Type Inference and Static Analysis

CS 502
Lecture 4
9/2/08
Static Semantics

- Check for correctness
  - Beyond simple syntactic checking
- Enable optimization
  - Boxing and unboxing
  - Shape and points-to analysis
  - Optimized representations
- Enable improved code generation
Role of Types

- Type-checking
  - Catching errors
  - Types as contract between implementors and programmers
  - Static vs. dynamic
    - ML vs. Scheme
    - What about Java?

- Efficient compilation
  - Static checking guarantees certain properties
    - E.g., + only applied to integers
Types in Compilation

- Early examples
  - Fortran enforced strict separation between floating-point and integer operations.
  - Modern language implementations that support overloaded operations require some form of runtime type-test.

- Handling variable sized data:
  - In Pascal, array sizes were part of an array’s type.
  - Static layout of size information is necessary to generate efficient code.
    - What happens when size information is unknown?
    - May need to “box” data
Guarantees provided by types

- Types provide guarantees useful to an implementation
  - Two pointers with incompatible types cannot alias one another
  - Loads and stores on objects of different types can be segregated
  - Supports more aggressive code motion
  - Optimizing method lookup in an object-oriented language
    - If dynamic type of a class is known, replace expensive lookup with direct jump
  - Only objects that are references or pointers need to be traversed by a garbage collector.
Types at Runtime

- Scheme requires runtime type tests
  - E.g., is car applied to a list?

- Object-oriented languages support runtime type inspection:
  - Needed to support downcasts
  - Type hierarchy must be exposed in the underlying implementation

- Handling polymorphic equality in ML:
  - Equality applies to objects of many different types

- Compilers must often propagate information to runtime:
  - Distinguish between pointers and integers
Type Checking

☐ Strong vs. Weak:
- Are type errors allowed to go undetected?

☐ Static vs. Dynamic:
- Are type errors allowed to manifest at runtime?

☐ We are concerned with implementing type checkers that provide strong, static typing.
Case Study: Polymorphism

- Polymorphism means “having many forms”
- Ad-hoc polymorphism:
  - Overloading + to operate over strings and numbers
  - Parameteric polymorphism: programs parameterized over types
    - Explicit
    - Implicit
Polymorphism

- **Explicit:**
  - Type parameters in function definition
    ```
    let val id = fn(t:type) => fn(a:t) => a
    in id(int) 3; id(bool) true
    ```
  - Requires having values of type “type”
  - Very powerful, but type system is complex

- **Implicit:**
  - Type parameters not admitted.
  - Instead types can contain type variables which represent unknown types.
Implicit Polymorphism

- Example:
  - let val id = fn(a:'a) => a
    in id(3); id(true)

- Implicit polymorphism is a restricted form of explicit polymorphism
  - Types must be discovered to recover information lost by omitting type parameters
Inference

- If we omit type parameters, we must discover whether the intended use of an expression matches its actual use.

- Implications for compilation:
  - How do we generate code for a polymorphic procedure that may be applied to objects with very different representations?

- First need to understand how inference works.
Type Variables

- In ML, a type can be a type variable
  - ‘a, ‘b,….
  - a type operator (int, bool, int -> int, …)

- Types containing type variables are polymorphic; otherwise they are monomorphic.

- How do we characterize languages like Pascal? Java?
Type-checking

- Match type operators and instantiate type variables.
- Need to define where type variables can appear.
- Must also enforce contextual dependencies:
  - 'a → 'a : substituting “int” for 'a must be done uniformly for all occurrences of the type variable in the type.
Type-checking

- Perform context-sensitive type instantiation using unification.
  - Unification fails when
    - trying to match two distinct type operators (int and bool)
    - instantiating a type variable to a term containing that variable ('a and 'a → int)
  - Example: try to type-check the following expression:
    \( \text{fn } x \Rightarrow x(x) \)
Constraint-Based Typing

- constraints define equations between type expressions that may contain type variables
- Type inference rules calculate types (and their constraints)
- Validate the correctness of a given set of constraints under a substitution (a mapping between type variables and types)
Substitutions

- A substitution $\sigma$ is a mapping from type variables to types. A context is a list of variables to their types. Given a context $\Gamma$ and an expression $t$, a solution is a pair $(\sigma, T)$ where $T$ is a type such that type checking $\sigma t$ in the context $\sigma \Gamma$ yields $T$.

