

Programs as data

Interpretation vs compilation, stack machines

Peter Sestoft
Monday 2011-09-05



Plan for today

- F# polymorphic functions and types
- Concepts: free and bound variables, ...
- Interpreters and compilers
- Compilation of expressions
 - To replace names by indices (numbers)
 - To stack machine code, without variables
 - To stack machine code, with variables
- The Postscript language



F# polymorphic functions

```
let rec len xs =  
  match xs with  
  | []      -> 0  
  | x::xr  -> 1 + len xr;;
```

The function doesn't look at the list elements

```
val len : 'a list -> int
```

... so the function is polymorphic

```
len [7; 9; 13]  
len [true; true; false; true]  
len ["foo"; "bar"]  
len [("Peter", 47)]
```

... and works in any type of list

- Same as a generic method in Java or C#

```
static int Count<T>(IEnumerable<T> xs) { ... }
```

F# polymorphic types

```
type 'a tree =  
  | Lf  
  | Br of 'a * 'a tree * 'a tree
```

The datatype has same structure regardless of node value type

```
Br(42, Lf, Lf)  
Br("quoi?", Lf, Lf)  
Br(("Peter", 47), Lf, Lf)
```

What type instances here?

- Same as a generic type in Java or C#:

```
class ArrayList<T> { ... }  
interface IEnumerable<T> { ... }  
struct Pair<T,U> { ... }  
delegate R Func<A,R>(A x);
```

Kinds of polymorphism

- *Parametric polymorphism*, as in ML, F#, Java and C#:
 - The type variable 'a' stands for an arbitrary type
 - A parametric polymorphic function works the same way regardless what the type variable stands for
- *Bounded parametric polymorphism*, as in Java, C#:
 - The type variable T stands for a type with certain properties.
 - For instance, a List<T> is printable if all its elements are:

```
class List<T> : IPrintable where T : IPrintable { ... }
```

- *Ad hoc polymorphism, or overloading*:
 - Java operator (+) works on int, double, String but not boolean
- *Virtual method calls* are sometimes said to be 'polymorphic'

```
Vehicle v = getMyVehicle();  
... v.getWeight();
```

May call getWeight()
on Bike, Car, Tank, ...

- A parametric polymorphic type is an assertion about a function
 - What terminating pure F# function has type 'a -> 'a ??
 - What terminating pure F# function has type 'a * 'b -> 'b * 'a ??



Polymorphic functions on polymorphic types

```
let rec preorder1 t =  
  match t with  
  | Lf          -> []  
  | Br(v, t1, t2) -> v :: preorder1 t1 @ preorder1 t2
```

- Return the tree's node values in pre-order
 - first root, then left subtree, then right subtree
- Works on any type of tree
- What is the type of this function?



Accumulating parameters

- The append (@) operation may be slow
- A faster version of preorder, no append!

```
let rec preo t acc =  
  match t with  
  | Lf          -> acc  
  | Br(v, t1, t2) -> v :: preo t1 (preo t2 acc);;  
  
let preorder2 t = preo t [];;
```

Accumulating
parameter

$O(n)$ versus $O(n^2)$

Can be 1000 x faster
Try `#time;;` in F#

- Function `preorder2` is correct because:
`preo t acc = preorder1 t @ acc`
- and therefore:
`preorder2 t = preo t [] = preorder1 t`

Proof, by induction on the tree

- Case `t = Lf`:
`preo Lf acc`
`= acc`
`= [] @ acc`
`= preorder1 Lf @ acc`
- Case `t = Br(v, t1, t2)`:
`preo (Br(v, t1, t2)) acc`
`= v :: preo t1 (preo t2 acc)`
`= v :: preo t1 (preorder1 t2 @ acc)`
`= v :: preorder1 t1 @ (preorder1 t2 @ acc)`
`= (v :: preorder1 t1 @ preorder1 t2) @ acc`
`= preorder1 (Br(v, t1, t2)) @ acc`

Ikke pensum,
bare nyttigt...

