

## CS560

---

LECTURE 4  
Jan 27, 2005

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

## Announcements

---

- ◆ Homework 1 due
- ◆ Term project part 1, due next week
- ◆ Homework 2 is posted, due 2/10
- ◆ Midterm 2/10, discuss more next week
- ◆ Any questions?

## Overview of Class Tonight

---

- ◆ Discussion left over from Last week.
  - Imperative vs Functional Languages
- ◆ Algol Family of Languages
  - Algol, C, Pascal, ML
- ◆ Intro to ML
- ◆ Types
  - Type safety
  - Type checking
  - Type inference

## Functional vs Imperative Programs

---

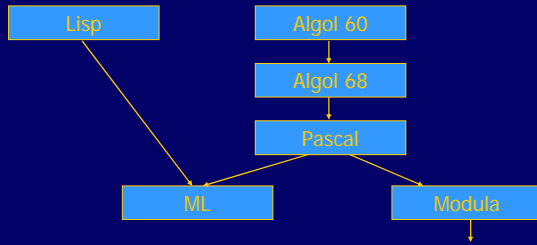
### Assumptions:

Functional languages support higher-order functions, garbage collection, no assignment.

Imperative languages support assignment, but not higher-order functions or GC.

1. Are there inherent reasons why imperative languages are superior to functional ones for the majority of programming tasks?
2. Which is easier to implement on a machine with limited disk and memory?
3. Which require bigger executables when compiled?
4. What consequences might these facts had in the early days of computing?
5. Are these concerns still valid?

## Language Sequence



Many other languages:  
 Algol 58, Algol W, Euclid, EL1, Mesa (PARC), ...  
 Modula-2, Oberon, Modula-3 (DEC)

## Algol 60

### ◆ Basic Language of 1960

- Simple imperative language + functions
- Successful syntax, BNF -- used by many successors
  - statement oriented
  - Begin ... End blocks (like C { ... } )
  - if ... then ... else
- Recursive functions and stack storage allocation
- Fewer ad hoc restrictions than Fortran
  - General array references:  $A[x + B[3]*y]$
- Type discipline was improved by later languages
- Very influential but not widely used in US

## Algol 60 Sample

```

real procedure average(A,n);
  real array A; integer n;
  begin
    real sum; sum := 0;
    for i = 1 step 1 until n do
      sum := sum + A[i];
    average := sum/n;
  end;

```

no array bounds

no ; here

set procedure return value by assignment

## Algol Joke

### ◆ Question

- Is  $x := x$  equivalent to doing nothing?

### ◆ Interesting answer in Algol

```

integer procedure p;
begin
  ....
  p := p;
  ....
end;

```

- Assignment here is actually a recursive call

## Some trouble spots in Algol 60

- ◆ Type discipline improved by later languages
  - parameter types can be array
    - no array bounds
  - parameter type can be procedure
    - no argument or return types for procedure parameter
- ◆ Parameter passing methods
  - Pass-by-name had various anomalies
    - “Copy rule” based on substitution, interacts with side effects
  - Pass-by-value expensive for arrays
- ◆ Some awkward control issues
  - goto out of block requires memory management

## Algol 60 Pass-by-name

- ◆ Substitute text of actual parameter
  - Unpredictable with side effects!
- ◆ Example

```
procedure inc2(i, j);
integer i, j;
begin
  i := i+1;
  j := j+1
end;
inc2 (k, A[k]);
```

→

```
begin
  k := k+1;
  A[k] := A[k] + 1
end;
```

Is this what you expected?

## Algol 68

- ◆ Considered difficult to understand
  - Idiosyncratic terminology
    - types were called “modes”
    - arrays were called “multiple values”
  - vW grammars instead of BNF
    - context-sensitive grammar invented by A. van Wijngaarden
  - Elaborate type system
  - Complicated type conversions
- ◆ Fixed some problems of Algol 60
  - Eliminated pass-by-name
- ◆ Not widely adopted



## Algol 68 Modes

- ◆ Primitive modes
    - int
    - real
    - char
    - bool
    - string
    - compl (complex)
    - bits
    - bytes
    - sema (semaphore)
    - format (I/O)
    - file
  - ◆ Compound modes
    - arrays
    - structures
    - procedures
    - sets
    - pointers
- Rich and structured type system is a major contribution of Algol 68

