Compiling functional languages

http://www.cse.chalmers.se/edu/year/2011/course/CompFun/

Lecture 2
C representation

Johan Nordlander

Recall our core language

```
prog    ::=   module K where ds
d       ::=   x = e
e       ::=   x | K es | lit | e e | \x -> e | let ds in e | case e of alts
alt     ::=   lit -> e | K xs -> e
```

Good for analysis and optimization,
but still not directly mappable to C

Restrict form even further

prog	::=	**module** K **where** ds
d	::=	f = \xs -> b | x = K es | x = e
b	::=	**let** ds **in** b | **case** x **of** alts | b ‖ b | **fail** | e
e	::=	x | f | x es | f es | lit | K
alt	::=	K -> b | K xs -> b | lit -> b

Main difference:
expression syntax now depends on <u>position</u>

- The right-hand side of a declaration (d)
- The body of a function (b)
- Arguments to functions and constructors (e)

Minor differences: marking known functions (f)
+ multi-argument abstraction & application

C correspondence

Declarations:

f = \xs -> b	A C function declaration (if on the top level)
x = K es	A malloc() call followed by assignments
x = e	A single assignment

Function bodies:

let ds **in** b	A sequence of assignments (if ds not recursive)
case x **of** alts	A switch statement
e	A return statement
fail and e||e	break and sequential composition

Expressions:

x	Variable x
f es	A function call to f (if arity matches)
lit	Literal lit (if not a string literal)
K_i	Integer literal i
x es and f	Deferred...

Data layout

typedef int *Ptr;

Basic assumptions:

(Ptr)(int)x = x   (int)(Ptr)y = y

Construction:

$x = K_i\ e_1 \ldots e_n$

```
Ptr x = malloc((n+1)*sizeof(int));
x[0] = i;
x[1] = (int)e1; ... x[n] = (int)en;
```

Deconstruction:

case x **of**

....

$K_i\ x_1 \ldots x_n$ -> $body_i$

```
switch (x[0]) {
    ...
    case i: { Ptr x1 = (Ptr)x[1]; ...
              Ptr xn = (Ptr)x[n];
              bodyi }
```

Nullary constructors

Could just use the generic form:

$x = K_i$	`Ptr x = malloc(sizeof(int));` `x[0] = i;`

case x **of**
    ...
    K_i -> body
	switch (x[0]) {
	    ...
	    case i: { body }

For better memory efficiency, encode as <u>small pointer</u>:

K_i	(Ptr) i

case x **of**
    K_i -> $body_i$
    ...
    K_j x_1 ... x_n -> $body_j$
	switch ((int)x) {
	    case i: { $body_i$ }
	    ...
	    default: **switch** (x[0]) {
	        case j: { Ptr x_1 = (Ptr)x[1]; ...
	            Ptr x_n = (Ptr)x[n];
	            $body_j$ }

Single constructors

Could just use the generic form:

$x = K_0\ e_1 \ldots e_n$	`Ptr x = malloc((n+1)*sizeof(int));`
	`x[0] = 0;`
	`x[1] = (int)e1; ... x->arg[n] = (int)en;`

case x **of**
$K_0\ x_1 \ldots x_n$ -> $body_i$
	switch `(x[0]) {`
	case `0: { Ptr x1 = (Ptr)x[1]; ...`
	`Ptr xn = (Ptr)x[n];`
	`body0 }`

For better efficiency, encode <u>without a tag</u>:

$x = K_0\ e_1 \ldots e_n$	`Ptr x = malloc(n*sizeof(int));`
	`x[0] = (int)e1; ...`
	`x[n-1] = (int)en;`

case x **of**
$K_0\ x_1 \ldots x_n$ -> $body_0$
	`Ptr x1 = (Ptr)x[0]; ...`
	`Ptr xn = (Ptr)x[n-1];`
	`body0`

On primitives

- Syntactically, a primitive operation is just a named function f the compiler already knows about (implicitly declared)
- However, there are two sets of such names:
  - Primitives of the source language (up to you)
  - Primitives of the target (defined by C)
- Requires a <u>conscious mapping</u> (again up to you!)

