Programs as Data
Backwards optimizing compilation of micro-C

Peter Sestoft

Monday 2013-11-04

Today

- Husk kursusevaluering denne uge!
- Spørgetime: hvornår?
  - Muligt: man 16/12, ons 18/12, fre 20/12
  - Eksamen er torsdag 2. januar-fredag 3. januar

- Mulige bachelorprojekter – sidst i forelæsn.

- Deficiencies in the old micro-C compiler
- A backwards, continuation-based, compiler
- Optimizing code on the fly

Micro-C compiler shortcomings

- The compiler often generates **inefficient code**

```
GETBP
CSTI 0
ADD
LDI
```

could be

```
GETBP
LDI
```

```
INCSP -1
INCSP -1
```

could be

```
INCSP -2
```

- The **compiler itself is inefficient**, using (`@`) a lot:

```
| If(e, stmt1, stmt2) ->
  let labelse = newLabel()
  let labend  = newLabel()
  in cExpr e varEnv funEnv @ [IFZERO labelse]
     @ cStmt stmt1 varEnv funEnv @ [GOTO labend]
     @ [Label labelse] @ cStmt stmt2 varEnv funEnv
     @ [Label labend]
```

- **Tail calls** are not executed in constant space

Last slide from lecture 7

Example, if-statement (ex19.c)

```
void main(int x) {
  if (x == 0) print 33; else print 44;
}
```

- The old micro-C compiler produces this:

```
    GETBP; CSTI 0; ADD; LDI; CSTI 0; EQ; IFZERO L2;
    CSTI 33; PRINTI; INCSP -1; GOTO L3;
L2: CSTI 44; PRINTI; INCSP -1;
L3: INCSP 0; RET 0
```

- We want it to produce this better code:

```
    GETBP; LDI; IFNZRO L2;
    CSTI 33; PRINTI; RET 1;
L2: CSTI 44; PRINTI; RET 1
```

Generating code backwards

```
cExpr expr varEnv funEnv : instr list
```

OLD

```
cExpr expr varEnv funEnv C : instr list
```

NEW

- C is the code following the code for expr
- That is, C represents the continuation of expr
- Code is generated backwards
- Why is this useful?
  - Code for expr "knows what happens next"
  - So can optimize code for expr; if C is [NOT] and expr is 1, then [CSTI 1; NOT] becomes [CSTI 0]
  - The compiler avoids the expensive @ operations

The old and new expression compiler

```
and cExpr e varEnv funEnv : instr list =
    match e with
    | CstI i           -> [CSTI i]
    | Prim1(ope, e1) ->
      cExpr e1 varEnv funEnv
      @ (match ope with
             | "!"       -> [NOT]
             | "printi" -> [PRINTI] ...)
```

OLD

Make lists of instructions, append them

```
and cExpr e varEnv funEnv C : instr list =
    match e with
    | ...
    | CstI i           -> CSTI i :: C
    | Prim1(ope, e1) ->
      cExpr e1 varEnv funEnv
            (match ope with
                 | "!"       -> NOT :: C
                 | "printi" -> PRINTI :: C
                 | ...)
```

NEW

Put new code in front of given code

NB: Same code so far, generated more efficiently

Code improvement rules based on bytecode equivalences

Code	Better equivalent code	When
0; ADD	<no code>	
0; SUB	<no code>	
0; NOT	1	
n; NOT	0	$n \neq 0$
1; MUL	<no code>	
1; DIV	<no code>	
0; EQ	NOT	
INCSP 0	<no code>	
INCSP m; INCSP n	INCSP (m+n)	
0; IFZERO a	GOTO a	
n; IFZERO a	<no code>	$n \neq 0$
0; IFNZRO a	<no code>	
n; IFNZRO a	GOTO a	$n \neq 0$

