

CS558 Programming Languages

Winter 2008

Lecture 12

FLEXIBILITY OF DYNAMIC TYPECHECKING

Static typechecking offers the great advantage of **catching errors early**, and generally supports more efficient execution.

Why ever consider dynamic typechecking?

- **Simplicity.** For short or simple programs, it's nice to avoid the need for declaring the types of identifiers.
- **Flexibility.** Static typechecking is inherently more **conservative** about what programs it admits.

For example, suppose function `f()` happens to always return `false`. Then

```
(if f() then "a" else 2) + 2
```

will never cause a runtime type error, but it will still be rejected by a static type system.

Perhaps more usefully, dynamic typing allows **container** data structures, to contain mixtures of values of arbitrary types, like this "list":

```
[2, true, 3.14]
```

TYPE INFERENCE

Some statically-typed languages, like Standard ML, offer alternative ways to approach these goals, via **type inference** and **polymorphic typing**.

Type inference works like this:

- The types of identifiers are automatically inferred from the way they are **used**.
- The programmer is no longer required to declare the types of identifiers (although this is still permitted).
- Requires that the types of operators and literals are known.

INFERENCE EXAMPLES

(Assume just `int` and `bool` as base types.)

```
let fun f x = x + 2
in f y
end
```

The type of `x` must be `int` because it is used as an arg to `+`. So the type of `f` must be `int -> int`, and `y` must be an `int`.

```
let fun f x = [x]
in f true
end
```

Suppose `x` has some type `t`. Then the type of `f` must be `t -> t list`. Since `f` is applied to a `bool`, we must have `t = bool`.

(For the moment, we're assuming the `f` must be given a unique **monomorphic** type; in real ML, this isn't true...)

SYSTEMATIC INFERENCE

A harder example:

```
let fun f x = if x then p else q
in 1 + (f r)
end
```

Can only infer types by looking at **both** the function's body and its applications.

In general, we can solve the inference task by extracting a collection of typing **constraints** from the program's AST, and then finding a simultaneous solution for the constraints using **unification**.

Extract constraints that tell us how types **must** be related if we are to be able to find a typing derivation. Each node generates one or more constraints.

INFERENCE FOR ML-LIKE FUNCTIONS

We can rewrite the example slightly as

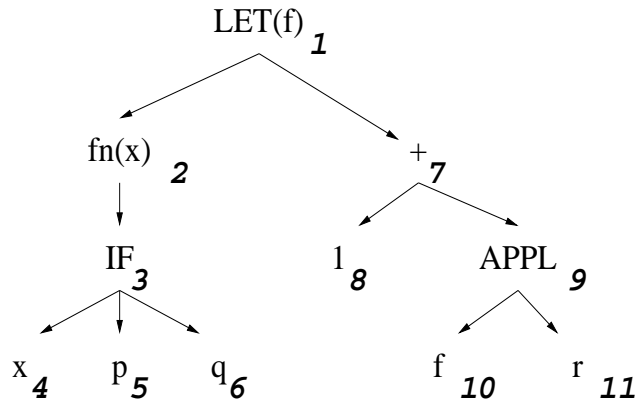
```
let val f = fn x => if x then p else q
in 1 + (f r)
end
```

We'll need some extra inference rules:

$$\frac{TE + \{x \mapsto t_1\} \vdash e : t_2}{TE \vdash \text{fn } x \Rightarrow e : t_1 \rightarrow t_2} \text{ (Fn)}$$

$$\frac{TE \vdash e_1 : t_1 \rightarrow t_2 \quad TE \vdash e_2 : t_1}{TE \vdash e_1 e_2 : t_2} \text{ (Appl)}$$

