

CS560

LECTURE 5
Feb 3, 2005

Slides: From John Mitchell, Concepts in Programming Languages, 2003.

Announcements

- ◆ Homework 2 due next week
 - [Question about due date.](#)
- ◆ Term project part 1, due today.
- ◆ Homework 3 will be posted week 7, due week 9
- ◆ Midterm 2/10 i.e., next week
- ◆ Any questions?

Overview of Class Tonight

- ◆ Types
 - [Type safety](#)
 - [Type checking](#)
 - [Type inference](#)
- ◆ Scope
 - [scope/lifetime](#)
 - [activation records](#)
 - [parameter passing](#)
 - [globals variables](#)
- ◆ Chapter 6 and 7 (7.1 – 7.3.3)

Type

A type is a collection of computable values that share some structural property.

- | | |
|--|--|
| ◆ Examples | ◆ "Non-examples" |
| <ul style="list-style-type: none">• Integers• Strings• int → bool• (int → int) → bool | <ul style="list-style-type: none">• {3, true, λx.x}• Even integers• {f:int → int if x>3 then f(x) > x*(x+1)} |

Distinction between types and non-types is language dependent.

Uses for types

- ◆ Program organization and documentation
 - Separate types for separate concepts
 - Represent concepts from problem domain
 - Indicate intended use of declared identifiers
 - Types can be checked, unlike program comments
- ◆ Identify and prevent errors
 - Compile-time or run-time checking can prevent meaningless computations such as `3 + true` - "Bill"
- ◆ Support optimization
 - Example: short integers require fewer bits
 - Access record component by known offset

Type errors

- ◆ Hardware error
 - function call `x()` where `x` is not a function
 - may cause jump to instruction that does not contain a legal op code
- ◆ Unintended semantics
 - `int_add(3, 4.5)`
 - not a hardware error, since bit pattern of float 4.5 can be interpreted as an integer
 - just as much an error as `x()` above

General definition of type error

- ◆ A *type error* occurs when execution of program is not faithful to the intended semantics
- ◆ Do you like this definition?
 - Store 4.5 in memory as a floating-point number
 - Location contains a particular bit pattern
 - To interpret bit pattern, we need to know the type
 - If we pass bit pattern to integer addition function, the pattern will be interpreted as an integer pattern
 - Type error if the pattern was intended to represent 4.5

Compile-time vs run-time checking

- ◆ Lisp uses run-time type checking
 - (`car x`) check first to make sure `x` is list
- ◆ ML uses compile-time type checking
 - `f(x)` must have `f : A → B` and `x : A`
- ◆ Basic tradeoff
 - Both prevent type errors
 - Run-time checking slows down execution
 - Compile-time checking restricts program flexibility
 - Lisp list: elements can have different types
 - ML list: all elements must have same type

Relative type-safety of languages

- ◆ Not safe: BCPL family, including C and C++
 - Casts, pointer arithmetic
- ◆ Almost safe: Algol family, Pascal, Ada.
 - Dangling pointers.
 - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p
 - No language with explicit deallocation of memory is fully type-safe
- ◆ Safe: Lisp, ML, Smalltalk, and Java
 - Lisp, Smalltalk: dynamically typed
 - ML, Java: statically typed

Type checking and type inference

- ◆ Standard type checking


```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2;};
```

 - Look at body of each function and use declared types of identifiers to check agreement.
 - ◆ Type inference


```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2;};
```

 - Look at code without type information and figure out what types could have been declared.
- ML is designed to make type inference tractable.

