Functional programming languages

Amer Diwan

The course so far

- So far we have looked at imperative languages characterized by
  - Variables: These are (usually named) memory locations that contain values.
  - Assignment Operations: A mechanism for storing into variables.
  - Iteration: Mechanisms for repeating a series of steps.

An example

- ```
  for i := 2 to n do
   j := 2; i_is_prime := true;
   while i_is_prime and (j <= i div 2) do
    if ((i mod j) # 0) then
     j := j + 1;
    else i_is_prime := false;
    endif;
    if i_is_prime then write (i); endif;
   endwhile;
  endfor;
  ```

Difficulties with imperative languages

- Each statement depends on the side effects of statements around it
  - Difficult to understand (referentially transparent)
  - Difficult to prove correct;
  - Difficult to optimize;
  - Difficult to parallelize;
  ```

The culprit

- Side effects
  - Assignments cause side effects
  - Loops depend on side effects
- Functional languages provide alternatives to the above

"Assignments cause side effects"

- So don't have assignments
- Use names rather than variables
  - `val i:int = 10;`
    versus
    `VAR i: INTEGER := 10;`

Names versus variables

- A name, say “x”, simply is a name for a value: it cannot be assigned
- `val xsquare:int = random();`
- A name doesn’t have to be a compile-time constant

Another example

- ```
  val x: int = 20;
  fun addx y = x + y;
  val x: int = 40;
  addx 30;
  ```
  ```

But what about iteration?

- Use function calls
  - `while i < 100 do`
    `... i := i + 1; end`

    versus

    ```
    fun f i:int  =
      ... if i < 100 f(i+1)
    ```

But...

- ```
  fun f i:int =
     ... if i < 100 f(i+1)
  ```
- The parameter to **`f`** changes every "iteration". Isn't that an assignment?
- No! Each time **`f`** is called, a "fresh" name "**`i`**" is created for the parameter
  ```

Some examples

- `fun id x:int = x;`
- ```
  fun sqsum (x:int, y:int) =
  let sum:int = x+y in
    sum*sum
  end
  ```
- ```
  val t:int = 10;
  sqsum(t, 5);
  sqsum(t, t);
  id (sqsum(t, t));
  ```

Properties of functions in functional languages

- Can functions have side effects?
- Can function calls be reordered?
- Are function calls referentially transparent?
- Does it matter if you use VAR or VALUE to pass parameters?

first-class functions

- Functional languages support first-class functions in their full glory
- (I'll sometimes omit types when obvious from context)
- ```
  fun appto (f, x) = f(x);
  fun square x = x * x;
  appto (square, 20);
  ```

Another example

- ```
  fun sum(x, y) = x + y;
  fun square x = x * x;
  fun compose (f, g) = fn (x,y) =>
  f(g(x,y))
  val sqsum = compose(square, sum);
  sqsum (10,20);
  ```
- Functions may of course be nested...

Other features in functional languages

- Functional languages may have a type system which most likely supports
  - polymorphism
  - data types
  - compile or run time type checking
  - exceptions
  - ...
- We will look at SML as a case study

SML's most interesting features

- Type system
  - polymorphic
  - type inference
  - static typing
- Pattern matching

Types in SML: Tuples

- ```
  val apair: int*int = (5, 7);
  val atriple: int*int*int = (3,4,5);
  val anotherpair: string*int = ("hello",
  10);
  val pairofpairs: (int*int)*(int*int) =
  ((3,4), (5,6));
  ```
- All SML functions take a single argument: but that argument can be a tuple!

Types in SML: Variant Records

- ```
  datatype PairOrTriple =
      Pair of int*int
      | Triple of int*int*int;
  ```
- ```
  val x: PairOrTriple = Pair(10,20);
  val y: PairOrTriple = Triple(10,20,30);
  fun f x: PairOrTriple = ...;
  f x;
  f y;
  ```
  ```

But how does the code use the datatypes?