INFERENCE EXAMPLE



SOLVING INFERENCE CONSTRAINTS

Node	Rule	Constraints
1	Let	$t_f = t_2$ $t_1 = t_7$
2	Fn	$t_2 = t_x \rightarrow t_3$
3	If	$t_4 = \text{bool}$ $t_3 = t_5 = t_6$
4	Var	$t_4 = t_x$
5	Var	$t_5 = t_p$
6	Var	$t_5 = t_q$
7	Add	$t_7 = t_8 = t_9 = \text{int}$
8	Int	$t_8 = \text{int}$
9	Appl	$t_{10} = t_{11} \rightarrow t_9$
10	Var	$t_{10} = t_f$
11	Var	$t_{11} = t_r$

Solution : $t_1 = t_7 = t_8 = t_9 = t_3 = t_5 = t_p = t_6 = t_q = \text{int}$
 $t_4 = t_x = t_{11} = t_r = \text{bool}$ $t_2 = t_f = t_{10} = \text{bool} \rightarrow \text{int}$

DRAWBACKS OF INFERENCE

Consider this variant program:

```
let fun f x = if x then p else false
in 1 + (f r)
end
```

Now the body of `f` return type `bool`, but it is used in a context expecting an `int`.

The corresponding extracted constraints will be **inconsistent**; no solution can be found. Can report this to the programmer.

But which is wrong, the definition of `f` or the use? Doesn't really work to associate the error message with a single program point. (In general, may need to consider an arbitrarily long chain of program points.)

PARAMETRIC POLYMORPHISM

By default ML infers the most polymorphic possible type for every function. In this case, it would give `head` the type `'a list → 'a`, where `'a` (pronounced “alpha”) is implicitly universally quantified. Each use of `head` occurs at a particular **instance** of `'a` (first at `bool`, then at `int`).

This is called **parametric polymorphism** because the function definition is (implicitly) parameterized by the instantiating type.

In this model, the **behavior** of the polymorphic function is **independent** of the instantiating type. In fact, an ML compiler typically generates just one piece of object code for each polymorphic function, shared by all instances. (More later.) An alternative is to generate type-specific versions of the code for each different instance.

Consider

```
let fun head (x::xs) = x
in head [1,2,3] end
```

By extracting the constraints as above, and solving, we will conclude that `head` has type `int list → int`.

It is also perfectly sensible to write:

```
let fun head (x::xs) = x
in head [true,false,true] end
```

giving `head` the type `bool list → bool`. Note that the definition of `head` hasn't changed at all!

So reasonable to ask: why can't we write something like:

```
let fun head (x::xs) = x
in (head [true,false,true],
    head [1,2,3]) end
```

Can do this if we treat the type of `head` as **polymorphic**.

PARAMETRIC POLYMORPHISM VS. OVERLOADING

Most languages provide some form of **overloading**, where the same symbol means different things depending on the types to which it is applied. E.g., overloading of arithmetic operators to work on either integers or reals is very common.

Aim is to do “what we expect;” rules can get quite complicated (especially when **coercions** are considered) !

Some languages (e.g., Ada, C++) support **user-defined** overloading, normally for user-defined types (e.g. complex numbers).

In conventional languages, overloading is resolved **statically**; that is, the compiler selects the appropriate version of the operator once and for all at compiler time. (Different from object-oriented dynamic overriding; more later.)

Overloading is sometimes called “**ad-hoc polymorphism**”. It is fundamentally **different** from parametric polymorphism, because the implementation of the overloaded operator changes according to the underlying types.

ML has a well-conceived set of type constructs.

- Primitives: unit, int, word, real, char, string, exn, array, vector, ref.
- Record (tuple) types ($t_1 \times t_2$):

```
type emp = string * int (unlabeled fields)
val x : emp = ("abc",3)
type emp =
  {name: string, age: int} (labeled fields)
val x : emp = {name="abc",age=3}
```

Record values may be written without declaring an explicit named type first.

- Functions: $t_1 \rightarrow t_2$

All functions take just one argument; the effect of multi-argument functions can be obtained by passing a record or by Currying.

ML's **datatype** mechanism can be used to define many different useful types.

- Each datatype declaration defines a new type and specifies its **data constructors** (which take 0 or 1 arguments).
- Value of the type are taken apart using **pattern matching** in a case statement or function declaration.