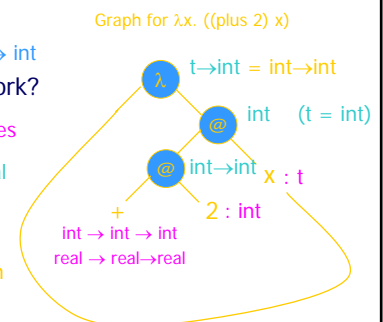
ML Type Inference

- ◆ Example
 - fun f(x) = 2+x;
 - > val it = fn : int → int
- ◆ How does this work?
 - + has two types: $\text{int} * \text{int} \rightarrow \text{int}$, $\text{real} * \text{real} \rightarrow \text{real}$
 - 2 : int has only one type
 - This implies + : $\text{int} * \text{int} \rightarrow \text{int}$
 - From context, need x: int
 - Therefore $f(x:\text{int}) = 2+x$ has type $\text{int} \rightarrow \text{int}$

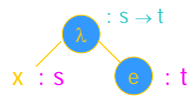
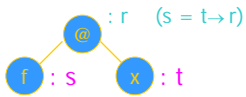
Overloaded + is unusual. Most ML symbols have unique type. In many cases, unique type may be polymorphic.

Another presentation

- ◆ Example
 - fun f(x) = 2+x;
 - > val it = fn : int → int
- ◆ How does this work?
 - Assign types to leaves
 - Propagate to internal nodes and generate constraints
 - Solve by substitution



Application and Abstraction



◆ Application

- f must have function type
domain → range
- domain of f must be type of argument x
- result type is range of f

◆ Function expression

- Type is function type
domain → range
- Domain is type of variable x
- Range is type of function body e

Types with type variables

◆ Example

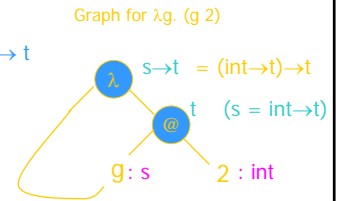
- fun f(g) = g(2);
- > val it = fn : (int → t) → t

◆ How does this work?

Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution



Use of Polymorphic Function

◆ Function

- fun f(g) = g(2);
- > val it = fn : (int → t) → t

◆ Possible applications

- | | |
|---------------------------|----------------------------|
| - fun add(x) = 2+x; | - fun isEven(x) = ...; |
| > val it = fn : int → int | > val it = fn : int → bool |
| - f(add); | - f(isEven); |
| > val it = 4 : int | > val it = true : bool |

Recognizing type errors

◆ Function

- fun f(g) = g(2);
- > val it = fn : (int → t) → t

◆ Incorrect use

- fun not(x) = if x then false else true;
- > val it = fn : bool → bool
- f(not);

Type error: cannot make $\text{bool} \rightarrow \text{bool} = \text{int} \rightarrow \text{t}$

Another Type Inference Example

◆Function Definition

- fun f(g,x) = g(g(x));
 > val it = fn : (t → t)*t → t

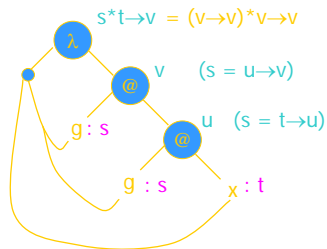
Graph for $\lambda(g,x). g(g\ x)$

◆Type Inference

Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution



Main Points about Type Inference

◆Compute type of expression

- Does not require type declarations for variables
- Find *most general type* by solving constraints
- Leads to polymorphism

◆Static type checking without type specifications

◆May lead to better error detection than ordinary type checking

- Type may indicate a programming error even if there is no type error (example following slide).

Information from type inference

◆An interesting function on lists

```
fun reverse (nil) = nil
  | reverse (x::lst) = reverse(lst);
```

◆Most general type

reverse : 'a list → 'b list

◆What does this mean?

Since reversing a list does not change its type, there must be an error in the definition of "reverse"

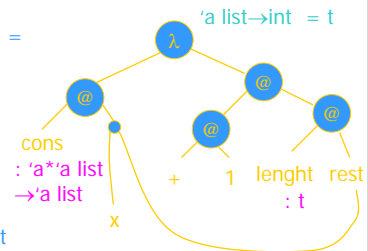
Type inference with recursion

◆Second Clause

```
length(cons(x,rest)) =
  1 + length(rest)
```

◆Type inference

- Assign types to leaves, including function name
- Proceed as usual
- Add constraint that type of function body = type of function name