## Other features of Algol 68

- ◆ **Storage management**
  - Local storage on stack
  - Heap storage, explicit alloc and garbage collection
- ◆ **Parameter passing**
  - Pass-by-value
  - Use pointer types to obtain Pass-by-reference
- ◆ **Assignable procedure variables**
  - Follow “orthogonality” principle rigorously

Source: Tanenbaum, Computing Surveys

## Pascal

- ◆ **Revised type system of Algol**
  - Good data-structuring concepts
    - records, variants, subranges
  - More restrictive than Algol 60/68
    - Procedure parameters cannot have procedure parameters
- ◆ **Popular teaching language**
- ◆ **Simple one-pass compiler**

## Limitations of Pascal

- ◆ **Array bounds part of type**
  - procedure p(a : array [1..10] of integer)
  - procedure p(n: integer, a : array [1..n] of integer)

illegal

  - **Attempt at orthogonal design backfires**
    - parameter must be given a type
    - type cannot contain variables

How could this have happened? Emphasis on teaching
- ◆ **Not successful for “industrial-strength” projects**
  - Kernighan -- Why Pascal is not my favorite language
  - Left niche for C; niche has expanded!!

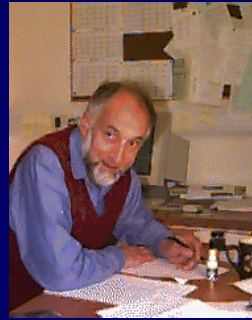
## ML

- ◆ **Typed programming language**
- ◆ **Intended for interactive use**
- ◆ **Combination of Lisp and Algol-like features**
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- ◆ **General purpose non-C-like, not OO language**

## Goals in study of ML

- ◆ Survey a modern procedural language
- ◆ Discuss general programming languages issues
  - Types and type checking
    - General issues in static/dynamic typing
    - Type inference
    - Polymorphism and Generic Programming
  - Memory management
    - Static scope and block structure
    - Function activation records, higher-order functions
  - Control
    - Force and delay
    - Exceptions
    - Tail recursion and continuations

## History of ML



- ◆ Robin Milner
- ◆ Logic for Computable Functions
  - Stanford 1970-71
  - Edinburgh 1972-1995
- ◆ Meta-Language of the LCF system
  - Theorem proving
  - Type system
  - Higher-order functions

## Logic for Computable Functions

- ◆ Dana Scott, 1969
  - Formulate logic for proving properties of typed functional programs
- ◆ Milner
  - Project to automate logic
  - Notation for programs
  - Notation for assertions and proofs
  - Need to write programs that find proofs
    - Too much work to construct full formal proof by hand
  - Make sure proofs are correct

## LCF proof search

- ◆ Tactic: function that tries to find proof

$$\text{tactic}(\text{formula}) = \begin{cases} \text{succeed and return proof} \\ \text{search forever} \\ \text{fail} \end{cases}$$

- ◆ Express tactics in the Meta-Language (ML)
- ◆ Use type system to facilitate correctness

## Tactics in ML type system

- ◆ Tactic has a functional type  
tactic : formula  $\rightarrow$  proof
- ◆ Type system must allow "failure"

tactic(formula) =  $\left\{ \begin{array}{l} \text{succeed and return proof} \\ \text{search forever} \\ \text{fail and raise exception} \end{array} \right.$

## Function types in ML

$f : A \rightarrow B$  means  
for every  $x \in A$ ,

$f(x) = \left\{ \begin{array}{l} \text{some element } y=f(x) \in B \\ \text{run forever} \\ \text{terminate by raising an exception} \end{array} \right.$

In words, "if  $f(x)$  terminates normally, then  $f(x) \in B$ ."  
Addition never occurs in  $f(x)+3$  if  $f(x)$  raises exception.

This form of function type arises directly from motivating application for ML. Integration of type system and exception mechanism mentioned in Milner's 1991 Turing Award.

