

Programming Languages, F2002

Lecture 9, Wednesday 10 April 2002

- A continuation-based micro-C compiler for the abstract machine
- Optimized compilation of constants
- Optimized compilation of conditional and unconditional jumps
- Dead code elimination
- Recognizing tail calls
- Compiling tail calls to execute in constant space
- Optimized compilation of composition logical expressions

IT-C

Programming Languages, F2002

Page 9-1

Some shortcomings of the old micro-C compiler

- Clumsy code is generated, especially for `if`- and `while`-conditions (file `imp/ex19.c`):

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

This produces the following code:

```
GETBP, CSTI 0, ADD, LDI, CSTI 0, EQ, IFZERO L2,
CSTI 33, PRINTI, INCSB ^1, GOTO L3,
L2: CSTI 44, PRINTI, INCSB ^1,
L3: INCSB 0, RET 0
```

We shall improve the compiler to produce this instead:

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

- Tail-calls are not executed in constant space (example `imp/ex12.c`), and therefore run out of stack space.
- The compiler itself inefficiently appends lists of generated instructions (using `@`):

```
[GOTO labrest, label labbegin] @ cStnt body env
@ [label labtest] @ cExpr e env @ [IFNZRO labbegin]
```

IT-C

Programming Languages, F2002

Page 9-2

A better micro-C compiler

One could write a code simplifier that would remove `CSTI 0`, `ADD` from the generated program, etc. Then we could run the simplifier in a second pass after the compiler, but this is somewhat inelegant.

Instead we shall study a one-pass compiler that:

- optimizes the compilation of logical connectives (such as `!`, `&&` and `||`) into efficient control flow code;
- generates code for a logical expression `e1 && e2` that is adapted to its context of use:
 - will its value bound to a variable, as in `b = e1 && e2`;
 - or will its value be used in the condition in an `if`- or `while`-statement, as in `if (e1 && e2) ...`
- avoids generating jumps to jumps in most cases;
- eliminates most dead code, that is, instructions that cannot be executed;
- recognizes tail calls and compiles them as jumps (instruction `TCALL`).

IT-C

Programming Languages, F2002

Page 9-3

A backwards, or continuation-based, compiler

The new compiler generates code backwards, prepending new instructions to the code already generated.

The old (forwards, direct) micro-C compiler had type:

```
fun cExpr (e : expr) (env : envv) : instr list = ...
```

The new (backwards, continuation-based) compiler takes an extra argument, a code continuation:

```
fun cExpr (e : expr) (env : envv) (C : instr list) : instr list = ...
```

The code `C` already generated is a representation, at compile-time, of the run-time continuation of `e`.

This allows the compiler to perform optimizations.

Example: If `e` is `CSTI 0`, then we should generate `CSTI 1`.

However, if `C` starts with `NOT`, then we can instead throw away the `NOT` and generate `CSTI 0`.

The resulting code will be `[CSTI 0]` instead of `[CSTI 1, NOT]`.

IT-C

Programming Languages, F2002

Page 9-4

The old expression compilation function

```

fun cExpr (e : expr) (env : venv) : instr list =
  case e of
  ...
  | Cst (CSTI i) => [CSTI i]
  | Prml(ope, e1) =>
      cExpr e1 env
      @ (case ope of
        "i" => [NOT]
        | "prntc" => [PRINTC]
        | "prnti" => [PRINTI]
        | "prntc" => [PRINTC]
        | _ => raise Fail "unknown primitive 1")
  ...

Towards a new expression compilation function

fun cExpr (e : expr) (env : venv) (C : instr list) : instr list =
  case e of
  ...
  | Cst (CSTI i) => CSTI i :: C
  | Prml(ope, e1) =>
      cExpr e1 env
      (case ope of
        "i" => NOT :: C
        | "prnti" => PRINTI :: C
        | "prntc" => PRINTC :: C
        | _ => raise Fail "unknown primitive 1")
  ...

```

Optimizing the code while generating it

```

fun cExpr (e : expr) (env : venv) (C : instr list) : instr list =
  case e of
  ...
  | Cst (CSTI i) => addCST i C
  ...

The function for optimized generation of constants can be written like this:

fun addCST i C =
  case (i, C) of
  (0, EQ) :: C1 => addNOT C1
  | (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
  | (_, INCSP m) :: C1 => if m < 0 then addINCSP (m+1) C1
  | (0, IFZERO lab :: C1) => addGOTO lab C1
  | (_, IFZERO lab :: C1) => C1
  | (0, IFNZRO lab :: C1) => C1
  | (_, IFNZRO lab :: C1) => addGOTO lab C1
  | _ => CSTI i :: C

```

Correctness of the optimizations

The optimizations reflect equivalences between machine code sequences:

0, EQ	has the same meaning as	NOT	
0, ADD	has the same meaning as	<empty>	
0, SUB	has the same meaning as	<empty>	
0, NOT	has the same meaning as	1	
n , NOT	has the same meaning as	0	when $n \neq 0$
1, MUL	has the same meaning as	<empty>	
1, DIV	has the same meaning as	<empty>	
n , INCSP m	has the same meaning as	INCSP ($m - 1$)	
0, IFZERO a	has the same meaning as	GOTO a	
n , IFZERO a	has the same meaning as	<empty>	when $n \neq 0$
0, IFNZRO a	has the same meaning as	<empty>	
n , IFNZRO a	has the same meaning as	GOTO a	when $n \neq 0$

Above, <empty> is the empty sequence of instructions.