Obtaining restricted form

- Collecting multiple arguments:

```
translate (...((x e1) e2) ... en)      =   x e1 e2 ... en
translate (\x1 -> ... -> \xn -> e)     =   \x1 ... xn -> e
```

- Normalizing an e depending on position:

```
normD (f = \xs -> e)    =   f = \xs -> normB e
normD (x = K es)        =   ...
normD (x = e)           =   ds ++ x = e'  where (ds,e') = normE e

normB (\xs -> e)        =   let normD (x = \xs -> e) in x
normB (case/let)        =   ...
normB e                 =   let ds in e'   where (ds,e') = normE e
```

Obtaining restricted form

- Normalizing an e depending on position:

```
normE x                 =  ([], x)
normE lit               =  ([], lit)
normE K                 =  ([], K)
normE (f e1 ... en)     =  (ds1 ++ ... ++ dsn, f e1' ... en')
    where (dsi, ei')    =  normE ei
normE (x e1 ... en)     =  ...
normE (e e1 ... en)     =  (normD (x = e) ++ ds, e')
    where (ds, e')      =  normE (x e1 ... en)
normE (let ds in e)     =  (normD ds ++ ds', e')
    where (ds', e')     =  normE e
normE (case ...)        =  (normD (x = \_ -> case ...), x ())
normE e                 =  (normD (x = e), x)
```

Manipulating scopes

- The hazards of moving and merging decls:

  normE (**let** x = 3 **in** **let** x = 4 **in** x) = ([x = 3, x = 4], x) ???

  normE (f (**let** x = 3 **in** x) x) = ([x = 3], f x x) ???

- Solution: alpha-convert local scope

  normE (**let** x = 3 in **let** x = 4 in x) = ([x = 3, y = 4], y) y new

  normE (f (**let** x = 3 **in** x) x) = ([y = 3], f y x) y new

- Detecting name capture:
  - check against all vars in scope (keep an environment)
  - or check the vars actually free in shadowed exprs

Free variables

- A standard notion in lambda calculus:

fv x	=	{x}
fv (K e_1 … e_n)	=	fv e_1 U … U fv e_n
fv lit	=	{ }
fv (e e')	=	fv e U fv e'
fv (\x -> e)	=	fv e \ {x}
fv (**let** ds **in** e)	=	(fv ds U fv e) \ dom ds
fv (**case** e **of** alt_1 ;…; alt_n)	=	fv e U fv alt_1 U … U fv alt_n
fv (x_1 = e_1 ;…; x_n = e_n)	=	fv e_1 U … U fv e_n
fv (K xs -> e)	=	fv e \ {xs}
fv (lit -> e)	=	fv e
dom (x_1 = e_1 ;…; x_n = e_n)	=	{x_1,…,x_n}

After normalization

- A program form corresponding to C syntax, but with some serious caveats:
  - Function declarations must be on top level only
  - Only function declarations may be recursive
  - Function calls must match arity of callee
  - Function names must not be used as values
  - Unknown functions cannot be called
- (Assuming string literals already desugared away...)

Recursive declarations

- The simple case: recursive functions (corresponds to cross-referencing code-blocks on the assembly level)
- Also a non-issue in lazy languages (where every name denotes a code-block in general)
- Normally forbidden in strict languages:
  - Recursive non-functions: x = K y; y = L x
  - Recursive non-values: x = f y; y = g x

Recursive declarations

- Natural approach: forbid recursive non-functions and non-values in our language as well (can be checked initially and easily preserved)
- However, checking after sorting according to dependency order adds valuable expressiveness:

```
let   f = \xs -> ... x ... g ...
      x = 2+3
      g = \ys -> ... f ...
      y = f []
in e
```

⇒

```
let x = 2+3
in let  f = \xs -> ... x ... g ...
        g = \ys -> ... f ...
   in let y = f []
      in e
```