## Higher-Order Functions

- ◆ Tactic is a function
- ◆ Method for combining tactics is a function on functions
- ◆ Example:

$f(\text{tactic}_1, \text{tactic}_2) =$   
 $\lambda \text{ formula. try tactic}_1(\text{formula})$   
 $\quad \text{else tactic}_2(\text{formula})$

## Basic Overview of ML

- ◆ Interactive compiler: *read-eval-print*
  - Compiler infers type before compiling or executing  
Type system does not allow casts or other loopholes.
- ◆ Examples
  - (5+3)-2;
  - > val it = 6 : int
  - if 5>3 then "Bob" else "Fido";
  - > val it = "Bob" : string
  - 5=4;
  - > val it = false : bool

## Declarations

```
--- val <id> = <expression>;  
val <id> = <printvalue> :<type>
```

```
--- fun <id> <args> = <expression>;  
val <id> = fn <argtype> → <resulttype>
```

Identifiers versus variables

- value of identifiers cannot be changed by assignment
- treatment of identifiers in C

## Overview by Type

### ◆ Booleans

- true, false : bool
- if ... then ... else ... (types must match)

### ◆ Integers

- 0, 1, 2, ... : int
- +, \*, ... : int \* int → int and so on ...

### ◆ Strings

- "Austin Powers"

### ◆ Reals

- 1.0, 2.2, 3.14159, ... decimal point used to disambiguate

## Compound Types

### ◆ Tuples

- (4, 5, "noxious") : int \* int \* string

### ◆ Lists

- nil
- 1 :: [2, 3, 4] infix cons notation

### ◆ Records

- {name = "Fido", hungry=true}  
: {name : string, hungry : bool}

## Patterns and Declarations

### ◆ Patterns can be used in place of variables

<pat> ::= <var> | <tuple> | <cons> | <record> ...

### ◆ Value declarations

- General form  
val <pat> = <exp>

#### • Examples

```
val myTuple = ("Conrad", "Lorenz");  
val (x,y) = myTuple;  
val myList = [1, 2, 3, 4];  
val x::rest = myList;
```

#### • Local declarations

```
let val x = 2+3 in x*4 end;
```

## Functions and Pattern Matching

### ◆ Anonymous function

- `fn x => x+1;`      like Lisp lambda

### ◆ Declaration form

- `fun <name> <pat1> = <exp1>`  
| `<name> <pat2> = <exp2> ...`  
| `<name> <patn> = <expn> ...`

### ◆ Examples


- `fun f (x,y) = x+y;`      actual par must match pattern (x,y)
- `fun length nil = 0`  
| `length (x::s) = 1 + length(s);`

## Exercise

### ◆ What does this function do?

- Show an example call to the function.


```
fun myfun (f, nil) = nil
|   myfun (f, x::xs) = f(x) :: myfun (f,xs);
```

```
myfun (fn x => x+1, [1,2,3]);       [2,3,4]
```

## Exercise

### ◆ Apply function to every element of list

```
fun map (f, nil) = nil
|   map (f, x::xs) = f(x) :: map (f,xs);
```

```
map (fn x => x+1, [1,2,3]);       [2,3,4]
```

## More functions on lists

### ◆ Reverse a list

```
fun reverse nil = nil
|   reverse (x::xs) = append ((reverse xs), [x]);
```

### ◆ Append lists

```
fun append(nil, ys) = ys
|   append(x::xs, ys) = x :: append(xs, ys);
```

## Core ML

### ◆ Basic Types

- Unit
- Booleans
- Integers
- Strings
- Reals
- Tuples
- Lists
- Records

### ◆ Patterns

- ◆ Declarations
- ◆ Functions
- ◆ Polymorphism
- ◆ Overloading
- ◆ Type declarations
- ◆ Exceptions
- ◆ Reference Cells

## Variables and assignment

### ◆ General terminology: L-values and R-values

- Assignment `y := x+3`
  - Identifier on left refers to a memory location, called L-value
  - Identifier on right refers to contents, called R-value

### ◆ Variables

- Basic properties
  - A variable names a storage location
  - Contents of location can be read, can be changed
- ML
  - A variable is another type of value
  - Explicit operations to read contents or change contents
  - Separates naming (declaration of identifiers) from “variables”