Joint exercise 1 (code for ex13.c)

source

```
void main(int n) {
  int y;
  y = 1889;
  while (y < n) {
    y = y + 1;
    if (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0))
      print y;
  }
}
```

generated by old compiler

```
INCSP 1; GETBP; CSTI 1; ADD; CSTI 1889; STI; INCSP -1; GOTO L3;
L2: GETBP; CSTI 1; ADD; GETBP; CSTI 1; ADD; LDI; CSTI 1; ADD; STI;
INCSP -1; GETBP; CSTI 1; ADD; LDI; CSTI 4; MOD; CSTI 0; EQ;
IFZERO L7; GETBP; CSTI 1; ADD; LDI; CSTI 100; MOD; CSTI 0; EQ; NOT;
IFNZRO L9; GETBP; CSTI 1; ADD; LDI; CSTI 400; MOD; CSTI 0; EQ;
GOTO L8; L9: CSTI 1; L8: GOTO L6; L7: CSTI 0; L6: IFZERO L4; GETBP;
CSTI 1; ADD; LDI; PRINTI; INCSP -1; GOTO L5; L4: INCSP 0; L5:
INCSP 0; L3: GETBP; CSTI 1; ADD; LDI; GETBP; CSTI 0; ADD; LDI; LT;
IFNZRO L2; INCSP -1; RET 0
```

- Simplify the bytecode, eg. by deleting superfluous instructions

Optimizing code while generating it

```
and cExpr e varEnv funEnv C : instr list =
    match e with
    | CstI i            -> addCST i C
```

```
let rec addCST i C =
    match (i, C) with
    | (0, ADD        :: C1) -> C1
    | (0, SUB        :: C1) -> C1
    | (0, NOT        :: C1) -> addCST 1 C1
    | (_, NOT        :: C1) -> addCST 0 C1
    | (1, MUL        :: C1) -> C1
    | (1, DIV        :: C1) -> C1
    | (0, EQ         :: C1) -> addNOT C1
    | (0, IFZERO lab :: C1) -> addGOTO lab C1
    | (_, IFZERO lab :: C1) -> C1
    | ...
    | _                     -> CSTI i :: C
```

If no optimization possible, generate CSTI instruction

Some examples

```
GETBP; 0; ADD; LDI        →   GETBP; LDI

<x>; 0; EQ; IFZERO a      →   <x>; NOT; IFZERO a
```

Optimizing negations before tests

Code	Better equivalent code
NOT; NOT	<no code>
NOT; IFZERO a	IFNZRO a
NOT; IFNZRO a	IFZERO a

```
let addNOT C =
    match C with
    | NOT          :: C1 -> C1
    | IFZERO lab :: C1 -> IFNZRO lab :: C1
    | IFNZRO lab :: C1 -> IFZERO lab :: C1
    | _                   -> NOT :: C
```

`<x>; NOT; IFZERO a` → `<x>; IFNZRO a`

Optimizing stack pointer updates

Code	Better equivalent code
INCSP m1; INCSP m2	INCSP (m1+m2)
INCSP m1; RET m2	RET (m2-m1)
INCSP 0	<no code>

```
let rec addINCSP m1 C : instr list =
    match C with
    | INCSP m2              :: C1 -> addINCSP (m1+m2) C1
    | RET m2                :: C1 -> RET (m2-m1) :: C1
    | Label lab :: RET m2 :: _    -> RET (m2-m1) :: C
    | _ -> if m1=0 then C else INCSP m1 :: C
```

```
INCSP 0; RET 0
```

→

```
RET 0
```

```
INCSP -1; RET 0
```

→

```
RET 1
```

Avoiding jumps to jumps

- A conditional jump (IFZERO, IFNZRO) to some code needs a label on that code, but
  - if the code has a label already, use that
  - if the code starts with a GOTO lab, use lab

```
let addLabel C : label * instr list =
    match C with
      | Label lab :: _ -> (lab, C)
      | GOTO lab :: _  -> (lab, C)
      | _              -> let lab = newLabel()
                          in (lab, Label lab :: C)
```

Avoiding jumps to jumps and returns

- An unconditional jump (GOTO) to some code needs a label on the code, but
  - If the code has a label already, use that
  - If the code starts with a GOTO lab, use lab
  - If the code executes RET m, just do RET m

```
let makeJump C : instr * instr list =
    match C with
     | RET m                 :: _ -> (RET m, C)
     | Label lab :: RET m :: _ -> (RET m, C)
     | Label lab             :: _ -> (GOTO lab, C)
     | GOTO lab              :: _ -> (GOTO lab, C)
     | _ -> let lab = newLabel()
            in (GOTO lab, Label lab :: C)
```