- In languages with tagged unions, the common method is to use conditionals and a field like notation:
  e.g., if (x.tag = Pair) then ... end
- SML uses pattern matching

Pattern matching

- ```
  fun sum (Pair(x,y)) = x + y
     |sum (Triple(x,y,z)) = x + y + z;
  val a: PairOrTriple = Pair(10,20);
  val b: PairOrTriple = Triple(10,20,30);
  sum a;
  sum b;
  ```
- ```
  fun sum (x,y): int*int = x + y;
  val t: int*int = (10,20);
  sum t;
  sum (5,10);
  ```

Another example

- ```
  fun x_coord (Pair(x,_)) = x
    | x_coord (Triple(x,_,_) = x;
  ```

Polymorphism in SML

- OO languages inclusion polymorphism
  - Based on the inclusion or subtyping of types
- SML uses a different (but still very powerful) kind of polymorphism: parametric polymorphism
  ```

Examples

- **`fun id x:int = x;`**
- Seems a waste to restrict the parameter of **`id`** just to an integer.
  - id doesn’t use any information about “x” so should be able to pass anything for “x”.
- **`fun id x: 'a = x;`**
- **`'a`** is a "polytype": it stands for any type

Examples continued

- **`fun id x: 'a = x;`**
- **`id 10;`**
- **`id "hello";`**
- **`id (10,20);`**
- What is the return type of id?

A polymorphic "id" is cool but so what...

- ```
  x: int list = 1::2::3::nil;
  y: int list = 0::x;
  fun count l: int list =
     if l = nil then 0
     else 1+count(tl(l)); or
  fun count nil = 0
     |count (h::t : int list)= 1+count t;
  count x;
  count y;
  ```
- But, can I do:
  ```
  count "hello"::"world"::nil;
  ```

A new "count"

- ```
  fun count nil = 0
     |count (h::t : 'a list)= 1+count t;
  ```

Another example

- ```
  fun compose (f:int->int,
               g:int*int->int) =
      fn (x,y) => f(g(x,y))
  ```
- Let's find a polymorphic type for this

Let's write some more functions

- List append
- "Twice": applies the same function twice
  ```

More interesting polymorphic types

- ```
  datatype 'a Option = Some of 'a | None;
  fun string2int(s: string,
                 None: int Option) = ...
       | string2iInt(s: String,
                 Some(b): int Option) = ...
  ```

SML's polymorphism versus O-O style polymorphism

- ```
  TYPE List =
    OBJECT next: List; ... END;
  TYPE IntList = List OBJECT val: INT;
                     END;
  ```
- Let's write a list reverse in an M-3/Java/C++...
- What is the type of list reverse?

List reverse in SML

- ```
  fun reverse nil = nil
      |reverse (h::t) =
           (reverse t) @ (h::nil)
  ```
- Let's give reverse a polymorphic type

An example

- ```
  fun lookup (nil, x) = false
       |  lookup (h::t, x) = x = h orelse
  lookup(t, x);
  ```
- What is a type for lookup?

Let’s look at a call to “lookup”

- fun id1 x = x;
  fun id2 x = x;
  fun id3 x = x;
  val alist = id1::id2::nil;
  lookup(alist, id1);
  lookup(alist, id2);

Equality types

- Polytypes with double quotes are types on which you can do equality: e.g. ''a
-
```
fun lookup (nil, x: ''a) = false
    |  lookup (h::t: ''a list, x: ''a)
         = x = h orelse lookup(t, x);
```

Type inference in SML

- Why do we need type inference
  - Save programmer effort
  - More polymorphism
  - Find bugs
- A simple algorithm for type inference

Saving programmer effort and program clutter

- fun find (nil, _) = false
  | find (hd::tl, tofind) = tofind = hd orelse find(tl, tofind)
- OR
- fun find(nil: ''a list, tofind: ''a) = false
  | find((hd:''a)::(tl:''a list), tofind: ''a) = tofind = hd orelse find(tl, tofind)
- This still doesn't include return types
- Counter argument: user types are useful documentation. What do you think?

Getting most general type

- Let's suppose I have lists of integers and I need a function that checks if an integer is in my list
  - fun find(nil: int list, tofind: int) = false
    | find((hd:int)::(tl:int list), tofind: int) = tofind = hd or else find(tl, tofind)
- Works for me but not too reusable

Getting most general type (cont.)

- fun add_to_list(alist, toadd) =
  if find(alist, toadd) then alist else toadd::alist
- Possible typing:
  fun add_to_list(alist: 'a list, toadd: 'a) =
  if find(alist, toadd) then alist else toadd::alist
- Is this right?
- Polymorphic types are hard to manually get "right".

