15-150 Fall 2018

Stephen Brookes

Lecture 9
Type-checking, polymorphism, and type inference
1 Outline

- Types and type-checking
- Type variables and polymorphic types
- Type inference and most general types
- Parameterized datatypes and type abstraction

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. We will introduce “polymorphic” types, 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 \texttt{int}, \texttt{real} and \texttt{bool}, and compound types built from other types using type constructors, such as \texttt{int * int}, \texttt{bool * int * real}, \texttt{int -> int}, and \texttt{int list}. For each type there is a set of “values” of that type. Values of type \texttt{int} correspond to mathematical integers. Values of type \texttt{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 to names throughout a syntactically determined region. 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.

\[
\text{If } e \text{ has type } t, \text{ and } e \text{ evaluates to a value } v, \\
\text{ then } v \text{ is a value of type } t.
\]

More informally, \textit{Well-typed expressions don’t go wrong}. Consider the ML function

\[
\text{fun split (L:int list):int list * int list =} \\
\text{case L of} \\
\hspace{1cm} [ ] \hspace{1cm} => ([ ], [ ]) \\
\hspace{1cm} [x] \hspace{1cm} => ([x], [ ]) \\
\hspace{1cm} x::y::R \hspace{1cm} => \text{let val (A,B) = split R in (x::A, y::B) end}
\]

(This is an alternative way to define the \texttt{split} function used in class, with a \texttt{case} expression rather than clauses.)

We can prove easily by induction on the length of \texttt{L} that for all integer lists \texttt{L}, \texttt{split(L)} terminates. It follows from the type guarantee that the \textit{value} of \texttt{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”.

3
4 Type-checking

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’ve talked mostly informally about such rules. For example:

- A numeral, such as 0 or 1, has type \texttt{int}.

- The infix arithmetic operators + and * can be used on two integers, or on two reals:
  - $e_1 + e_2$ has type \texttt{int} if $e_1$ and $e_2$ have type \texttt{int}.
  - $e_1 + e_2$ has type \texttt{real} if $e_1$ and $e_2$ have type \texttt{real}.
  - $e_1 + e_2$ is not well-typed otherwise.
  - $e_1 * e_2$ has type \texttt{int} if $e_1$ and $e_2$ have type \texttt{int}.
  - $e_1 * e_2$ has type \texttt{real} if $e_1$ and $e_2$ have type \texttt{real}.
  - $e_1 * e_2$ is not well-typed otherwise.

Similarly the arithmetic comparison operators < and $\leq$ can be used either on two integers or two reals:

- $e_1 < e_2$ has type \texttt{bool} if $e_1$ and $e_2$ have type \texttt{int}.
- $e_1 < e_2$ has type \texttt{bool} if $e_1$ and $e_2$ have type \texttt{real}.
- $e_1 < e_2$ is not well-typed otherwise.

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

- If $e_1$ and $e_2$ have type \texttt{int}, then $e_1 = e_2$ has type \texttt{bool}.
- $e_1 = e_2$ is not well-typed if $e_1$ and/or $e_2$ has type \texttt{real}.

(There are other types, as we will see, on which equality makes sense and is allowed.)

- A conditional expression if $e_1$ then $e_2$ else $e_3$ has type $t$, if and only if $e_1$ has type \texttt{bool} and $e_2$ and $e_3$ both have type $t$. 

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

• An application \( e_1 \ e_2 \) is well-typed, with type \( t \), if and only if there is some type \( t_2 \) such that \( e_1 \) has type \( t_2 \rightarrow t \) and \( e_2 \) has type \( t_2 \).
  In other words, and very naturally, a function expression can only be applied to an argument expression of the “correct” type.

• An (anonymous) non-recursive function expression \( \text{fn} \ x:t_2 \Rightarrow e \) is well-typed, with type \( t \), if and only if \( t \) has the form \( t_2 \rightarrow t' \) for some type \( t' \) such that, if we assume that \( x \) has type \( t_2 \), we can show that \( e \) has type \( t' \). In other words, the type of a function expression indicates what kind of arguments the function expects to be applied to, and what type the result will be.

