expression:
    rev ("f" :: "oo" :: nil)
```

We could of course introduce a new function

```
fun revstrings(L:string list):string list =
  case L of
     [ ] => [ ]
   | (x::R) => (revstrings R) @ [x];
```

But this is clearly a bad idea in general. There are infinitely many types of list, and we don't want to have to invent an infinite family of specialized functions for reversal, one for each type!

The reversal functions above are designed in the obvious structural manner, with a base case for the empty list and an inductive case for a non-empty list that makes a recursive call on the tail of the list. Each of these functions (rev and revstrings) does the *same* thing regardless of what the underlying type is for the members of the list.

ML allows us to use a "polymorphic" type, a type containing type variables that represents a whole family of "similar" types. So we can define a single list-reversal function, and use it at *any* type that "fits".

```
fun rev(L : 'a list):'a list =
  case L of
     [ ] => [ ]
   | (x::R) => (rev R) @ [x]
```

This declaration introduces rev:'a list -> 'a list.

Here 'a is an ML type variable, and this "polymorphic" type stands for the family of types of form t list -> t list, where t ranges over the set of types. If we choose a type to use for the type variable 'a, we get an *instance* of this type.
For example 'a list -> 'a list has many instances, including:

```
int list -> int list
string list -> string list
'b list -> 'b list
('b *' b) list -> ('b * 'b) list
```

For another example, the type ('a->'b)->('a->'b) is an instance of 'a->'a.

Notice that 'a -> 'a and 'b -> 'b are instances of each other! When ML reports types to users, it usually chooses type variables early in the alphabetical ordering, starting with 'a, so we see

```
- (fn x => x);
val it = fn - : 'a -> 'a
- (fn x => (fn y => (x, y));
val it = fn - : 'a -> ('b -> 'a * 'b)
```

More examples of functions with polymorphic types:

```
fun fst(x, y) = x;
fun snd(x, y) = y
```

Here we get fst:'a * 'b -> 'a and snd:'a * 'b -> 'b.

All of the standard ML operations on lists have polymorphic types:

```
    (op ::) : 'a * 'a list -> 'a list
    (op @) : 'a list * 'a list -> 'a list
     nil : 'a list
     hd : 'a list -> 'a
     tl : 'a list -> 'a list
     null : 'a list -> bool
```

The accumulator version of list-reversal can be given a polymorphic type:

```
fun revver (L:'a list, A:'a list):'a list =
   case L of
      [ ] =>  A
   |  x::R => revver(L, x::A)
```

introduces `revver :  'a list * 'a list -> 'a list`.

By the way, the proof that we gave earlier, that for all *integer lists* `L` and `A`, `revver(L,A)=(rev L)@A`, didn't make any mention of the assumption that the lists contained integers. The same proof can easily be rewritten to show that for all types `t` and all `L,A:t list`, `revver(L,A)=(rev L)@A`.

Similarly the `zip` function can be generalized:

```
fun zip (L:'a list, R:'b list):('a*'b)list =
   case (L, R) of
      ([ ], _) => [ ]
   |  (_, [ ]) => [ ]
 |  (a::A, b::B) = (a,b) :: zip (A, B)
```

introduces `zip:'a list * 'b list -> ('a * 'b) list`.

Exercise: define a polymorphic version of the unzip function, and say what its type is.

Overloading

Overloading is not the same as polymorphism. ML "overloads" the symbols `+` and `*` and allows their use either at type `int * int -> int` or at type `real * real -> real`, but there is no polymorphic version of `+` with the type `'a * 'a -> 'a`. Let's explain why ML uses overloading rather than polymorphic types for the arithmetic operators.

A well-typed ML expression `e` of type `t` can safely be used in any ML program context that "expects" an expression of type `t`, or an expression whose type is an instance of `t`. For example, `rev` has type `'a list -> 'a list`, so the following expressions are well typed:

```
(a)  rev [1,2,3]
(b)  [true, false] @ (rev [ ])
```

In (a) we use `rev` at type `int list -> int list` and in (b) at type `bool list -> bool list`. Each of these types is an instance of the most general type, which is `'a list -> 'a list`. In (a) we use `rev` as the function in an application expression, and we expect the function to be applicable to an integer list; hence we instantiate `‘a` as `int`.

If ML regarded `+` and `*` as polymorphic operators of type `'a * 'a -> 'a`, instead of allowing overloading, we would have