Finding bugs

- fun reverse (nil) = nil
  | reverse(x::lst) = reverse(lst);
  reverse: 'a list -> 'b list
- The type doesn't look right. What is wrong?

Summary of goals for type inference

- Give programmer the benefit of static typing without the effort
- Compute the most general type for functions to get maximum reusability
- Compute poor man's version of program “specifications"--useful for finding bugs

SML type inference example

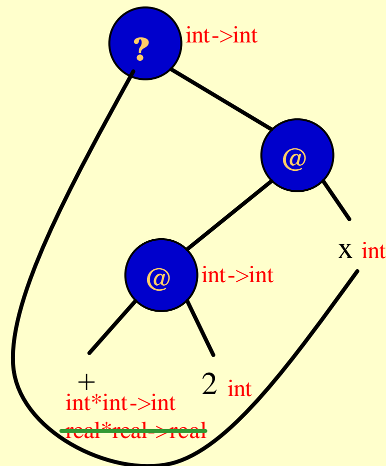
- Example
  - fun f(x) = 2 + x;
    f = fn: int->int
- How does this work?
  - + has two types: int*int->int, real*real->real
  - 2: int has only one type
  - Thus +: int*int->int
  - From context, need x: int
  - Therefore f(x:int) = 2 + x has type int->int

SML type inference: another example

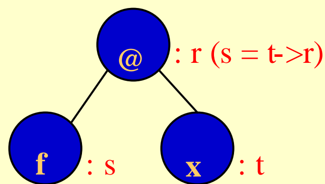
- fun f(x) = x + x
- What is the type of f?

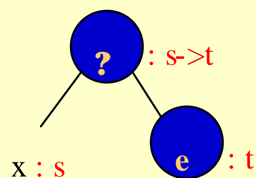
Another presentation

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



Generating constraints





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

Solving constraints

- Unification
- Basic idea:
  - If a constraint says t = u, t and u are type expressions, then unify values of t and u
  - If t or u is a primitive type then it is easy
  - If t and u are non-primitive types, then unify their components
  - If t is a type variable, replace uses of t with u

Solving constraints: examples

- x = int =>

  x = int
- int->'a = 'b->'b =>

  int = 'b and 'a = 'b =>

  int = 'a
- int * bool = 'a * 'a =>

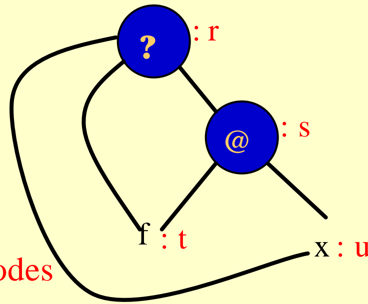
  'a = int and 'a = bool =>

  int = bool

  Type error!

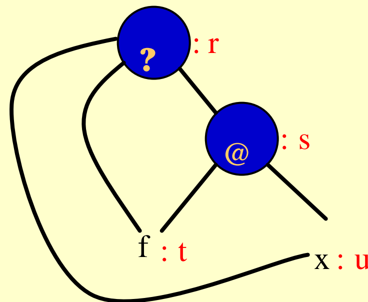
Inferring polymorphic types

- Example
  - fun apply(f, x) = f(x);
    f = fn : ('a->'b) * 'a -> 'b
- How does this work
  - Assign types to leaves
  - Assign types to interior nodes
  - Generate constraints
  - Unify!



Example (cont.)

- Constraints:
  - t = t1->t2
    t1 = u
    t2 = s
    r = t * u -> s
- Substituting for t
  - t1 = u
    t2 = s
    r = (t1->t2) * u ->s
- Substitute u for t1 and s for t2
  - r = (u->s) * u->s



More on type inference

- Perfect type inference is undecidable
- SML type inference is exponential but seems to work
  - Programmer needs to intervene sometimes when overloaded functions are involved but otherwise it is mostly automatic

Pros and Cons of type inference versus programmer supplied types

- Expressiveness
- Simplicity
- Safety
- Ease of implementation
- Efficiency

Next lecture

- Exception handling, Section 8.5