Set operations in F#

- We represent a set as a list without duplicates; simple but inefficient for large sets
- The empty set \emptyset is represented by []
- Set membership: $x \in vs$

```
let rec mem x vs =  
    match vs with  
    | []      -> false  
    | v::vr  -> x=v || mem x vr;;
```

```
> mem 42 [2; 5; 3];;  
val it : bool = false  
> mem 42 [];;  
val it : bool = false  
> mem 42 [2; 67; 42; 5];;  
val it : bool = true
```



Set union and difference in F#

- Set union: $A \cup B$

```
let rec union (xs, ys) =  
    match xs with  
    | []      -> ys  
    | x::xr  -> if mem x ys then union(xr, ys)  
                else x :: union(xr, ys);;
```

- Set difference: $A \setminus B$

```
let rec minus (xs, ys) =  
    match xs with  
    | []      -> []  
    | x::xr  -> if mem x ys then minus(xr, ys)  
                else x :: minus(xr, ys);;
```



Concepts: Free and bound variable occurrences

- A variable occurrence x is *bound* if it is in the *ebody* of a binding `let x=erhs in ebody`
- Otherwise it is *free*
- Which occurrences are **bound** and which **free** here:
 - let z=x in z+x
 - let z=3 in let y=z+1 in x+y
 - let z=(let x=4 in x+5) in z*2
 - let z=(let x=4 in x+5) + x in z*2
- A variable is *free* if it has some free occurrence
- Usually, a program must have no free variables...
- (... in C it may, but then must be bound by linking)

Finding the set of free variables

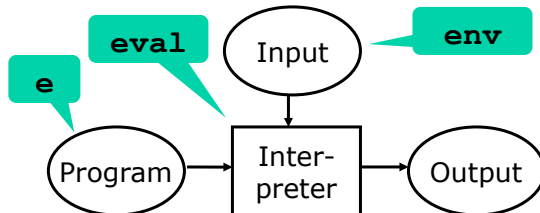
```
let rec freevars e : string list =  
  match e with  
  | CstI i -> []  
  | Var x   -> [x]  
  | Let(x, erhs, ebody) ->  
    union (freevars erhs, minus (freevars ebody, [x]))  
  | Prim(ope, e1, e2) -> union (freevars e1, freevars e2)
```

- An expression is *closed* if it has no free variables

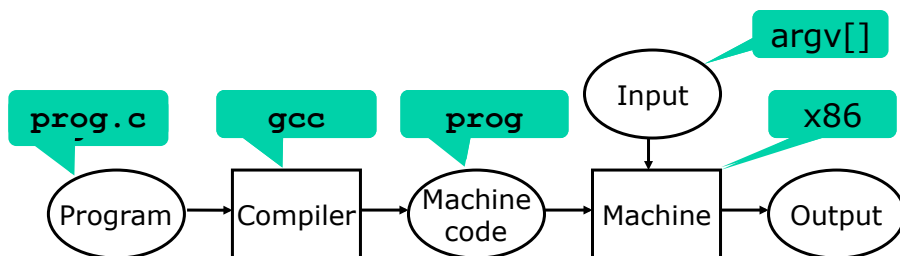
```
let closed e = (freevars e = [])
```

Interpretation and compilation

- Interpretation = one-stage execution/evaluation:



- Compilation = two-stage execution/evaluation:



Why compilation?

- Better correctness and safety. The compiler can:
 - check that all names are defined: classes, methods, fields, variables, types, functions, ...
 - check that the names have the correct type
 - check that it is legal to refer to them (not private etc)
 - improve the code, e.g. inline calls to private methods
- Better performance
 - The compiler checks are performed once, but the machine code gets executed again and again
- Why *not* compilation?
 - Compilation reduces flexibility by imposing static type checks and static name binding
 - Web programming often requires more flexibility
 - ... hence Python, Ruby, JavaScript, VBA, Groovy, ...

Replacing variable names with indices

- After compilation, there are no variable names, only indices (locations), at runtime
- Instead of symbolic names:

```
Let("z", CstI 17, Prim("+", Var "z", Var "z"))
```

- We shall use variable indexes:

```
Let(CstI 17, Prim("+", Var 0, Var 0))
```

No variable name

0 means closest variable binding

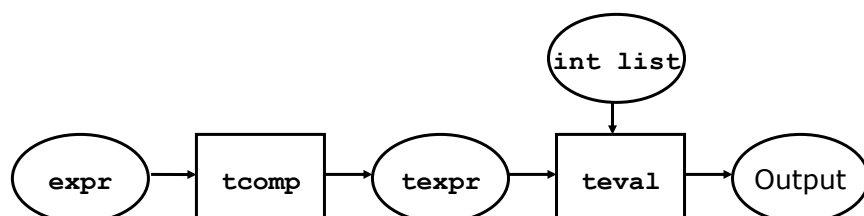
- Index = number of let-bindings to cross:

```
Let("z", CstI 17, Let("y", CstI 25, Prim("+", Var "z", Var "y")))
```