Only top-level functions

- Naive idea: just move the declarations!
- Problem: loss of local scope

```
f = \a  -> let g = \b -> a + b          g = \b -> a + b
           in g 7                 ⇒     f = \a -> g 7
```

- A way forward: first turn free variables into parameters!

```
f = \a  -> let g = \a b -> a + b        g = \a b -> a + b
           in g a 7               ⇒     f = \a -> g a 7
```

Lambda-lifting

- An algorithm for lifting functions (lambda-abstractions) out of their scope [Johnsson (1985), variant in SPJ's book]
- Fact: lifting itself is trivial (just avoid name-clashes) — adding the necessary parameters to functions is the intersting part
- We'll study a formulation that only performs the interesting part!

A lambda-lifter

- Assume fv splits its output as (fs,xs)
- Assume ext maps each f in scope to its extra args
- The interesting cases:

```
lift ext (f es)               =   f (ext f ++ lift ext es)
lift ext (let ds in b)        =   let (lift ext' ds) in (lift ext' b)
    where (f1...fn, xs)       =   fv ds \ dom ds
          xs'                 =   xs U ext f1 U ... U ext fn
          ext' f              =   if (f `elem` dom ds) then xs' else ext f
...
lift ext (f = \xs -> e)       =   f = \(ext f ++ xs) -> lift ext e
```

A lambda-lifter

- On the top-level:

  module K **where** ds ⇨ **module** K **where** lift ext_0 ds
  where ext_0 f = []

- Result: a program where each declaration `f = \xs -> e` has zero free variables
- Such decls can easily be moved to the top
- Variant: exclude the global non-functions when listing free variables in lift

Anonymous functions

- Our latest expression grammar:

Calling one

Creating one

e ::= x | x es | f | f es | lit | K

Creating one if arguments are too few

- Must be supported – not a functional language otherwise!
- Requires the concept of <u>closures</u>!

Closures

- The generic representation of functions: a function pointer with a list of free variables
- The limits of lambda-lifting:

```
g = \a  -> let f = \b -> a + b
           in h f
h = \x -> x 7
```

⇒

```
f = \a b -> a + b
g = \a -> h (f a)
h = \x -> x 7
```

Too few arguments regardless!

- Closures can represent partial applications, even in the presence of free variables
- Nevertheless, lambda-lifting before closure-conversion simplifies the presentation somewhat

Closure-conversion

- Assume a lambda-lifted f = \x1 ... xn -> e

```
closureConvert f  =  CL f0 n
closureConvert (f e1 ... em)   =
   | m < n       =   CL fm (n-m) e1 ... em
   ...
   where fm is a new top-level function
      fm = \xthis xm+1 ... xn -> case xthis of
                                  CL _ _ y1 ... ym -> f y1 ... ym xm+1 ... xn

closureConvert (x e1 ... em)   =  case x of CL funknown n
                                     | m == n  ->  funknown x e1 ... em
                                     ...
```

Closures

- CL is an ordinary constructor name (a K) and a closure term is just a constructor application that references an f
- After closure conversion, these applications will be our <u>only</u> references to function names outside function calls
- Note: static typing will actually require a CL_k for each closure arity k (a well as existentials and subtyping!), but we're past type-checking here!

Closure-conversion

- Example before and after lambda-lifting:

```
g = \a -> let f = \b -> a + b            f = \a b -> a + b
          in f g                  ⇒      g = \a -> h f
h = \x -> x 7                            h = \x -> x 7
```

- And after closure-conversion:

```
f = \a b -> a + b
g = \a -> CL f_1 1 a
h = \x -> case x of CL f_unknown 1 -> f_unknown x 7

f_1 = \x_this x_2 -> case x_this of CL _ _ y_1 -> f y_1 x_2
```

- But we're still ignoring arity mismtaches...