`INCSP -1; GOTO L3` → `INCSP -1; RET 0` → `RET 1`

Compilation of if-statements

```
if (e)
  s1
else
  s2
```

```
    <e>
    IFZERO L1
    <s1>
    GOTO L2
L1: <s2>
L2:
```

```
let rec cStmt stmt varEnv funEnv C : instr list =
    match stmt with
    | If(e, stmt1, stmt2) ->
      let (jumpend, C1) = makeJump C            to L2
      let (labelse, C2) =
          addLabel (cStmt stmt2 varEnv funEnv C1)      L1
      in cExpr e varEnv funEnv
             (IFZERO labelse
                    :: cStmt stmt1 varEnv funEnv
                           (addJump jumpend C2))
```

Compilation of while-statements

```
while (e)
  s
```

```
    GOTO L2
L1: <s>
L2: <e>
    IFNZRO L1
```

```
let rec cStmt stmt varEnv funEnv C : instr list =
    match stmt with
    | While(e, body) ->
      let labbegin = newLabel()
      let (jumptest, C1) =
         makeJump (cExpr e varEnv funEnv
                        (IFNZRO labbegin :: C))
      in addJump jumptest (Label labbegin
                        :: cStmt body varEnv funEnv C1)
```

L1

to L2

Compiling shortcut logical expressions

- Logical expression (`m==0 && n==0`) may
  - produce a value, as in `b = (m==0&&n==0);`
  - decide a test, as in `if (m==0&&n==0) ...`

```
     <e1>
     IFZERO L1
     <e2>
     GOTO L2
L1:  0
L2:
```

Standard code for value of `e1&&e2`

When used in if (...) ...

... we can optimize it

```
     <e1>
     IFZERO L1
     <e2>
     GOTO L2
L1:  0
L2:  IFZERO L3
```

```
     <e1>
     IFZERO L3
     <e2>
     IFZERO L3
```

Compiling e1 && e2

```
and cExpr e varEnv funEnv C : instr list =
    match e with
    | Andalso(e1, e2) ->
      match C with
        | IFZERO lab :: _ ->
          cExpr e1 varEnv funEnv
              (IFZERO lab :: cExpr e2 varEnv funEnv C)
        | IFNZRO labthen :: C1 ->
          let (labelse, C2) = addLabel C1
          in cExpr e1 varEnv funEnv
              (IFZERO labelse
               :: cExpr e2 varEnv funEnv (IFNZRO labthen :: C2))
        | _ ->
          let (jumpend,  C1) = makeJump C
          let (labfalse, C2) = addLabel (addCST 0 C1)
          in cExpr e1 varEnv funEnv
               (IFZERO labfalse
                :: cExpr e2 varEnv funEnv (addJump jumpend C2))
    | Orelse(e1, e2) -> ... dual to Andalso ...
```

if (e1&&e2)

while (e1&&e2)

b = (e1&&e2)

Joint exercise 2 (code for ex13.c)

source

```
void main(int n) {
  int y;
  y = 1889;
  while (y < n) {
    y = y + 1;
    if (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0))
      print y;
  }
}
```

generated by optimizing compiler

```
INCSP 1; GETBP; CSTI 1; ADD; CSTI 1889; STI; INCSP -1;
GOTO L3; L2: GETBP; CSTI 1; ADD; GETBP; CSTI 1; ADD; LDI;
CSTI 1; ADD; STI; INCSP -1; GETBP; CSTI 1; ADD; LDI; CSTI 4;
MOD; IFNZRO L3; GETBP; CSTI 1; ADD; LDI; CSTI 100; MOD;
IFNZRO L4; GETBP; CSTI 1; ADD; LDI; CSTI 400; MOD; IFNZRO L3;
L4: GETBP; CSTI 1; ADD; LDI; PRINTI; INCSP -1; L3: GETBP;
CSTI 1; ADD; LDI; GETBP; LDI; LT; IFNZRO L2; RET 1
```

- Layout so structure is clearly visible
- Compare to code generated from Java or C# in PLC figure 9.9

Eliminating dead code

- Dead code is code that cannot be executed:

```
GOTO L1; CSTI 42; PRINTI; INCSP -1; L1:
```

Dead code

- Code following GOTO or RET is dead unless preceded by a label:

```
let rec deadcode C =
    match C with
      | []                -> []
      | Label lab :: _    -> C
      | _         :: C1   -> deadcode C1
```

Discard dead code

Why tail calls?