Indexes instead of variable names

- We shall compile to this "target" language:

```
type texpr = (* target expressions *)
| TCstI of int
| TVar of int (* index at runtime *)
| TLet of texpr * texpr
| TPrim of string * texpr * texpr
```



Evaluating texprs

- The runtime environment of a texpr is a list of values – not of (name, value) pairs

```
let rec teval (e : texpr) (renv : int list) : int =
  match e with
  | TCstI i -> i
  | TVar n -> List.nth renv n
  | TLet(erhs, ebody) ->
      let xval = teval erhs renv
      let renv1 = xval :: renv
      in teval ebody renv1
  | TPrim("+", e1, e2) -> teval e1 renv + teval e2 renv
  | TPrim("*", e1, e2) -> teval e1 renv * teval e2 renv
  | TPrim("-", e1, e2) -> teval e1 renv - teval e2 renv
  | TPrim _ -> failwith "unknown primitive"
```



Replacing variable names with indices

```
let rec getindex vs x =
  match vs with
  | [] -> failwith "Variable not found"
  | y::yr -> if x=y then 0 else 1 + getindex yr x;;

let rec tcomp (e : expr) (cenv : string list) : texpr =
  match e with
  | CstI i -> TCstI i
  | Var x -> TVar (getindex cenv x)
  | Let(x, erhs, ebody) ->
      let cenv1 = x :: cenv
      in TLet(tcomp erhs cenv1, tcomp ebody cenv1)
  | Prim(ope, e1, e2) -> TPrim(ope, tcomp e1 cenv, tcomp e2 cenv)
```

let z=3 in let y=z+1 in z+y

[]

["z"]

["y", "z"]

- What if the expression is not closed?

Binding-times in the environment

- Run-time environment in expr interpreter:
[("y", 4); ("z", 3)]
- Compile-time environment in expr compiler:
["y"; "z"]
- Run-time environment of texpr "machine":
[4; 3]
- The interpreter runtime environment splits to
 - A compile-time environment in the compiler
 - A runtime environment in the "machine"
- We meet such "binding-time" separation again later...

Towards more machine-like code

- Consider expression $2 * 3 + 4 * 5$
- Write it in *postfix*: $2\ 3\ *\ 4\ 5\ *\ +$
- This is sequential code for a *stack machine*:

Instructions:	Stack contents:
2 3 * 4 5 * +	2
3 * 4 5 * +	2 3
* 4 5 * +	6
4 5 * +	6 4
5 * +	6 4 5
* +	6 20
+	26

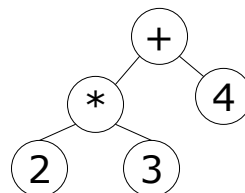
12-minute exercises

- What is the postfix of
 - $2 * 3 + 4$
 - $2 + 3 * 4$
 - $2 * (3 + 4)$
 - $2 - 3 - 4 - 5$
 - $2 - (3 - (4 - 5))$
 - $2 + 3 * 4 / 5$
- Evaluate the postfix versions using a stack



Expression stack machine without variables

Instruction	Stack before	Stack after	Effect
RCSTI n	s	s, n	Push const
RADD	s, n1, n2	s, n1+n2	Add
RSUB	s, n1, n2	s, n1-n2	Subtract
RMUL	s, n1, n2	s, n1*n2	Multiply
RDUP	s, v	s, v, v	Duplicate top elem
RSWAP	s, v1, v2	s, v2, v1	Swap



Compilation of expr to stack machine code

- A constant `i` compiles to code `[RCst i]`
- An operator application `e1+e2` compiles to:
 - code for operand `e1`
 - code for operand `e2`
 - code for the operator `+`