• A recursive function definition \( \text{fun} \ f(x:t_2):t = e \) is well-typed, and introduces the name \( f \) with type \( t_2 \rightarrow t \), if and only if \( e \) has type \( t \) when we assume that \( x \) has type \( t_2 \) and \( f \) has type \( t_2 \rightarrow t \). 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

\[
\text{fun} \ \text{fact}(n:\text{int}):\text{int} = \text{if} \ n=0 \ \text{then} \ 1 \ \text{else} \ n \times \text{fact}(n-1)
\]

is well-typed, and introduces the name \( \text{fact} \) of type \( \text{int} \rightarrow \text{int} \). This is because, if we assume that \( n:\text{int} \) and \( f:\text{int} \rightarrow \text{int} \), the expression

\[
\text{if} \ n=0 \ \text{then} \ 1 \ \text{else} \ n \times \text{fact}(n-1)
\]

is well-typed and has type \( \text{int} \).

Exercise: verify this assertion about types, using the rules from above.
Type-checking 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 \( p \) to 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. To avoid confusion with value-matching (matching a pattern against a value of some type), we will talk about fitting a pattern to a type.

- Fitting pattern \( 0 \) to type \( t \) succeeds (with no bindings) if and only if \( t \) is \( \text{int} \).
- Fitting the wildcard pattern \( _ \) to type \( t \) succeeds (with no bindings).
- Fitting a variable pattern \( x \) to type \( t \) succeeds, and binds \( x \) to type \( t \).
- Fitting a pair pattern \( (p_1, p_2) \) to type \( t \) succeeds if and only if \( t \) is a pair type of form \( t_1 \times t_2 \), fitting \( p_1 \) to \( t_1 \) succeeds, and fitting \( p_2 \) to \( t_2 \) succeeds; on success, the fit produces the bindings from the component fittings. [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.]
- Fitting \( [\ ] \) to \( t \) succeeds (with no bindings) if and only if \( t \) is a list type of form \( t_1 \text{ list} \) for some type \( t_1 \).
- Fitting \( p_1::p_2 \) to \( t \) succeeds if and only if \( t \) is a list type \( t_1 \text{ list} \), fitting \( p_1 \) to \( t_1 \) succeeds, and fitting \( p_2 \) to \( t_1 \text{ list} \) succeeds; on success, the fit produces the bindings from the component fittings.

Type bindings and scope

The type of an expression with free variables, such as \( x+x \), 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.
• If we assume \( x : \text{int} \), the expression \( x + x \) has type \text{int}.

• If we assume \( x : \text{real} \), the expression \( x + x \) has type \text{real}.

• If we assume \( x : t \) for some other type (\( t \) is not \text{int} or \text{real}), the expression \( x + x \) is not well-typed.

Our use of terms like “if we assume” may be misleading here; it would be just as valid to say that “when \( x \) has type \text{int}, the expression \( x + x \) has type \text{int}.

Here is a snapshot from the ML implementation 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 Error: overloaded variable not defined at type
  symbol: +
  type: bool
```

Make sure you understand what happened here, and how the ML reporting information is consistent with what we said above.
5 Type variables and polymorphic types

We introduced in class a function \texttt{rev:int list -> int list} for reversing a list of integers:

\begin{verbatim}
fun rev(L:int list):int list =
  case L of
    [ ] => [ ]
  | (x::R) => (rev R) @ [x];
\end{verbatim}

For example, \texttt{rev [1,2,3] = [3,2,1]}. This declaration introduces the name \texttt{rev} with type \texttt{int list -> int list}. The type annotation included in the function definition echoes this, because we wrote that the argument type (the type of \texttt{L}) would be \texttt{int list} and the result type (the type of the value returned by the function body) would be \texttt{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 \texttt{int list}. If we want to reverse a list of strings, i.e. a value of type \texttt{string list}, we can’t use \texttt{rev} as defined here.

\begin{verbatim}
- 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)
\end{verbatim}

We could of course introduce a new function

\begin{verbatim}
fun revstrings(L:string list):string list =
  case L of
    [ ] => [ ]
  | (x::R) => (revstrings R) @ [x];
\end{verbatim}

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 (\texttt{rev} and \texttt{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”.

\begin{verbatim}
fun rev(L : 'a list):'a list =
case L of
  [ ] => [ ]
| (x::R) => (rev R) @ [x];
\end{verbatim}

This declaration introduces \texttt{rev:'a list -> 'a list}.

Here \texttt{'a} is an ML type variable, and this “polymorphic” type stands for the family of types of form \texttt{t list -> t list}, where \texttt{t} ranges over the set of types. If we choose a type to use for the type variable \texttt{'a}, we get an instance of this type.

For example \texttt{int list -> int list} is an instance of \texttt{'a list -> 'a list}. So is \texttt{string list -> string list}. And so is \texttt{('b*'b)list -> ('b*'b)list}. We can even substitute (or instantiate) using a type with type variables. For example, the type \texttt{('a->'b)->('a->'b)} is an instance of \texttt{a->'a}.

More examples of functions with polymorphic types:

\begin{verbatim}
fun fst(x:'a, y:'b):'a = x;
fun snd(x:'a, y:'b):'b = y;
\end{verbatim}

Here we get \texttt{fst:'a*'b->'a} and \texttt{snd:'a*'b->'b}.

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

\begin{verbatim}
(* (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 *)
\end{verbatim}

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.

NOTE: 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.

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 \texttt{id}, the application \texttt{id id} is well-typed. This expression actually has type `'a -> 'a`, and we can show this by observing that in \texttt{id id} we can give the first occurrence of \texttt{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 \texttt{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 \texttt{=} to test for equal values. For example, \texttt{`a}, \texttt{`b}, \texttt{`c} and so on are (ordinary) type variables, and can be instantiated with any type. And \texttt{''a}, \texttt{''b}, \texttt{''c} and so on are “equality type variables” and can only be instantiated with an equality type. The types \texttt{int} and \texttt{bool} are equality types, but \texttt{real} is not. When \texttt{t} is an equality type, so is \texttt{t list}. When \texttt{t_1}, \ldots, \texttt{t_k} are equality types so is \texttt{t_1 * \cdots * t_k}.

For example:

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

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

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

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

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

- \texttt{operator domain: ''Z * ''Z}
- \texttt{operand: 'a * 'Y}

in expression:

```
x = y
```

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

```haskell
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:

```haskell
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 $\text{fn (x, y) => x}$ is $\text{'a * 'b -> 'a}$. We can use this expression by applying it to any tuple, e.g. $(\text{fn (x,y) => x}) (1, \text{true})$ or $(\text{fn (x,y) => x}) (\text{true}, 1)$.

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

$$\text{split : 'a list -> 'a list * 'a list}$$
$$\text{merge : int list * int list -> int list}$$

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

$$\text{fun sort [ ] = [ ]}$$
$$| \text{sort [x] = [x]}$$
$$| \text{sort L = let val (A,B) = split L in merge(sort A, sort B) end;}$$

To check that $\text{sort}$ has type $\text{int list -> int list}$, let’s check that each clause fits with this type.

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

(ii) Similarly the second clause is fine.

(iii) For the third clause, suppose we give $L$ the type $\text{int list}$. We need to show that the right-hand-side of the clause gets type $\text{int list}$. 
We can choose to use type \( \text{int list} \rightarrow \text{int list} * \text{int list} \) for \( \text{split} \), as this is an instance of its given type. Then \( \text{split} \) \( L \) has type \( \text{int list} * \text{int list} \). The val-declaration will give both \( A \) and \( B \) the type \( \text{int list} \), and these type bindings are in scope when we figure out a type for the let-body. Assuming the type \( \text{int list} \rightarrow \text{int list} \) for the recursive calls to \( \text{sort} \), the expression \( (\text{sort} \ A, \text{sort} \ B) \) has type \( \text{int list} * \text{int list} \). So \( \text{merge}(\text{sort} \ A, \text{sort} \ B) \) has type \( \text{int list} \). 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.

```ml
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

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

To check that this \( \text{sort} \) has type \( 'a \text{ list} \rightarrow \text{int list} \), 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 \( 'a \text{ list} \) the right-hand-side \( [\ ] \) can be used at the required result type \( \text{int list} \).

(ii) For the other clause, suppose we give \( L \) the type \( 'a \text{ list} \). We need to show that the right-hand-side of the clause gets type \( \text{int list} \).

We know that \( \text{split} \) can be used at type \( 'a \text{ list} \rightarrow 'a \text{ list} * 'a \text{ list} \). Then \( \text{split} \) \( L \) has type \( 'a \text{ list} * 'a \text{ list} \). The val-declaration will give both \( A \) and \( B \) the type \( 'a \text{ list} \), and these type bindings are in scope when we figure out a type for the let-body. Assuming the type \( 'a \text{ list} \rightarrow \text{int list} \) for the recursive calls to \( \text{sort} \), the expression \( (\text{sort} \ A, \text{sort} \ B) \) has type \( \text{int list} * \text{int list} \). So \( \text{merge}(\text{sort} \ A, \text{sort} \ B) \) has type \( \text{int list} \). This is the required result type, so the third clause is fine too.
OK, so ML says our `sort` has type `'a list -> int list`. Isn’t that a problem? Surely if we apply `sort` to a list of strings we can’t expect it to produce a list of integers?? Well, of course not. But there’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`.