```
   fun f x = x+x;
 - val f = fn : ‘a -> ‘a
```

and in the scope of this declaration the expressions `f 42`, `f 42.0`, and `f [1,2,3]` would each be well typed (of types `int`, `real` and `int list`, respectively). But evaluating `f [1,2,3]` would cause a runtime error. A polymorphic `+` would invalidate the type guarantee, in that there would be "well typed" expressions whose evaluation doesn't loop forever, yet fails to terminate at a value of the intended type.

Using expressions in a type-safe way

An expression with a polymorphic type can be used at any instance of this type. For example, the following code fragment is trivial, but well-typed:

```
   fun id(x:'a):'a = x;
   val (x, y, z) = (id 42, id true, id [1,2,3])
```

Indeed, here is what the ML implementation says:

```
- fun id(x:'a):'a = x;
val id = fn : 'a -> 'a
- val (x,y,z) = (id 42, id true, id [1,2,3]);
val x = 42 : int
val y = true : bool
val z = [1,2,3] : int list
```

A bit more mind-boggling, perhaps, but in the scope of the above declaration for `id`, the application `id id` is well-typed. This expression actually has type `'a -> 'a`, and we can show this by observing that in `id id` we can give the first occurrence of `id` (the "function") the type `('a -> 'a) -> ('a -> 'a)` and the second occurrence (the "argument") the type `'a -> 'a`. The rule for application then tells us that `id id` has the "result type" of the function's type, i.e. `'a -> 'a`.

5.1 Equality and equality types

ML uses a special "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, `'a, 'b, 'c` and so on are (ordinary) type variables, and can be instantiated with any type. And `''a, ''b, ''c` and so on are "equality type variables" and can only be instantiated with an equality type. The types `int` and `bool` are equality types, but `real` is not. When `t` is an equality type, so is `t list`. When $\mathtt{t}_1, \ldots, \mathtt{t}_k$ are equality types so is $\mathtt{t}_1 * \cdots * \mathtt{t}_k$.

For example:

```
fun mem(x:''a, L:''a list):bool =
   case L of
       [ ] => false
    | y::R => (x=y) orelse mem(x, R)
```

introduces `mem :''a * ''a list -> bool`.

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

```
fun mem(x:'a,L:'a list):bool =
= case L of [ ] => false | y::R => (x=y) orelse mem(x,R);
stdIn:22.35-22.38 Error: operator and operand don't agree [UBOUND match]
  operator domain: ''Z * ''Z
  operand:         'a * 'Y
  in expression:
    x = y
```

The error message is trying to say that `'a` needs to be an equality type (of form `''Z`).

For a type containing equality type variables, such as

```
''a * ''a list -> bool
```

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

6 Type inference

So far we have insisted that you annotate the types of arguments to a function and also indicate the result type of a function. For example:

```
fun sum(L:int list):int =
  case L of
     [ ] => 0
  | x::R => x + sum R
```

You may have realized by now that many such annotations are redundant. In this example, for instance, suppose we began with the stripped down version, minus the type annotations:

```
fun sum L =
  case L of
     [ ] => 0
  | x::R => x + sum R
```

As the typing rule for case expression requires, both branches of the case expression must have the same type; and the first branch is `0`, which has type `int`; so the other branch `x + sum R` needs to get type `int`; hence `x` must get type `int` and `sum R` needs to get type `int`. But if `x` has type `int`, the type for `x::R` and for `R` has to be `int list`. So the type for `L` has to be `int list`, and the result type has to be `int`. Thus the only plausible type for `sum` is `int list -> int`. Finally, if we assume this type for `sum`, plus the types we figured out for `x` and `R`, the right-hand side of the second clause (`x + sum R`) gets type `int`, the correct "result type" as needed.

What we just walked through was an intuitive explanation of how it can be possible to "infer" a type for an expression that can be given a type, even if we don't have any type annotations to work with, by appealing to the typing rules. Guided by the syntactic structure of the expression, we can determine constraints on the possible types that can be used for its sub-expressions and for the variables occurring in the expression.

The ML implementation uses a syntax-directed algorithm for figuring out if your code is typable (if it has a type) and – if so – producing a "most general" type. A most general type (or "m.g.t") for an expression `e` is a type `t` such that `e:t` follows from the typing rules, and such that for every type `t'` with this property, `t'` is an instance of `t`.

If `t` is the most general type of `e`, then ML allows you to use `e` at any instance of type `t`.

When you enter an expression into the ML interpreter, the type reported by ML is always the most general type. When the expression isn't typable, you get an error message instead.

For example, the m.g.t of `fn (x, y) => x` is `'a * 'b -> 'a`. We can use this expression by applying it to any tuple, e.g. `(fn (x,y)=>x) (1,true)` or `(fn (x,y)=>x) (true, 1)`.

For a clausal function declaration, ML requires that the clauses all get the same type. Here is an example. Suppose we have already introduced