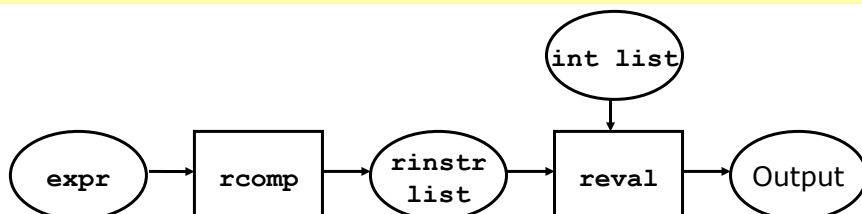
```
let rec rcomp (e : expr) : rinstr list =
  match e with
  | CstI i           -> [RCstI i]
  | Var _           -> failwith "rcomp cannot do Var"
  | Let _           -> failwith "rcomp cannot do Let"
  | Prim("+", e1, e2) -> rcomp e1 @ rcomp e2 @ [RAdd]
  | Prim("*", e1, e2) -> rcomp e1 @ rcomp e2 @ [RMul]
  | Prim("-", e1, e2) -> rcomp e1 @ rcomp e2 @ [RSub]
  | Prim _         -> failwith "unknown primitive";;
```

```
rcomp (Prim("+", Prim("*", CstI 2, CstI 3), CstI 4));;
val it : rinstr list = [RCstI 2; RCstI 3; RMul; RCstI 4; RAdd]
```

Stack machine (without variables)

- A direct implementation of state transitions:

```
let rec reval (inss : rinstr list) (stack : int list) =
  match (inss, stack) with
  | ([], v :: _) -> v
  | ([], [])     -> failwith "reval: no result on stack!"
  | (RCstI i :: insr, stkr) -> reval insr (i::stkr)
  | (RAdd :: insr, i2::i1::stkr) -> reval insr ((i1+i2)::stkr)
  | (RSub :: insr, i2::i1::stkr) -> reval insr ((i1-i2)::stkr)
  | (RMul :: insr, i2::i1::stkr) -> reval insr ((i1*i2)::stkr)
  | (RDup :: insr, i1::stkr) -> reval insr (i1 :: i1 :: stkr)
  | (RSwap :: insr, i2::i1::stkr) -> reval insr (i1 :: i2 :: stkr)
  | _ -> failwith "reval: too few operands on stack";;
```



Concepts

- An expression e is compiled to a sequence of instructions
- **Net effect principle:**
 - The *net effect* of executing the instructions is to leave the expression's value on the stack
- *Compiler correctness* relative to interpreter
 - Executing the compiled code gives the same result as executing the original expression
 - That is:


```
reval (rcomp e []) [] equals eval e []
```

How store (let-bound) variables?

- Idea: Put them in the stack! Classic, 1960'es
- So stack contains mixture of
 - intermediate results (as before)
 - values of bound variables
- To get a variable's value, index off the stack top
- Example: `2 * let x=3 in x+4 end`
- Code: `2 3 SVAR(1) 4 SADD SSWAP SPOP SMUL`

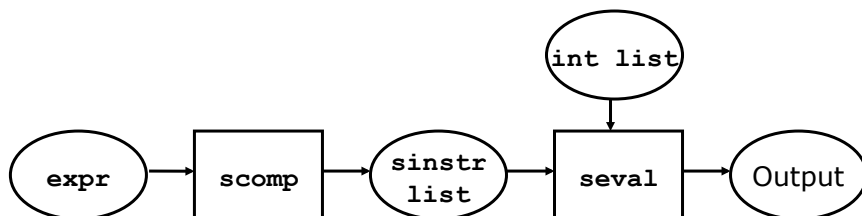
Instructions:		Stack:
2 3	SVAR(1)	2
4	SADD	2 3
3	SVAR(1)	2 3 3
4	SADD	2 3 3 4
4	SADD	2 3 7
4	SADD	2 7 3
	SSWAP	2 7
	SPOP	14
	SMUL	

Value of let-rhs is put on stack top

Must be removed after let-body

Expression stack machine with variables

Instruction	Stack before	Stack after	Effect
SCSTI n	s	s, n	Push const
SVAR x	s	s, s[x]	Index into stack
SADD	s, n1, n2	s, n1+n2	Add
SSUB	s, n1, n2	s, n1-n2	Subtract
SMUL	s, n1, n2	s, n1*n2	Multiply
SPOP	s, v	s	Remove top elem
SSWAP	s, v1, v2	s, v2, v1	Swap