Example:

Context: $f: X, a: Y$

Term $t$: $f(a)$

Possible solutions:

$([X \mapsto Y \rightarrow \text{int}], \text{int})$

$([X \mapsto Y \rightarrow Z], Z)$

$([X \mapsto Y \rightarrow \text{Z}, Z \mapsto \text{int}], Z)$

$([X \mapsto Y \rightarrow \text{int} \rightarrow \text{int}], \text{int} \rightarrow \text{int})$
Constraints

- Constraints defined by semantics of language constructs.
- Goal of type inference is to find a substitution of type variables to types that satisfy generated constraints.
- Example:

  Expression \( t = \text{fn}(x: 'a \rightarrow 'b). x(0) \)

  Constraint set: \( \text{int} \rightarrow 'c = 'a \rightarrow 'b \)

  Substitution: \[ 'a \mapsto \text{int}, 'b \mapsto \text{bool}, 'c \mapsto \text{bool} \]

  Goal is to find most general unifier for a given set of constraints.
Unification

- Allows us to calculate a solution (most general) to a constraint set:

\[
\text{unify}(C) = \\
\text{if } c \text{ is empty then } [] \\
\text{else let } \{ S = T \} U C' = C \text{ in} \\
\quad \text{if } S = T \\
\qquad \text{then unify}(C') \\
\quad \text{else if } S = X \text{ and } X \not\in \text{FV}(T) \\
\qquad \text{then unify}([X \rightarrow T]C') \circ [X \rightarrow T] \\
\quad \text{else if } T = X \text{ and } X \not\in \text{FV}(S) \\
\qquad \text{then unify}([X \rightarrow S]C') \circ [X \rightarrow S] \\
\quad \text{else if } S = S1 \rightarrow S2 \text{ and } T = T1 \rightarrow T2 \\
\qquad \text{then unify}(C' U \{ S1 = T1, S2 = T2 \}) \\
\quad \text{else fail}
\]
Type-checking

- Perform context-sensitive type instantiation using unification.
  - Unification fails when
    - trying to match two distinct type operators (int and bool)
    - instantiating a type variable to a term containing that variable ('a and 'a → int)
  - Example: try to type-check the following expression:
    \[ \text{fn } x \Rightarrow x(x) \]
Example

The type of length in the following program:

```
let fun length l = if (null l)
    then 0
    else succ(length(tl(l)))
in  ...
```

is 'a list → int. How does the ML typechecker deduce this type?

Perform a bottom-up inspection of the program, matching and synthesizing types while proceeding to the root.

- The type of an expression is computed from the type of its subexpressions and the type constraints imposed by the context.
- Important property: order in which we examine programs and perform unification does not affect final result.
Consider the type of `length`. Perform type-checking using a bottom-up derivation:

```plaintext
let fun length l = if (null l)
  then 0
  else succ(length(tl(l)))
in ...
```

**Example (cont)**

Consider the type of `length`. Perform type-checking using a bottom-up derivation:

<table>
<thead>
<tr>
<th>l:</th>
<th>'a</th>
<th>type of l initially unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>null:</td>
<td>'b list -&gt; bool</td>
<td>definition of null</td>
</tr>
<tr>
<td>null(l):</td>
<td>bool</td>
<td>by definition of null</td>
</tr>
<tr>
<td>0:</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>tl:</td>
<td>'c list -&gt; 'c list</td>
<td>by definition of tl</td>
</tr>
<tr>
<td>tl(l):</td>
<td>'c list</td>
<td>unification</td>
</tr>
<tr>
<td>l:</td>
<td>'c list</td>
<td></td>
</tr>
</tbody>
</table>
let fun length l = if (null l) then 0 else succ(length(tl(l))) in ...