```
split : 'a list -> 'a list * 'a list
merge : int list * int list -> int list
```

These are actually the most general types for the `split` and `merge` functions that we used to implement mergesort on integer lists in class (but without the type annotations!).

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

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

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

(ii) Similarly the second clause is fine.

(iii) For the third clause, suppose we give `L` the type `int list`. We need to show that the right-hand-side of the clause gets type `int list`.

We can choose to use type `int list -> int list * int list` for `split`, as this is an instance of its given type. Then `split L` has type `int list * int list`. The val-declaration will give both `A` and `B` the type `int list`, and these type bindings are in scope when we figure out a type for the let-body. Assuming the type `int list -> int list` for the recursive calls to `sort`, the expression `(sort A, sort B)` has type `int list * int list`. So `merge(sort A, sort B)` has type `int list`. This is the required result type, so the third clause is fine too.

The type information produced by the ML 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 sort [ ] = [ ]
 |  sort L = let val (A,B) = split L in merge(sort A, sort B) end
```

This function is only useful for sorting the empty list! It doesn't terminate when applied to a non-empty list! But it *is* well-typed. The surprise is that this type isn't what we probably expected!. ML tells us

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

To check that this `sort` function 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`) because that's an instance of `'a 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` has type `'a list -> 'a list * 'a list`. So `split L` has type `'a list * 'a list`. Matching the pattern `(A, B)` at type `'a list * 'a list` (succeeds and) gives `A` and `B` the type `'a list`, and these type bindings are in scope when we figure out a type for the let-body, which is `merge(sort A, sort B)`. Assuming the type `'a list -> int list` for the recursive calls to `sort`, the expression `(sort A, sort B)` has type `int list * int list`. Remember that `merge` has type `int list * int list -> int list`. So we see

> that merge(sort A, sort B) has type int list. This is the required result type, so the second function clause is fine too. In both cases we get the overall type 'a list -> int list.

OK, so ML says our sort has type 'a list -> int list. But 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's no contradiction here. The ML 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. It turns put that sort ["foo", "bar"] fails to terminate, so the type guarantee doesn't matter.

The lesson: if ML 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

Show that the most general type of the function revver defined by

```
fun revver([ ], A) = A
 |  revver(x::L, A) = revver(L, x::A)
```

is 'a list * 'a list -> 'a list. Perform a similar derivation for the most general type of the function defined by

```
fun revver([ ], A) = [ ]
 |  revver(x::L, A) = revver(L, x::A)
```

and explain how this result meshes with the function's applicative behavior.

7 Parameterized datatypes

We can use type variables to build parameterized datatypes. This allows us to abstract from common templates for data structures, such as lists, trees, and more.

Binary trees

The notion of binary tree is very general: it makes perfect sense to consider trees with values of a given type at the nodes: trees of integers, trees of integer lists, and so on. We can use type variables in datatype definitions to obtain parameterized type constructors, like `’a tree`.

The following parameterized datatype definition

```
datatype ’a tree = Empty | Node of ’a tree * ’a * ’a tree;
```

introduces a *type constructor* (actually a postfix operator on types) `tree`, 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 non-empty 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.