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

```ml
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

```ml
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.

General trees

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

    datatype tree = Empty | Node of tree * int * tree;

This datatype definition introduces a type name tree, a value Empty of type tree, and a function Node of type tree * int * tree -> tree.

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

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(t) holds”, we do:

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

The size function for trees generalizes in the obvious way:

\[
(* \text{size : 'a tree -> int *}) \\
\text{fun size (T:'a tree):int =} \\
\quad \text{case T of} \\
\quad \quad \text{Empty => 0} \\
\quad \quad \text{| Node(l,x,r) => size(l) + 1 + size;}
\]

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

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

Our old type of integer trees corresponds to the type \text{int tree} built from this datatype definition.

Many of the earlier functions dealing with lists and trees can be made polymorphic: the same code can be used at all instances of the most general type.
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 $t \rightarrow t'$, we can use instead the type $t \rightarrow t'$ 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 Self-test 9

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

   ```ml
   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 following expressions?

   (a) ```fn (x, y) => x+y```
3. This question is just a thought experiment to explain why ML uses overloading rather than polymorphic types for the arithmetic operators. You should read carefully and understand what’s being said.

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

\[(\text{fn } x \Rightarrow x + 42) \ [2,3,4],\]

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.

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

(a) \((\text{fn } x: \text{int} => x + x)\) has type \(\text{int} -> \text{int}\)

(b) \((\text{fn } x: \text{real} => x + x)\) has type \(\text{real} -> \text{real}\)

(c) \(\text{val double} = \text{fn } x: \text{int} => x + x\) declares \(\text{double} : \text{int} -> \text{int}\).

In the scope of this declaration

- \(\text{double } 42\) has type \(\text{int}\)
- \(\text{double } (\text{double } 42)\) has type \(\text{int}\)
- \(\text{double } \text{double}\) is not well typed, and neither is \((\text{double } \text{double}) \ 42\)
- The declaration

\[
\text{fun power}(x: \text{int}) = \text{if } x=0 \ \text{then } 1 \ \text{else power}(\text{double } (x-1))
\]

declares \(\text{power} : \text{int} -> \text{int}\).

- The declaration

\[
\text{fun power}(x: \text{int}) = \text{if } x=0 \ \text{then } 1 \ \text{else double(power } (x-1))
\]

declares \(\text{power} : \text{int} -> \text{int}\).

6. The built-in infix operator \(\circ\) is used to compose two functions. Here is the typing rule for composition:

- Let \(t_1, t_2\) and \(t_3\) be types. If \(f\) has type \(t_1 -> t_2\) and \(g\) has type \(t_2 -> t_3\), then \(g \circ f\) has type \(t_1 -> t_3\).

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

(a) \((\text{fn } x: \text{int} => x+1) \circ (\text{fn } y: \text{int} => y+y)\)

(b) \((\text{fn } x: \text{int} => x+x) \circ (\text{fn } y: \text{int} => y+1)\)

(c) \((\text{fn } x: \text{int} => x+1) \circ (\text{fn } y: \text{int} => (y>42))\)

(d) \((\text{fn } x: \text{int} => (x>42)) \circ (\text{fn } y: \text{int} => y+y)\)
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

$$(\text{fn } x \Rightarrow x+1) \circ (\text{fn } y \Rightarrow y+2)$$

is equivalent to $\text{fn } z \Rightarrow z+3$.

7. 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 $g \circ f$ 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.

```ml
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
```

8. Being correctly typed is only part of the story! Show that the following ML declaration declares $\text{ins2}$ of type $\text{int} \times \text{int list} \rightarrow \text{int list}$, the same type as $\text{ins}$, but this function is not equivalent to $\text{ins}$.

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

[Find an argument for which $\text{ins}$ and $\text{ins2}$ return unequal results.]

9. Suppose $e$ is a syntactic value with (most general) type $'a \rightarrow 'b$.

(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,
10. Find a syntactic value of (most general) type ’a -> ’b.

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

```ml
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)

12. Write an ML function

```ml
spray : 'a list -> 'a list
```

such that for all types t, all n ≥ 0, and all values \(x_1, \ldots, x_n\) of type t,

\[
spray \ [x_1, \ldots, x_n] = [x_1, x_3, \ldots, x_{2k-1}]
\]

where \(k\) is the largest integer such that \(2k - 1 \leq n\).

Find some other functions of type ’a list -> ’a list.