<table>
<thead>
<tr>
<th></th>
<th>'a → 'd</th>
<th>by definition of fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>'d</td>
<td></td>
</tr>
<tr>
<td>length(tl(l))</td>
<td>'a = 'c list</td>
<td>unification</td>
</tr>
<tr>
<td>succ</td>
<td>int → int</td>
<td>by definition of succ</td>
</tr>
<tr>
<td>succ(length(..))</td>
<td>int</td>
<td>unification</td>
</tr>
<tr>
<td>if (null...)</td>
<td>'d = int</td>
<td>by definition of conditional</td>
</tr>
<tr>
<td>fn l =&gt; ...</td>
<td>'c list → int</td>
<td></td>
</tr>
</tbody>
</table>
Basic algorithm

1. A variable $x$ introduced as a function argument assigned a new type variable. Store $<x,'a>$ in a type environment, where '$a$ is fresh.

2. In a conditional, predicate type unified with bool, the true and false branch unified with one another. This type (call it '$b$) is the type of the conditional.

3. The type of $e$ in a function $\text{fn x => e}$ is inferred in a context where $x$ is associated with a new type variable.

4. In an application, $(f \ x)$, $f$ is unified against $A \rightarrow 'b$ where $A$ is the type of $x$ and '$b$ is a new type variable.

   The type of $f$ is therefore a function type whose domain is unifiable to '$b$. '$b$ (or its instantiation) is returned as the type of the function.
To type-check let expressions introduce notion of genericity.

- What is the type of the expression:
  - `fn f => (f(3), f(true))`
  - Cannot type this in ML because f’s type is considered non-generic.
  - The first occurrence of f determines a type `int → 'a`, and the second determines a type `bool → 'a`.
  - Can’t unify these two terms

- Non-generic type variables cannot be instantiated multiple times within their defined context.
- To implement generic types, make a copy of the type for every distinct context in which it occurs.
Algorithm (cont)

- What about:

```plaintext
let val f = fn x => x

in (f(3), f(true))

end
```

- Here, we will assign f type `'a \rightarrow `'a` and view `'a` as generic:
  - `'a` can assume different values for different instantiations of f in the let-body.
Algorithm (cont)

- Need to be careful to not copy non-generic variables:
  
  ```ml
  let val f = fn g => let val h = g
  in pair(h(3),h(true))
  end
  
  in ...
  
  end
  ```

Def. A type variable occurring in the type of an expression e is generic (with respect to e) iff it does not occur in the type of the binder of any function definition enclosing e.
Algorithm (cont)

1. To typecheck a let expression, typecheck its declaration, obtaining an environment of identifiers and types used to typecheck the let-body.

2. Recursive definitions:
   
   ```
   let fun f(...) = ... f ....
   in ... f ....
   ```

   Instances of the type variable in the recursive definition must be non-generic, while instances in the body are generic.
Issues

- In the presence of imperative features, type inference algorithm just described would lead to incorrect inferences:

```haskell
let val r = ref (fn x => x)
in (r := (fn x => x + 1);
   !r true)
end
```

Would infer a type for r: 'a -> 'a ref.
What are the implications?
Value Restriction

- Approximate when it is safe to generalize type variables:
  - Use syntactic structure
  - Define a notion of expansiveness.
  - A non-expansive variable:
    - constants
    - nullary constructors
    - variables
    - function expressions (fn x => x)
Value Restriction

- All other expression including function applications, let expressions, conditionals are expansive:
  - Their evaluation may entail non-trivial computation that may lead to the creation of ref cells.
  - Value polymorphism:
    - given val pat = exp
    - types of variables appearing in pat can be closed by generalizing those type variables appearing free in their types but not in the context if, and only if, the expression on the right-hand side is non-expansive.
Value Restriction

- Example revisited:
  - The expression `val r = ref (fn x => x)`
- The right-hand side is expansive and the type variables associated with `x` cannot be generalized.