Just as before, and as with every datatype definition, we have a principle of structural induction for general trees. Exactly as before, except that we can use it for reasoning about values of type `t tree`, for any type `t`. So, for a given type `t`, to prove “For all values `T` of type `t tree`, $P(\texttt{t})$ holds”, we do:

- Base case: For `t=Empty`. Show that $P(\texttt{Empty})$ holds.
- Inductive case: For `t = Node(l, v, r)`. Assume as the Induction Hypothesis that $P(\texttt{l})$ and $P(\texttt{r})$ hold, then show that $P(\texttt{Node}(\texttt{l}, \texttt{v}, \texttt{r}))$ holds.

The `size` function for trees generalizes in the obvious way:

```
   fun size T =
      case T of
         Empty => 0
      | Node(l ,x, r) => size l + 1 + size r;
```

We can also generalize the inorder traversal function to work on general trees:

```
(* trav : 'a tree -> 'a list *)
fun trav Empty = [ ]
 |  trav (Node(t1, x, t2)) = (trav t1) @ (x :: trav t2);
(* trav(t) is the list of data built by inorder traversal of t *)
```

Our old type of integer trees corresponds to the type `int tree` built from this datatype definition.

Option types

Another datatype that we will use extensively later is for "option types". Suppose we want to define a "partial" function, whose result is "undefined" for certain inputs, and we want to be able to deal gracefully with the undefined cases. Instead of a function type of shape `t -> t'`, we can use instead the type `t -> t' option`. The name `option` is a type constructor, and it is introduced by the following parameterized datatype definition:

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

We also have constructors

```
   NONE : 'a option
   SOME : 'a -> 'a option
```

Every value of type `t option` is either `NONE`, or has the form `SOME(v)` where `v` is a value of type `t`. We can use patterns built from `NONE` and `SOME` to match against values of an option type.

As a simple example to illustrate the utility of option types, suppose we have a list of (key, value) pairs and we want to answer questions about whether or not there is a pair in the list with a given key, and if so say what the corresponding value is. We need to distinguish between when the key occurs and when it doesn't, and only in the good case do we need to report a value. Option types give us a natural way to do this: NONE means "the key isn't there", and SOME(v) means "the key is there, and is associated with the value v". Here is the function:

```
(* lookup : ''a * (''a * 'b) list -> 'b option *)
fun lookup (x, [ ]) = NONE
 |  lookup (x , (y,v)::L) = if x=y then SOME v else lookup (x, L);
```

Make sure you understand why ML tells us that the most general type of lookup involves an equality type!

8 Referential transparency for types

In the very first week we mentioned Referential Transparency, an important property identified much earlier by logicians and philosophers such as Frege. We can now return to this topic and give a more precise formulation, now that we have explained more about types and well-typed expressions.

We said before that "the type of an expression depends only on the types of its syntactic sub-expressions". A bit more precisely, suppose that E is an expression of type T, containing a sub-expression e of type t. Whenever e' is also an expression of type t, and E' is obtained from E by replacing e by e', E' will also have type T.

That's a rather more formal and precise definition of what we mean by "referential transparency" for types. The reason this happens is because any derivation of a type for E will make use of a type assertion about e; we can obtain a derivation for E' in the same way, just plugging in the type assertion about e' instead. (We won't actually *prove* this result here — it requires a proof by induction on typing derivations, and that's a topic for a more advanced class on type theory or logic.)

9 Self-test

1. Enter the following function declarations in the ML REPL.
   Try to understand how the results correspond to the typing rules.

   ```
   fun ins(x, [ ]) = [x]
   |   ins(x, y::L) = if x>y then y::ins(x,L) else x::y::L;

   fun sort [ ] = [ ]
   |   sort (x::L) = ins(x, sort L);

   fun sort1 [ ] = [ [ ] ]
   |   sort1 (x::L) = ins(x, sort1 L);

   fun sort2 [ [ ] ] = [ ]
   |   sort2 (x::L) = ins(x, sort2 L);

   fun sort3 [ ] = [ ]
   |   sort3 (x::L) = ins x (sort3 L)
   ```

2. In ML the arithmetic operators (like + and *) are not polymorphic, but they are “overloaded”. That means that there are really two versions of + and two versions of *, one for use with integers and one for use with reals, and you’re allowed to use the same syntax for both. Sometimes you may need to tell ML explicitly which one you mean. Check what happens when you enter the following expressions into the ML REPL:

   (a) `fn x => fn y => x+y`
   (You’ll probably see that the default type for + is `int * int -> int`.)