Matching arities

- Strategies for matching function arity with the number of arguments:
  - "Push/enter": arguments pushed and code entered unconditionally, matching done by <u>called function</u>
  - "Eval/apply": function evaluated and asked for arity <u>by caller</u>, then only applied if enough arguments are present

Push/enter

- Assume arguments $e_1 \dots e_m$ are on the stack
- In prologue to each function $f = \backslash x_1 \dots x_n \text{ -> } e$:
  - If $m = n$, return result of call (popping $e_1 \dots e_n$)
  - If $m < n$, pop $e_1 \dots e_m$ and return closure corresponding to $\backslash x_{m+1} \dots x_n \text{ -> } f\ e_1 \dots e_m\ x_{m+1} \dots x_m$
  - If $m > n$, <u>enter</u> result of current call after popping $e_1 \dots e_n$

Push/enter

- Simple model inspired by OO virtual methods
- Involves rather heavy use of indirect jumps
- Finding & counting all arguments on the stack is <u>hard using C calling conventions</u>
- In use: core characteristic of the original STG-machine (Peyton Jones, 1992), which is the back-end format used by GHC

Eval/apply

- Assume f has arity n
- For each call $f\ e_1 \ldots e_m$:
  - If $m = n$, just return the result of the call
  - If $m < n$, return closure corresponding to $\backslash x_{m+1} \ldots x_n \text{ -> } f\ e_1 \ldots e_m\ x_{m+1} \ldots x_m$
  - If $m > n$, let x be the result of $f\ e_1 \ldots e_n$ and continue applying call $x\ e_{n+1} \ldots e_m$
- Technique used by common ML implementations (SML-NJ, O'Caml), but nowadays also GHC

Checking arities

(eval/apply)

- Assuming $f = \backslash x_1 \ldots x_n \rightarrow e$

  closureConvert f = $CL\ f_0\ n$

  closureConvert ($f\ e_1 \ldots e_m$) =

  | $m == n$ = $f\ e_1 \ldots e_m$

  | $m < n$ = $CL\ f_m\ (n-m)\ e_1 \ldots e_m$

  | $m > n$ = $apply_{m-n}\ (f\ e_1 \ldots e_n)\ e_{n+1} \ldots e_m$

  where each $apply_k$ is a run-time system function TBD

- Note: checks are done at <u>compile-time</u>

Checking arities
(eval/apply)

- The full dynamic case (checks at <u>run-time</u>!):

closureConvert ($x\ e_1 \ldots e_m$) = $\mathrm{apply}_m\ x\ e_1 \ldots e_m$

$$
\begin{array}{lll}
\mathrm{apply}_m & = \backslash x_{this}\ x_1 \ldots x_m \to \textbf{case}\ x_{this}\ \textbf{of}\ CL\ f_{unknown}\ n \\
& \quad \mid m == n \to f_{unknown}\ x_{this}\ x_1 \ldots x_m \\
& \quad \mid m < n \quad \to CL\ \mathrm{pap}_{n-m,m}\ (n-m)\ x_{this}\ x_1 \ldots x_m \\
& \quad \mid m > n \quad \to \mathrm{apply}_{m-n}\ (f_{unknown}\ x_{this}\ x_1 \ldots x_n)\ x_{n+1} \ldots x_m
\end{array}
$$

$$
\begin{array}{l}
\mathrm{pap}_{k,m} = \backslash x_{this}\ x_1 \ldots x_k \to \textbf{case}\ x_{this}\ \textbf{of}\ CL\ \_\ \_\ y_{that}\ y_1 \ldots y_m \to \\
\qquad\qquad \mathrm{apply}_{m+k}\ y_{that}\ y_1 \ldots y_m\ x_1 \ldots x_k
\end{array}
$$

Summary

- C code generation involves
  1) Normalization (pretty straightforward)
  2) Lamda-lifting (known functions)
  3) Closure conversion (anonymous/partial apps)
- 3) supersedes 2) but is generally less efficient
- Challenge: avoid the need for special $pap_{k,m}$ functions for every combination of k and m
- Idea: make m a closure parameter as well, and write a generic $pap_{k,m}$ directly in assembly code