CS558 F’03 Lecture Notes
Lecture 6
**Type Inference**

Depending on the type system, it is often possible for a language processor to perform **type inference** in addition to type checking.

- 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.

Simple Examples (assuming just `int` and `bool` base types):

```plaintext
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`.

```plaintext
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`. 
**Systematic Inference**

A harder example:

```plaintext
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.

Will need some additional typing rules for functions:

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

\[
\frac{TE \vdash e_1 : t_1 \rightarrow t_2 \quad TE \vdash e_2 : t_1}{TE \vdash e_1 e_2 : t_2} \quad \text{(Appl)}
\]
Inference Example

\[
\text{let fun } f \ x = \text{if } x \ \text{then } p \ \text{else} \ q \\
\text{in } 1 \ + \ (f \ r) \\
\text{end}
\]

**Node Rule Constraints**

<table>
<thead>
<tr>
<th>Node</th>
<th>Rule</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Let</td>
<td>( t_f = t_2 ) ( t_1 = t_7 )</td>
</tr>
<tr>
<td>2</td>
<td>Fn</td>
<td>( t_2 = t_x \rightarrow t_3 )</td>
</tr>
<tr>
<td>3</td>
<td>If</td>
<td>( t_4 = \text{bool} ) ( t_3 = t_5 = t_6 )</td>
</tr>
<tr>
<td>4</td>
<td>Var</td>
<td>( t_4 = t_x )</td>
</tr>
<tr>
<td>5</td>
<td>Var</td>
<td>( t_5 = t_p )</td>
</tr>
<tr>
<td>6</td>
<td>Var</td>
<td>( t_5 = t_q )</td>
</tr>
<tr>
<td>7</td>
<td>Add</td>
<td>( t_7 = t_8 = t_9 = \text{int} )</td>
</tr>
<tr>
<td>8</td>
<td>Int</td>
<td>( t_8 = \text{int} )</td>
</tr>
<tr>
<td>9</td>
<td>Appl</td>
<td>( t_{10} = t_{11} \rightarrow t_9 )</td>
</tr>
<tr>
<td>10</td>
<td>Var</td>
<td>( t_{10} = t_f )</td>
</tr>
<tr>
<td>11</td>
<td>Var</td>
<td>( t_{11} = t_r )</td>
</tr>
</tbody>
</table>

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:

```ocaml
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.)
Polymorphism

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 \texttt{head} has type \texttt{int list \to int}.

It is also perfectly sensible to write:

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

giving \texttt{head} the type \texttt{bool list \to bool}. Note that the definition of \texttt{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 \texttt{head} as \textbf{polymorphic}.

By default ML infers the most polymorphic possible type for every function. In this case, it would give \texttt{head} the type \texttt{‘a list \to ‘a}, where \texttt{‘a} (pronounced “alpha”) is implicitly universally quantified. Each use of \texttt{head} occurs at a particular \textbf{instance} of \texttt{‘a} (first at \texttt{bool}, then at \texttt{int}).
Parametric Polymorphism vs. Overloading

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.

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 Types

ML has a well-conceived set of type constructs.

- **Primitives**: unit, int, word, real, char, string, exn, array, vector, ref.

- **Record (tuple) types** \((t_1 \times t_2)\):

  ```ml
  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.

  **Type abbreviations** are introduced by `type` declarations:

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

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** \((t_1 \oplus t_2)\)

```ml
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.

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

**Recursive types**

```ml
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)

```ml
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.)

Enumerations

```ml
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

```ml
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

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

Example: Trees

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

- \text{LEAF}(x) \rightarrow 0 x
- \text{TREE}(y, z) \rightarrow 1 y z

\text{TREE}(\text{LEAF}(11), \text{TREE}(\text{LEAF}(22), \text{LEAF}(33)))

```
>1
>1
>1
>0
>0
>0
>1
>0
>0
>0
```
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 directly by small integers.

- List values are represented without internal indirections:

  \[
  [11, 22, 33] \rightarrow \begin{array}{c}
  11 \\
  22 \\
  33 \end{array}
  \]

  Every value still occupies just one word (directly or indirectly). 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.
Abstract Data Types

Is a new type name a genuinely new type, equivalent to the built-in types?

Ideally, to mimic the behavior of built-in types, user-defined types should have an associated set of operators, and it should only be possible to manipulate types via their operators (and maybe a few generic operators such as assignment or equality testing).

In particular, when new types are given a representation in terms of existing types, it shouldn’t be possible for programs to inspect or change the fields of the representation.

Such a type is called an abstract data type (ADT), because to clients (users) of the type, its implementation is hidden.

We can implement an ADT by combining a type definition together with a set of function operating on the type into a module (or package, cluster, class, etc.) Additional hiding features are needed to make the type’s representation more-or-less invisible outside the module.

