15–150: Principles of Functional Programming

Type-checking, Polymorphism, and Type Inference

Spring 2020*

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 have started to build foundations for writing functional programs (key tools: pattern-matching, recursion), understanding the runtime behavior of programs and the potential of parallel evaluation (key concepts: work and span; key tools: recurrences, big-O), and specifying and proving correctness (key tools: induction, extensional equivalence).

These are the basic ingredients of functional programming, and we hope that the examples we have discussed so far already make a strong case for the elegance and power of the functional style. However, we haven't yet taken full advantage of some of the most elegant and powerful features of SML. In the next few lectures we will introduce you to some of these features, and we will 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 repeated patterns of code. We will introduce polymorphic types, parameterized datatype definitions, and we will talk about type inference and most general types.

3 Types

SML has a large repertoire of types, including primitive types like `int`, `real` and `bool`, and compound types built from other types using type constructors, such as `int * int`, `bool * int * real`, `int -> int`, and `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.

*Adapted nearly verbatim by Michael Erdmann from an earlier document by Stephen Brookes.

In SML 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 SML implementation rewards your attention to types with the following guarantee.

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

Consider the SML 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
```

We can prove by induction that for all integer list values `L`, `split(L)` reduces to a value. It follows from the type guarantee that the *value* of `split(L)` must be a pair of integer lists.

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 `int`.
- The infix arithmetic operators `+` and `*` can be used on two integers, or on two reals:
  - `e1+e2` has type `int` if and only if `e1` and `e2` have type `int`.
  - `e1*e2` has type `int` if and only if `e1` and `e2` have type `int`.
  - `e1+e2` has type `real` if and only if `e1` and `e2` have type `real`.
  - `e1*e2` has type `real` if and only if `e1` and `e2` have type `real`.

  For `=`, if `e1` and `e2` have type `int`, then `e1=e2` has type `bool`. (This is not "if and only if", because there are other types, as we will see, on which equality makes sense.) You *cannot* use `=` on reals.
- A conditional expression of the form `if e1 then e2 else e3` is well-typed, and 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 `e2 e1` is well-typed, with type `t2`, if and only if there is some type `t1` such that `e2` has type `t1 -> t2` and `e1` has type `t1`.
- An (anonymous) non-recursive function expression `fn x:t1 => e` is well-typed, with type `t`, if and only if `t` has the form `t1 -> t2` for some type `t2` such that, if we assume that `x` has type `t1`, we can show that `e` has type `t2`.

- A recursive function declaration `fun f(x:t1):t2 = 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 the previous page.

Type-checking and patterns

In the previous discussion we deliberately simplified the story by not dealing with patterns and pattern matching. We now expand the discussion. 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*, in which case the encompassing expression or declaration is not well-typed. Here are some of the rules, to give the main ideas.

- Matching pattern `0` (the integer zero) to type `t` succeeds (with no bindings) if and only if `t` is `int`.
- Matching a variable `x` to type `t` succeeds, and binds `x` to have type `t`.
- Matching the wildcard pattern _ to type `t` succeeds (with no bindings).
- Matching a pair pattern `(p1, p2)` to type `t` succeeds if and only if `t` is a pair type of the form `t1 * t2`, matching `p1` to `t1` succeeds, and matching `p2` to `t2` succeeds; on success, the match produces the bindings from the component matches. [Patterns in SML 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 `[ ]` to `t` succeeds (with no bindings) if and only if `t` is a list type of the form `t1 list` for some type `t1`.
- Matching `p1::p2` to `t` succeeds if and only if `t` is a list type `t1 list`, matching `p1` to `t1` succeeds, and matching `p2` to `t1 list` succeeds; on success, the match produces the bindings from the component matches.

Type bindings, 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:int`, the expression `x+x` has type `int`.
- If we assume `x:real`, the expression `x+x` has type `real`.
- If we assume `x:t` for some other type (`t` is not `int` or `real`), the expression `x+x` is not well-typed.

Here is a snapshot from the SML 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
```

5 Type variables and polymorphic types

5.1 Polymorphism

Repeated patterns of computation

Here is one implementation of the 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 declaration 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 produced by the function call) would be `int list` also.

The notion of “reversing a list” makes sense for *all* list types, not just for the type `int list`. However, if we want to reverse a list of strings, i.e., a value of type `string list`, we can’t use `rev` as defined above. If we try, we get this error:

```
- 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]
```

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!

Parameterized types

The reversal functions above are designed with the list structure in mind: Each has a “base” case clause for the empty list and an “inductive” case clause for nonempty lists. That second clause 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.

SML allows us to use *polymorphic* types. A polymorphic type contains 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 is part of the family:

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

This declaration introduces `rev:’a list -> ’a list`, with `’a` being an SML type variable.