Stack machine (with vars) in F#

```

let rec seval (inss : sinstr list) (stack : int list) =
  match (inss, stack) with
  | ([], v :: _) -> v
  | ([], []) -> failwith "seval: no result on stack"
  | (SCstI i :: insr, stk) -> seval insr (i :: stk)
  | (SVar i :: insr, stk) -> seval insr (List.nth stk i :: stk)
  | (SAdd :: insr, i2::i1::stkr) -> seval insr (i1+i2 :: stkr)
  | (SSub :: insr, i2::i1::stkr) -> seval insr (i1-i2 :: stkr)
  | (SMul :: insr, i2::i1::stkr) -> seval insr (i1*i2 :: stkr)
  | (SPop :: insr, _ :: stkr) -> seval insr stkr
  | (SSwap :: insr, i2::i1::stkr) -> seval insr (i1::i2::stkr)
  | _ -> failwith "seval: too few operands on stack";
  
```

```

type sinstr =
  | SCstI of int
  | SVar of int
  | SAdd
  | SSub
  | Smul
  | Spop
  | SSwap
  
```

This `seval` "machine" combines

- `teval`: variables as indices
- `reval`: stack machine code

Compiling to the seval "machine"

- The compile-time env. must distinguish between intermediate results and let-bound variables:

```
type rtvalue =
| Intrlm          (* An intermediate result *)
| Bound of string; (* A let-bound variable *)

let rec scomp (e : expr) (cenv : rtvalue list) : sinstr list =
match e with
| CstI i -> [SCstI i]
| Var x -> [SVar (getindex cenv (Bound x))]
| Let(x, erhs, ebody) ->
    scomp erhs cenv @ scomp ebody (Bound x :: cenv)
    @ [SSwap; SPop]
| Prim("+", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Intrlm :: cenv) @ [SAdd]
| Prim("-", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Intrlm :: cenv) @ [SSub]
| Prim("*", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Intrlm :: cenv) @ [SMul]
| Prim _ -> failwith "scomp: unknown operator";;
```

The compile-time environment

- The compile-time environment keeps track of variable positions in the stack
- The compile-time environment is a stack; an abstraction of the run-time stack

Position in expression:

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

Compile-time env:

```
TEMP
TEMP x
TEMP x TEMP
TEMP
```


seval stack machine in Java (almost C)

```
while (pc < code.length)
  switch (instr = code[pc++]) {
  case SCST:
    stack[sp+1] = code[pc++]; sp++; break;
  case SVAR:
    stack[sp+1] = stack[sp-code[pc++]]; sp++; break;
  case SADD:
    stack[sp-1] = stack[sp-1] + stack[sp]; sp--; break;
  case SSUB:
    stack[sp-1] = stack[sp-1] - stack[sp]; sp--; break;
  case SMUL:
    stack[sp-1] = stack[sp-1] * stack[sp]; sp--; break;
  case SPOP:
    sp--; break;
  case SSWAP:
    { int tmp = stack[sp];
      stack[sp] = stack[sp-1];
      stack[sp-1] = tmp;
      break; }
  default:
    throw new RuntimeException("Illegal instruction");
}
```

code : int[]
pc = program counter, points into code
stack : int[]
sp = stack pointer, points into stack

Stack machines everywhere

- Burroughs B5000 (1961) hardware
- Forth virtual machine (1970)
- P-code, UCSD Pascal (1977)
- Western Digital Pascal microEngine hardware
- Postscript (1984)
- Java Virtual Machine (1994)
- picoJava JVM core hardware
- .NET Common Language Runtime (1999)
- ARM Jazelle instructions (2005) hardware
- Intel cpu stack pointer prediction hardware
- ... zillions of others

Postscript (.ps) is a postfix, stack-based language

- A Postscript printer is an interpreter:

```
4 5 add 8 mul =
```

$(4 + 5) * 8$

```
/x 7 def  
x x mul 9 add =
```

let $x=7$ in
 $x*x+9$

```
/fac { dup 0 eq  
      { pop 1 }  
      { dup 1 sub fac mul }  
      ifelse } def
```

$n!$, factorial
function

```
gs -sNODISPLAY on ssh.itu.dk
```

Reading and homework

- This week's lecture:
 - PLCSD chapter 2
 - Exercises 2.2, 2.3, 2.4, 2.8
 - Send zip-file BPRD-02-Dit-Navn.zip to drc@itu.dk no later than Tuesday 13 Sept
- Next week's lecture:
 - PLCSD chapter 3
 - Mogensen 2010 sections 2.1-2.7 (except 2.6.1), 2.9, 3.1-3.6