When designing an ADT, it is simpler and more elegant to provide purely functional operators.
**Example: Environments in SML**

```ml
signature ENV =
sig
  type env
  val empty : env
  val extend : (env * string * int) -> env
  val lookup : (env * string) -> int option
end

structure Env :> ENV =
struct
  type env = (string * int) list

  val empty = nil
  fun extend (env,k,v) = (k,v)::env
  fun lookup ((k0,v0)::rest,k) = 
      if k = k0 then SOME v0
      else lookup (rest,k)
  | lookup (nil,k) = NONE
end (* Env *)
```

- ML also has a (non-module-based) `abstype` declaration form, but module form is just as powerful and more flexible.
Example: Environments in Java

class Env {
    private Link contents = null;
    private Env (Link c) {
        contents = c;
    }

    static final Env empty = new Env(null);

    Env extend(String k, int v) {
        return new Env(new Link(k,v,contents));
    }

    int lookup(String k) {
        Link c = contents;
        while (c != null) {
            if (c.key.equals(k))
                return c.value;
            else
                c = c.next;
        }
        return -1;
    }
}

- Note role of private constructor.
- Don’t confuse abstract data types with abstract classes!
Algebraic Specification

If clients are to be able to use an ADT without knowing anything about the implementation, they need a full specification of the operations’ behavior.

Type signatures give only a partial specification.

A standard approach is to add axioms describing the behavior of different combinations of axioms. Example:

ADT env
Signatures:
  empty : env
  extend : env * key * value -> env
  lookup : env * key -> value option

Axioms:
  lookup(empty, k₀) = NONE
  lookup(extend(e, k, v), k₀) =
      if k = k₀ then SOME v else lookup(e, k₀)

Another example:

ADT set
Signatures:
  empty : set
  insert : set * elem -> set
  union : set * set -> set
  member : set * elem -> bool

Axioms:
Choosing Axioms

How many axioms are enough?

We can identify two important subsets of operations:

- **constructors** return new instances of the ADT.

- **observers** (or **inspectors**) take one or more instances of the ADT as arguments and return some other type(s) as result.

Example: for the Env ADT, the constructors are `empty` and `extend`; the sole observer is `lookup`.

The only way to create an ADT value is to call a constructor. So every ADT value can be built up inductively by applying constructors.

The only aspect of an ADT value that matters is how it behaves when passed to an observer. (We can’t tell anything else about the value!)

So, it suffices if we give enough axioms to define the behavior of every observer on every possible constructor.
Implementations from Axioms

It turns out that we can often use the axioms to build an implementation “for free.” The idea is to represent each value of the ADT by the sequence of constructors used to build it.

The resulting implementation may not be very efficient, but it can be useful for prototyping. Example:

```
structure Env :> ENV = struct
  datatype env =
    EMPTY
  | EXTEND of env * string * int

  val empty = EMPTY
  fun extend (e,k,v) = EXTEND(e,k,v)
  fun lookup (EMPTY,k0) = NONE
  | lookup (EXTEND(e,k,v),k0) =
      if k = k0 then
        SOME v
      else
        lookup (e,k0)
  end (* Env *)
```
Observational Equivalence

We can use the axioms to prove the observational equivalence of two ADT values, even in cases where the representations of the values are different!

Example: suppose we have

\[ e_1 = \text{extend(extend (empty,"a",1), "b", 2)} \]
\[ e_2 = \text{extend(extend (empty,"b",2), "a", 1)} \]

Using the axioms, we can prove that, for any key \( k \),

\[ \text{lookup}(e_1, k) = \text{lookup}(e_2, k) \]

Hence \( e_1 \) and \( e_2 \) are observationally equivalent, even though they may have different representations (e.g. in the implementations we gave).

In conventional languages, axioms only have the status of comments. So reasoning using observational equivalence is dangerous unless we have proved that the actual implementation obeys the axioms; we can imagine systems that checked (or helped us check) this.
Interface vs. Implementation

Ideally, the client of an ADT is not supposed to know or care about its internal **implementation** details – only about its exported **interface**. Thus, it makes sense to separate the **textual** description of the interface from that of the implementation, e.g., into separate files.

For example, ML distinguishes **signatures** (module specifications) from **structures** (module bodies), and encourages them to be in separate files. Specifications give the names of types, and the names and types of functions in the package. Bodies give the definitions of the types and functions mentioned in the specification, and possibly additional private definitions.

One advantage of this separation is that clients of module X can be **compiled** on the basis of the information in the specification of X, without needing access to the the body of X (which might not even exist yet!)

Many languages, particularly in the C/C++ tradition, don’t make this separation very cleanly. Java doesn’t support it cleanly either (constructors are one sticking point).
Is abstraction always desirable?

Although the idea of defining explicitly all the operators for a type makes good logical sense, it can get quite inconvenient.

Programmers are used to assigning values or passing them as arguments without worrying about their types. They may also expect to be able to compare them, at least for equality, without regard to type.

So most languages that support ADT’s have built-in support for these basic operations, defined in a uniform way across all types. They also usually have facilities for overriding the built-in definitions with type-specific versions. (Some of the complexity of C++ derives from this.)

Unfortunately, it is impossible to generate code for operations that move or compare data without knowing things like the size and layout of the data. But these are characteristics of the type’s implementation, not its interface. So these “universal” operations break the abstraction barrier around types.

Thus, supporting these operations conflicts with separate compilation, often weakening support for the latter. The problem can also be solved, at some cost in efficiency, by storing all abstract values indirectly.