## ML imperative constructs

### ◆ ML reference cells

- Different types for location and contents
  - `x : int` non-assignable integer value
  - `y : int ref` location whose contents must be integer
  - `!y` the contents of location `y`
  - `ref x` expression creating new cell initialized to `x`
- ML assignment
  - operator `:=` applied to memory cell and new contents
- Examples
  - `y := x+3` place value of `x+3` in cell `y`; requires `x:int`
  - `y := !y + 3` add 3 to contents of `y` and store in location `y`

## ML examples

### ◆ Create cell and change contents

```
val x = ref "Bob";  
x := "Bill";
```

### ◆ Create cell and increment

```
val y = ref 0;  
y := !y + 1;
```

### ◆ While loop

```
val i = ref 0;  
while !i < 10 do i := !i + 1;  
!i;
```

## ML Tutorial

---

- ◆ Types  
<http://burks.brighton.ac.uk/burks/language/ml/giml/home.htm>
- ◆ Discussion of results

## Types

---

## Type

---

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

- |   |  |
|---|--|
| ◆ Examples  | ◆ "Non-examples"   |
| <ul style="list-style-type: none"><li>• Integers</li><li>• Strings</li><li>• <math>\text{int} \rightarrow \text{bool}</math></li><li>• <math>(\text{int} \rightarrow \text{int}) \rightarrow \text{bool}</math></li></ul> | <ul style="list-style-type: none"><li>• <math>\{3, \text{true}, \lambda x.x\}</math></li><li>• Even integers</li><li>• <math>\{f:\text{int} \rightarrow \text{int} \mid \text{if } x &gt; 3 \text{ then } f(x) &gt; x*(x+1)\}</math></li></ul> |

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  $f : A \rightarrow B$  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:  $\text{int} * \text{int} \rightarrow \text{int}$ ,  $\text{real} * \text{real} \rightarrow \text{real}$
- $2 : \text{int}$  has only one type
- This implies  $+ : \text{int} * \text{int} \rightarrow \text{int}$
- From context, need  $x : \text{int}$
- 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

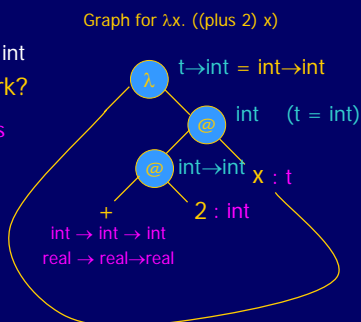
```
- fun f(x) = 2+x;
> val it = fn : int -> int
```

### ◆ How does this work?

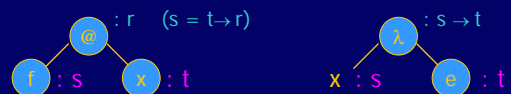
Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution



## Application and Abstraction



### ◆ Application

- $f$  must have function type  $\text{domain} \rightarrow \text{range}$
- domain of  $f$  must be type of argument  $x$
- result type is range of  $f$

### ◆ Function expression

- Type is function type  $\text{domain} \rightarrow \text{range}$
- Domain is type of variable  $x$
- Range is type of function body  $e$

## Types with type variables

### ◆ Example

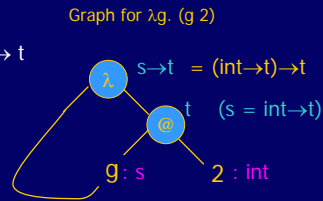
- fun f(g) = g(2);
- > val it = fn : (int → t) → t

### ◆ 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;
- > val it = fn : int → int
- f(add);
- > val it = 4 : int
- fun isEven(x) = ...;
- > val it = fn : int → bool
- f(isEven);
- > 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  $\text{bool} \rightarrow \text{bool} = \text{int} \rightarrow t$

## Another Type Inference Example

### ◆ Function Definition

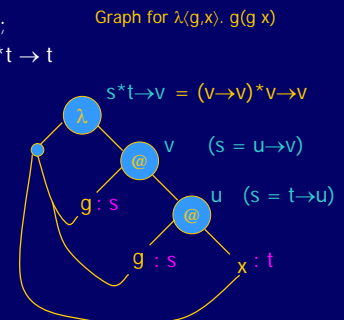
- fun f(g,x) = g(g(x));
- > val it = fn : (t → t)\*t → t

### ◆ 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"

## Questions?