Functional languages

Arithmetic expressions and functions

32*(51+1) 32
51 1

let f x = (3*x)
let f = fun x -> (3*x)
f(12)+2

let g x y = 3*x+y
let g = fun x -> (fun y -> 3*x+y)
8 + (g 33 5)

Functions on the right (functions as arguments)

let g = fun f -> f 3
let h = fun x -> x+5
g h

let p = fun x y -> x+y
let q = fun z -> z+2

Typical programs we want to execute, and how we write them

▶ notation for applications: g 3
▶ in maths: g(3)
▶ sometimes g@3 to stress that application is a binary operator
▶ using the let construct
▶ a program is a sequence of expression (the “main”)
▶ let x = 3 in let y = 4 in let z = 5 in (x+y*z)
▶ will also be written

let x = 3
let y = 4
let z = 5
(x+y*z)

▶ a nested let..in
let f = fun x ->
let y = g (x*x) in
if y>0 then y else x
← here is f’s return (y or x)

Fun, a small functional programming language

▶ syntax

```
e ::= fun x -> e | e1 e2 | x
   | let x = e1 in e2
   | e1 + e2 | 1, 2, 3, ...
```

x, y, z, ... ∈ V variable identifiers

▶ two versions of the operational semantics

Compiling to an Abstract Machine

The reason for closures

▶ recall the example that motivated the introduction of closures

let h = fun t -> t+t
let g = fun y -> 30 + (h y)
let h = 12
g 5

Program transformations in Fun

▶ hence the Abstract Machine transition

```
Closure(x,c); c | σ | s || c | σ | (x,c)[σ].s
```

notice the duplication of the environment
Free and bound variables in programs

- a Fun program
  ```ml
  let t = 3
  let u = fun x -> x+2
  let v = fun x -> x + u (2*x)
  let z = x+2*t in
  z + (f y)
  ```
  a definition (like the one for g above) makes sense provided the variables it uses make sense in the environment where the definition occurs

- in let x = e1 in e2, x is bound in e2
  in fun x -> e, x is bound in e

- a variable is free if it is not bound
  • nota: let x = e1 in e2 behaves like (fun x -> e2) e1
  • let x = e1 in e2 and fun x -> e are binders,
    the scope of x is e2 (resp. e)

Handling closures

- back to the example
  ```ml
  let h = fun t -> t+t
  let g = fun y -> 30 + (h y) g = (fun y -> 30 + (h y))(h, fun t -> t)
  let h = 12
  g 6
  ```
  in compilers for functional languages, closures are typically represented by a pair
  1. pointer to the code for the body
  2. pointer to the environment
  - has to be allocated in the heap
  - not the whole environment
  - may contain, recursively, other pointers to environments

Representing closures: closure conversion

- represent explicitly closures in the language
  Fun extended with tuples/records/structs

- modify functions:
  - when they are defined
    ```ml
    [fun x -> e] = let code = fun (c,x) ->
    let (_,x1,...,xn) = c in [e] in
    code (c,x1,...,xn)
    ```
    where x1,...,xn are the free variables of fun x -> e
  - "let (_,x1,...,xn) = c in [e]": the function reconstructs the environment before executing [e]

- and when they are called
  ```ml
  [e1 e2] = let c = [e1] in
  let code = proj0(c) in
  code (c,[e2])
  ```

Lambda lifting: a transformation from Fun to Fun

- transforming the program in order to obtain a flat structure for functions
  ```ml
  (in the "e" of a fun x -> e))
  ```
  example:
  ```ml
  let f x =
  let g y = x+2*y
  in
  g 5*x + g 3*x
  ```
  modifying the definition and the calls to these functions

- we obtain a set of recursive definitions of functions,
  - with no free variable
  - all at the same level

Introducing continuations

- replace "returns" with function calls
  ```ml
  from let f x y = x+2*y
  to let f x y k = k (x+2*y)
  ```
  k is the "future" of the computation

- calling a function
  ```ml
  let f x y = y + (g (2*x))
  ```
  first compute g (2*x)
  then (return inside f and) add y

- recursive functions
  ```ml
  let rec fact n = if n<2
  then 1 else n*(fact (n-1))
  ```
  ```ml
  let rec fact n k = if n<2 then k
  else let k' = fun u -> k (n*u) in fact (n-1) k'
  ```

Making it systematic: the CPS translation

- every value is translated into a program that waits for a receiver for this value
  ```ml
  [12] = fun k -> k 12  k: the receiver
  ```
  NB: using "k" for continuations is rather standard,
  let's forget about using k for integers (∈ ℤ)

- accordingly, define a translation from Fun to Fun,
  ```ml
  [e] = fun k -> ...
  ```

CPS: Continuation Passing Style

- we can now systematically translate any Fun program to its equivalent CPS
  ```ml
  [e1 e2] = [e1] [e2]
  ```
  on the board

Continuations and control

- the flow is explicit
  ```ml
  "fun v -> .." insures sequentialisation
  for instance, you know which summand you evaluate first
  ```
  ```ml
  while loops Demo do_while_cont.ml
  ```
  but also: return, break, continue, for

- exceptions
  ```ml
  try with / raise, try catch / throw
  ```
  it is possible to translate Fun+exceptions into Fun
Properties of the CPS translation on the board

Tail calls

- consider \( \text{let } f \ x = g(h(x)) \)
  - first call \( h \), then return in \( f \)
  - then call \( g \) ← tail call
  - then return in \( f \), and exit from \( f \)

- tail calls can be compiled in a specialised way, so that we exit from \( f \) when calling \( g \)
  - no push on the stack
- tail recursive functions: recursive calls are tail calls
  - the stack does not grow along recursion

Demo append.ml, term.ml

CPS as an intermediate representation

the target language of the transformation is almost an intermediate language

- every call is terminal (always one function alive)
- refining the CPS transform to yield simpler programs
  - "administrative" reductions
  - treatment of arithmetical expressions when there are no function calls
  - treating n-ary functions as such
donot translate \( \text{fun x1 -> fun x2 -> fun x3 -> e} \)
to \( \text{fun x1 -> fun k1 -> fun x2 -> fun k2} \)
\( \rightarrow \text{fun x3 -> fun x3 -> [e]} \)
but to \( \text{fun x1 x2 x3 k -> [e]} \)
or maybe to \( \text{fun (x1, x2, x3, k) -> [e]} \)
- distinguishing "true functions" from continuations (jumps)
- backwards transformation (out of CPS)
  - in order to compile using a stack
  - CPS form used for optimisation purposes

CPS vs SSA

- the CPS transform yields programs which
  - are rather difficult to read
  - involve elementary operations
    - arithmetic operations and function calls only on atoms (variables, constants)
    - function calls are terminal
- CPS form (and its refinements/variations/improvements) is used as an intermediate representation for functional compilers
- CPS: the functional counterpart of SSA
  - unique assignment to variables
  - dominators ↔ scope
  - \( \varphi \) nodes correspond to (some) continuations
  - join point in the CFG ↔ continuation
  - transfer of control and expressing \( \varphi \) ↔ calling a continuation