   (b) `fn x:real => fn y => x+y`

   (c) `fn x => fn y:real => x+y`

   (d) `(fn x => fn y => x+y) : real -> real -> real`

   By the way, each of these functions is curried. What happens with the folllowing expressions?

   (a) `fn (x, y) => x+y`

(b) `fn (x:real, y) => x+y`

(c) `fn (x, y:real) => x+y`

(d) `(fn (x, y) => x+y) : real -> real -> real`

3. The ML typing rules prevent evaluation of expressions such as

```
(fn x => x+42) [2,3,4],
```

whose evaluation – if allowed– would encounter an obviously stupid runtime error (here, adding 42 to a list). Look for other examples where the typing rules detect a common programming error.

4. Using the typing rules, show that the following statements are accurate.

(a) `val double = fn x:int => x+x` declares `double : int -> int`.
In the scope of this declaration

- `double 42` has type `int`
- `double (double 42)` has type `int`
- `double double` is not well typed, and neither is `(double double) 42`
- The declaration

```
fun power(x) = if x=0 then 1 else power(double (x-1))
```

declares `power : int -> int`.

- The declaration

```
fun power(x) = if x=0 then 1 else double(power (x-1))
```

declares `power : int -> int`.

5. The built-in infix operator `o` is used to *compose* two functions.
Here is the typing rule for composition:

- Let `t1`, `t2` and `t3` be types.
If `f` has type `t1 -> t2` and `g` has type `t2 -> t3`,
then `g o f` has type `t1 -> t3`.

Which of the following expressions is/are well typed? Explain briefly.

(a) `(fn x => x+1) o (fn y => y+y)`

(b) `(fn x => x+x) o (fn y => y+1)`

(c) `(fn x => x+1) o (fn y => (y>42))`

(d) `(fn x => (x>42)) o (fn y => y+y)`

For each well typed expression in the above list, find a function value that's equivalent to the given expression and has a simple form: in the body of the function you're only allowed to do arithmetic and boolean operations (no composition!).

For example, the composition expression

```
(fn x => x+1) o (fn y => y+2)
```

is equivalent to `fn z => z+3`.

6. The composition operator is actually polymorphic, which seems natural because you can compose functions of arbitrary types so long as the result type of the "first" function agrees with the argument type of the "second". In `g o f` we call `f` the first function and `g` the second, because the composite function behaves as if it applies `f` first, then applies `g` to the result. Using the ML REPL, see what happens with the following declarations.

```
fun comp1(f, g) = g o f;
fun comp2(f, g) = f o g;
fun comp3(f, g) = fn x => g(f x);
fun comp4 f = f o f
```

7. Being correctly typed is only part of the story! Show that the following ML declaration declares `ins2` of type `int * int list -> int list`, the same type as `ins`, but this function is not equivalent to `ins`.

```
fun ins2(x, [ ]) = [ ]
|   ins2(x, y::L) = if x>y then y::ins2(x,L) else x::y::L
```

[Find an argument for which `ins` and `ins2` return unequal results.]

8. Suppose `e` is a syntactic value with (most general) type `'a -> 'b`.

   (i) What does the type guarantee tell us about the applicative behavior of `e`?

(ii) For each of the following expressions, find a most general type, or explain why it doesn't have one. Also say what happens when the expression is evaluated,

   (a) `e 42`

   (b) `42 + e`

   (c) `let val f : int -> int = (e e) in f 42 end`

9. Find an ML syntactic value `f` of (most general) type `'a -> 'b`. What does the type guarantee imply about the evaluational behavior of `f 42` and `f true`?

10. What is the most general type for the following functions?

```
fun index (x, [ ]) = NONE
|   index (x, y::L) = if x=y then SOME 1 else
     case index(x, L) of
         NONE => NONE
       | SOME i => SOME (i+1)

fun lookup (i, [ ]) = NONE
|   lookup (1, x::L) = SOME x
|   lookup (n, x::L) = lookup (n-1, L)
```

In the scope of these declarations, which of the following expressions is/are well typed, and what are their values?

(a) `index (42, [true, false])`

(b) `index(2, [1.0, 2.0+3.0, 4.0+5.0])`

(c) `lookup (~42, [1,2,3])`

(d) `fn (x, L) => lookup(index(x, L), L)`