Exercise

- ◆ `fun f(g) = g(1) + 2;`
- ◆ `fun sum(x) = x + sum(x-1);`

Polymorphism vs Overloading

◆ Parametric polymorphism

- Single algorithm may be given many types
- Type variable may be replaced by *any* type
- `f : t → t` => `f : int → int, f : bool → bool, ...`

◆ Overloading

- A single symbol may refer to more than one algorithm
- Each algorithm may have different type
- Choice of algorithm determined by type context
- Types of symbol may be arbitrarily different
- `+` has types `int*int → int, real*real → real, no others`

Parametric Polymorphism: ML vs C++

◆ ML polymorphic function

- Declaration has no type information
- Type inference: type expression with variables
- Type inference: substitute for variables as needed

◆ C++ function template

- Declaration gives type of function arg, result
- Place inside template to define type variables
- Function application: type checker does instantiation

Example: swap two values

◆ ML

```
- fun swap(x,y) =  
    let val z = !x in x := !y; y := z end;  
val swap = fn : 'a ref * 'a ref -> unit
```

◆ C++

```
template <typename T>  
void swap(T& , T& y){  
    T tmp = x; x=y; y=tmp;  
}
```

Declarations look similar, but compiled very differently

Implementation

◆ML

- Swap is compiled into one function
- Typechecker determines how function can be used

◆C++

- Swap is compiled into linkable format
- Linker duplicates code for each type of use

◆Why the difference?

- ML ref cell is passed by pointer, local x is pointer to value on heap
- C++ arguments passed by reference (pointer), but local x is on stack, size depends on type

Another example

◆C++ polymorphic sort function

```
template <typename T>
void sort( int count, T * A[] ) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```

◆What parts of implementation depend on type?

- Indexing into array
- Meaning and implementation of <

Overloading

- ◆ A symbol has 2 or more meanings
- ◆ the compiler resolves the correct binding based on context
 - C++, Ada, Haskell, ML allow overloading function names & operators
 - Java function names only
- ◆ How is overloading handled by compiler/interpreter?

```
int max(int x, int y) //max #1
```

```
float max(int x, int y) //max #2
```

Type Equivalence

- ◆ Type equivalence is defined in two principal ways
 - Structural equivalence
 - Name equivalence
- ◆ Two types are **structurally equivalent** if they have identical type structures
 - They must have the same components
 - E.g. C
- ◆ Two type are **nominally equivalent** if they have the same name
 - It may or may not include an alias
 - E.g. Java

Type Equivalence: Structural Equivalence

- ◆ Is the order in the declaration relevant?

```
record foo1 {  
    int a;  
    int b;  
}
```

```
record foo2 {  
    int b;  
    int a;  
}
```

- ◆ Most languages consider these two types structurally equivalent

Type Equivalence: Structural Equivalence Pitfall

- ◆ Are these two types structurally equivalent?

```
record student {  
    char *name;  
    char *address;  
    int age;  
}
```

```
record school {  
    char *name;  
    char *address;  
    int age;  
}
```

- ◆ They are, and it is unlikely that the programmer intended to make these two types equivalent

Type Equivalence: Name Equivalence

- ◆ Assumes type definitions with different names are not equivalent
 - Otherwise, why did the programmer create two definitions?
- ◆ It solves the previous problem
 - `student` and `school` are not nominally equivalent
- ◆ Aliases:
 - Under *strict name equivalence*, aliases are not equivalent
 - type `A = B` is a definition (*i.e.* new object)
 - Under *loose name equivalence*, aliases are equivalent
 - type `A = B` is a declaration (*i.e.* binding)

Type Equivalence Example

```
record RecA {  
    char x;  
    int y;  
};
```

Which are structurally equiv?
Which are name equiv?

```
record RecB {  
    char x;  
    int z;  
};
```

Main Points about ML

- ◆ General-purpose procedural language
 - We have looked at “core language” only
 - Also: abstract data types, modules, concurrency,....
- ◆ Well-designed type system
 - Type inference
 - Polymorphism
 - Reliable -- no loopholes
 - Limited overloading

Chapter 7: Scope