The polymorphic type `’a list -> ’a list` stands for a family of types of the 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, `int list -> int list` and `string list -> string list` and `int list list -> int list list` are all instances of `’a list -> ’a list`. So is `(’b*’b)list -> (’b*’b)list`. We can even substitute (or instantiate) using a type with type variables. For example, the type `(’a->’b)->(’a->’b)` is an instance of `’a->’a`.

More examples of polymorphic types

More examples of functions with polymorphic types:

```
fun fst(x:’a, y:’b):’a = x
fun snd(x:’a, y:’b):’b = y
```

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

All of the standard SML 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 trev (L:'a list, acc:'a list):'a list =
   case L of
      [ ] =>  acc
   |  x::R => trev(R, x::acc)
```

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

And again, one can prove by structural induction that for all types `t` and all `L,acc:t list`, `trev(L,acc)`$\cong$`(rev L)@acc`.

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.

Using polymorphic types

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 SML 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 left occurrence of `id` (the "function") the type `('a -> 'a) -> ('a -> 'a)` and the right 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, namely, `'a -> 'a`.

Overloading

Overloading is not the same as polymorphism. SML "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`.

5.2 Equality and equality types

Caution: Many programming language experts consider equality types to be a misfeature of SML. We will not generally use equality types in this course, but mention them in case you run into them.

SML 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 the 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`.

What types are equality types? The basic types `int`, `char`, `string`, and `bool` are equality types. The type `real` is not. Datatypes consisting only of constant constructors are equality types (an example we will see later is the type `order`). Product types (tuples) are equality types iff each of the components of the product is an equality type. Similarly, any structured type (such as one built from a datatype declaration) is an equality type iff each of its component types is an equality type. For instance, an `int list` is an equality type whereas a `real list` is not. Function types and abstract types (to be defined later in the semester) are not equality types.

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 dropped 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.

Most general types

The SML implementation uses a syntax-directed algorithm for figuring out whether your code is well-typed 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 SML allows you to use `e` at any instance of type `t`.

When you enter an expression into the SML interpreter, the type reported by SML is always the most general type. When the expression is not well-typed, 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, SML 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 SML implementation can help to debug code! Consider the following erroneous sorting function, obtained from the above function by omitting the clause for singleton lists.

```
  fun 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 loops forever when applied to a nonempty list! Nonetheless, it *is* well-typed. The surprise is that this type isn't what we probably expected!). SML tells us

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

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

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

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

OK, so SML 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 is no contradiction here. The SML type guarantee is that we can use `sort` in any way consistent with an instance of this type. So let `L` be a string list. If `sort(L)` reduces to a value, the guarantee is that its value will be an integer list. The only situation where this happens is for `L=[ ]`, and the empty list is also a perfectly valid value of type `int list`.

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

Exercise

Work through how SML figure out that the most general type of the function `trev` defined by

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

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

Exercise

Now find the most general type of `trev` given this declaration:

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

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`.

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

The following parameterized datatype definition

```
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 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 = Node(l, v, r)`. Assume as the Induction Hypothesis that P(`l`) and P(`r`) hold, then show that P(`Node(l,v,r)`) holds.

The `size` function for trees generalizes:

```
   (* size : 'a tree -> int *)
   fun size (T:'a tree):int =
      case T of
         Empty => 0
      | Node(t1,x,t2) => size(t1) + 1 + size(t2)
```

or as:

```
   fun size (Empty : 'a tree) : int = 0
     | size (Node(t1,x,t2) : 'a tree) : int = size(t1) + 1 + size(t2)
```

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)
```

Our old type of integer trees corresponds to the type `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 example that we will use extensively later is *option types.* Sometimes 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`. Here `option` is a type constructor, and it is introduced by the following parameterized datatype definition:

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

We thus have

```
   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.

To illustrate the utility of option types, suppose we have a list of (key, value) pairs and we want to determine whether 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 a value v". The function appears below.

Notice that we determine equality of keys via a function that we pass explicitly to the lookup function, rather than rely on SML's built-in equality types. That way we can work with types that are not necessarily built-in equality types.

```
(* lookup : ('a * 'a -> bool) * 'a * ('a * 'b) list -> 'b option
   REQUIRES: true
   ENSURES:  lookup(eq, x, L) evaluates to
               SOME(v) of the leftmost (y,v) in L for
                  which eq(x,y) evaluates to true, if there is such a (y,v);
               NONE otherwise.
*)

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

Here is a function that uses pattern-matching on option values.
Find a good specification for it.

```
(* try : ('a -> 'b option) * 'a list -> 'b option *)

fun try(f, [ ]) = NONE
 |  try(f, x::L) = case (f x) of
                      NONE   => try(f, L)
                    | SOME v => SOME v
```