Sums ($t_1 \oplus t_2$)

```
datatype temp = F of real
              | C of real
fun boiling (t:temp) : bool =
  case t of
    F r => r >= 212.0
  | C r => r >= 100.0
```

Can combine case into function definition, e.g.

```
fun boiling (F r) = r >= 212.0
  | boiling (C r) = r >= 100.0
```

DATATYPES (2)

Recursive types

```
datatype inttree = Branch of inttree * inttree
                 | Leaf of int
fun sumleaves (Leaf i) = i
  | sumleaves (Branch(l,r)) =
    (sumleaves l) + (sumleaves r)
```

Parameterized type constructors (polymorphic types)

```
datatype 'a bintree =
  Branch of 'a bintree * 'a bintree
  | Leaf of 'a
fun depth (Leaf _) = 0
  | depth (Branch(l,r)) = max(depth l,depth r) + 1
type inttree = int bintree
type booltree = bool bintree
```

(Type 'a list is just a special case of a parameterized type constructor, with extra syntactic sugar for writing literals.)

DATATYPES (3)

Enumerations

```
datatype day =
  Mon | Tue | Wed | Thu | Fri | Sat | Sun
fun weekday (d:day) : bool =
  case d of
    Sat => false
  | Sun => false
  | _ => true
```

(Type bool is just a special case of an enumeration.)

Singleton types

```
datatype complexR = CR of real * real
datatype complexP = CP of real * real
fun convert (CP (r,theta)) =
  CR(r*(cos theta),r*(sin theta))
val x : complexR = ...
... convert x ... (* STATIC TYPE ERROR ! *)
```

REPRESENTATION

TYPE ABBREVIATIONS

ML also has type **abbreviations**, which are introduced by type declarations. These simply serve to give convenient names to possibly lengthy types.

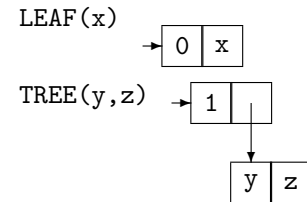
```
type t = int * bool
val x : t = (2,true)
fun f (a:int,b:bool) = ...
... f x ... (* TYPE-CHECKS FINE *)
```

Note that no new constructor name is involved.

Basic representation idea for user-defined datatypes: each value is represented boxed, more specifically as a two-element record, containing a **tag** field and a **contents** field (which may itself be a record).

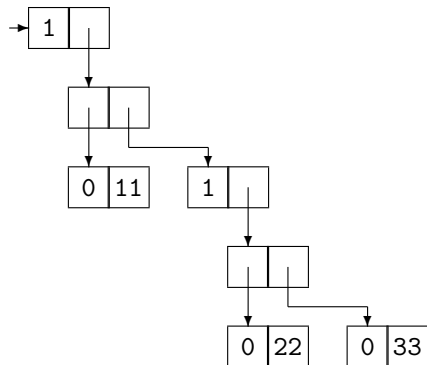
Example: Trees

```
datatype tree = LEAF of int
              | TREE of tree * tree
```



REPRESENTATION EXAMPLE

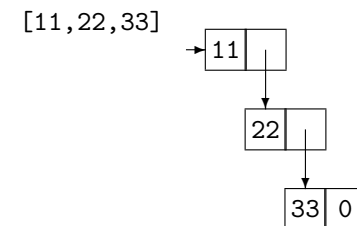
TREE(LEAF(11), TREE(LEAF(22), LEAF(33)))



STANDARD REPRESENTATION OPTIMIZATIONS

The above scheme is not very efficient for important special classes of datatypes, so in practice certain optimizations are used.

- **Nullary** constructors (like enumeration values) are represented as unboxed small integers.
- List values are represented without internal indirections:



Every value still occupies just one word (boxed or unboxed) This is an example of **uniform data representation**. ML implementations usually use this representation (although they are not required to). This makes it particularly easy to generate code for polymorphic functions.