Lecture 14: Type Checking & Polymorphism

CSCI 131
Fall, 2008

Kim Bruce

ML Type Inference

1. An identifier should be assigned the same type throughout its scope.
2. In an "if-then-else" expression, the condition must have type boolean and the "then" and "else" portions must have the same type. The type of the expression is the type of the "then" and "else" portions.
3. A user-defined function has type 'a → 'b, where 'a is the type of the function's parameter and 'b is the type of its result.
4. In a function application of the form f x, there must be types T and U such that f has type T → U and x has type T, and the application itself has type U.

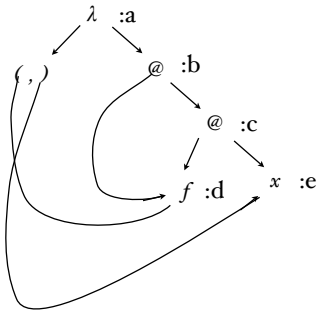
Examples of Type Inference

- Use rules to deduce types:

```
val map = fn f => fn l =>
            if l = [] then []
                 else (f (hd l)):: (map f (tl l))
```

- map: 'a → 'b *because function*
- f: 'a, `fn l => ...` : 'b, Thus 'b = 'c → 'd
- l: 'c, `if l = [] then ...` : 'd
- ...

λ = fn
@ = application




```
fun double(f,x) = f(f(x))
```
or equivalently
```
val double = fn (f,x) => f(f(x))
```

Outcome of Type Inference

- Overconstrained: no solution

```
- tl 7;
stdIn:22.1-22.5 Error: operator and operand don't
agree [literal]
operator domain: 'Z list
operand: int
in expression:
tl 7
```

- Underconstrained: polymorphic
- Uniquely determined

By the way, ...

- SML type inference is doubly exponential in the worst case!
- Can write down terms t_n such that the length of the type of t_n is of length 2^{2^n}

Restrictions on ML Polymorphism

- Type ('a → 'b) →('a list → 'b list) stands for:
  - ∀'a. ∀'b. ('a → 'b) →('a list → 'b list)
- ML functions may not take polymorphic arguments. E.g., no type:
  - ∀'a. (∀'b.'a → 'b) →('a list → 'b list)
  - fun double f x = f (f x);
  - val dbleTl = double tl;
  - results in error "value restriction" (avoid issues w/ polymorphic references)
  - fun dbleTl lst = double tl lst; OK

Restrictions on Implicit Polymorphism

Polymorphic types can be defined at top level or in let clauses, but can't be used as arguments of functions:

```
let fun id x = x
   in (id "ab", id 17) end;
```

is fine, but can't write

```
let fun test g = (g [], g "ab")
   in test (fn x => x) end;
```

Can't find type of test w/unification.
More general type inference is undecidable.

Explicit Polymorphism

Easy to type w/ explicit polymorphism:

```
let
  fun test (g: forall t.t -> t): (int list, string) =
                   (g (int list) [], g string "ab")
   in
      test (Fn t => fn (x:t) => x)
   end;
```

Languages w/explicit polymorphism:
Clu, Ada, C++, Eiffel, Java 5

Explicit Polymorphism

- Clu, Ada, C++, Java
- C++ macro expanded at link time rather than compile time.
- Java compiles away polymorphism, but checks it statically.

Summary

- Modern tendency: strengthen typing & avoid implicit holes, but leave explicit escapes
- Push errors closer to compile time by:
  - Require over-specification of types
  - Distinguishing between different uses of same type
  - Mandate constructs that eliminate type holes
  - Minimizing or eliminating explicit pointers
- Holy grail: Provide type safety, increase flexibility

Lexing

- Lexer returns a list of all tokens from the input stream.
- Build from either regular expressions or (equivalently) finite automaton recognizing the tokens.
- See program in class examples.
- ML program used signatures and structures.
  - Used by ML to group together related operations
  - Used by ML to hide details of implementation
    - signature filters out functions