- TODO

Compiling tail calls

- A call is a *tail call* if it is the last action of a function, e.g. ex12.c:

```
int main(int n) {
  if (n)
    return main(n-1);
  else
    return 17;
}
```

Tail call; nothing to do after it

- In the code, a tail call is followed by RET:

```
L1: GETBP; CSTI 0; ADD; LDI; IFZERO L2;
    GETBP; CSTI 0; ADD; LDI; CSTI 1; SUB;
    CALL L1; RET 1; GOTO L3
L2: CSTI 17; RET 1;
L3: INCSP 0; RET 0
```

Tail call

Stack machine execution of TCALL

```
int main() { … g(…) … }
int g(…) { return f(…); }
int f(…) { return …; }
```

main calls g

g calls f

f returns

Ordinary call and two returns

	ret addr	old bp1	g's vars etc

CALL f

	ret addr	old bp1	g's vars etc	ret addr	old bp2	f's vars etc

RET (from f)

	ret addr	old bp1	g's vars etc

RET (from g)

Tail call and one return

	ret addr	old bp1	g's vars etc

TCALL f

	ret addr	old bp1	f's vars etc

RET (from f)

- Return from tail call goes directly to original caller

Micro-C machine call and return

- CALL creates a stack frame
- RET destroys a stack frame
- TCALL destroys one frame and creates another:

19	CALL m a	$s, v_1, .., v_m$	$\Rightarrow$	$s, \boxed{r, bp, v_1, .., v_m}$
20	TCALL m n a	$s, \boxed{r, b, u_1, .., u_n}, v_1, .., v_m$	$\Rightarrow$	$s, \boxed{r, b, v_1, .., v_m}$
21	RET m	$s, \boxed{r, b, v_1, .., v_m}, v$	$\Rightarrow$	s, v

- There is nothing to do after tail call, except return
- So the caller's stack frame can be discarded before the tail call
- So a sequence of tail calls should not require unbounded stack space

Recognizing tail calls in the compiler

- To call a function, compile arguments and emit a call:

```
and callfun f es varEnv funEnv C : instr list =
    ...
    cExprs es varEnv funEnv (makeCall argc labf C)
```

- A tail call is a call followed by RET
- Easy to see when generating code backwards:

```
let makeCall m lab C : instr list =
    match C with
    | RET n              :: C1 -> TCALL(m, n, lab) :: C1
    | Label _ :: RET n :: _    -> TCALL(m, n, lab) :: C
    | _                        -> CALL(m, lab) :: C
```

The effect of optimizations

New code for ex12.c

```
L1: GETBP; LDI; IFZERO L2;
    GETBP; LDI; CSTI 1; SUB; TCALL (1,1,"L1");
L2: CSTI 17; RET 1
```

- Compiling ex11.c with old and new comp.
- Finding 14200 solutions to 12-queen puzzle

	Code size (instr)	Running time (sec.)
Old direct compiler	792	9.06
New backwards compiler	701	8.00

Tail call optimization

- Java does not optimize tail calls
  - And JVM does not optimize tail calls
  - The security model requires stack inspection
- C# does not optimize tail calls
  - But .NET/CLI bytecode supports tail calls

```
IL_000e:  tail.
IL_0010:  callvirt int32 MyClass::MyMethod(int32)
```

- Scheme, ML, F# ... optimize tail calls
  - Needed because loops are written using tail calls

```
let rec sum xs acc =
    match xs with
      | []    -> acc
      | x::xr -> sum xr (x+acc);;
```

Preview of next week

- The Scala programming language
  - Functional, higher-order, statically typed, like F#
  - Plus object-oriented, like Java and C#
  - Generates JVM bytecode
  - Interoperates with Java libraries (eg in Eclipse)
  - Lots of innovations (and some complexity)
  - Developed by Martin Odersky, in Lausanne, CH
  - Gathering industrial interest, especially in Europe

Reading and homework

- This week's lecture:
  - PLC chapter 12
  - Exercises 12.1, 12.2, 12.3, 12.4
- Next week:
  - *Schinz, Haller: A Scala tutorial for Java programmers (2010)*
  - *Odersky et al: An overview of the Scala programming language (2006)*