Optimization of negations and jumps

These equivalences hold for logical negation (NOT):

NOT, NOT	has the same meaning as	<empty>	if applied to boolean values
NOT, IFZERO a	has the same meaning as	IFNZRO a	
NOT, IFNZRO a	has the same meaning as	IFZERO a	

They are exploited in this code generation function:

```

fun addNOT C =
  case C of
  NOT :: C1 => C1
  | IFZERO lab :: C1 => IFNZRO lab :: C1
  | IFNZRO lab :: C1 => IFZERO lab :: C1
  | _ => NOT :: C

```

Optimization of stack pointer updates

These equivalences hold for stack pointer manipulations (INCSP m_1):

INCSP 0	has the same meaning as	(empty)
INCSP m_1 , INCSP m_2	has the same meaning as	INCSP ($m_1 + m_2$)
INCSP m_1 , RET m_2	has the same meaning as	RET ($m_2 - m_1$)

They are exploited in these code generation functions:

```
fun makeINCSP 0 C = C
  | makeINCSP n C = INCSP n :: C

fun addINCSP m1 C : instr list =
  case C of
  INCSP m2      :: C1 => makeINCSP (m1+m2) C1
  | RET m2       :: C1 => RET (m2-m1) :: C1
  | Label lab :: RET m2 :: C1 => RET (m2-m1) :: C
  | _            => makeINCSP m1 C
```

TC

Avoid generating unconditional jumps to jumps or returns

When we generate an *unconditional* GOTO to some code C, there are even better optimization opportunities.

If C begins with a RET instruction, we can return immediately rather than jump to the RET.

If C already begins with a label Lab, we can jump to Lab.

If C begins with a GOTO Lab we can jump to Lab.

Otherwise we must put a new label in front of C and generate a jump to that label:

```
fun makeJump C : instr * instr list =
  case C of
  Label lab :: RET m :: C1 => (RET m, C)
  | RET m :: C1             => (RET m, C)
  | Label lab              :: C1 => (GOTO lab, C)
  | GOTO lab               :: C1 => (GOTO lab, C)
  | _                      => let val lab = newLabel()
                               in (GOTO lab, Label lab :: C) end
```

TC

Avoid generating conditional jumps to jumps

When we generate a conditional jump (IFZERO or IFNZRO) to some code C, we must put a label in front of C.

If C already begins with a label Lab, we can jump to that label.

If C begins with GOTO Lab we can jump straight to the Lab, avoiding a jump to a jump.

Otherwise, we must put a new label in front of C, and jump to that label:

```
fun addLabel C : label * instr list =
  case C of
  Label lab :: C1 => (lab, C)
  | GOTO lab :: C1 => (lab, C)
  | _              => let val lab = newLabel()
                     in (lab, Label lab :: C) end
```

TC

The compilation of statements, part 1: if-statements

Old: <code>, IFZERO I1, <stmt1code>, GOTO I2, I1: <stmt2code>, I2: ...

```
fun cStmt stmt (env : env) (C : instr list) : instr list =
  case stmt of
  IF(e, stmt1, stmt2) =>
    let val (jumpend, C1) = makeJump C
        val (labelse, C2) = addLabel (cStmt stmt2 env C1)
    in
      cExpr e env (IFZERO labelse
                  :: cStmt stmt1 env (addJump jumpend C2))
    end
  | ...
```

Using makeJump propagates a subsequent jump (RET) into the true-branch of an if, as in imp/ex19.c.

Using addLabel improves compilation of nested ifs and absent false-branches (file imp/ex18.c):

```
void main(int m, int n) {
  if (m == 0) {
    if (n == 0)
      print 1111;
  } else
    print 3333;
}
```

TC

The compilation of statements, part 2: while-loops

Old: GOTO L2, L1: <bodycode>, L2: <ecode>, IFNZRO L1, ...

```
fun cStmt stmt (env : envv) (C : instr list) : instr list =
  case stmt of
  ...
  | While(e, body) =>
    let val labbegin = newLabel()
        val (jumptest, C1) =
          makeJump (cExpr e env (IFNZRO labbegin :: C))
        in
          addJump jumptest (Label labbegin :: cStmt body env C1)
        end
    | Expr e =>
      cExpr e env (addINCSP ~1 C)
  | ...
```

Using makeJump seems to make a difference only if the condition e is constant true.

In that case the while body will not be skipped, see file imp/ex7.c. Not a very important improvement.

The compilation of statements, part 3

Two passes are made over a Block { ... }: a forwards pass and a backwards pass.

The forwards pass is needed to determine the number of variables that the block allocates on the stack.

These must be popped, using INCSP, at the end of the block.

The backwards pass compiles the block's statements.

```
fun cStmt stmt (env : envv) (C : instr list) : instr list =
  case stmt of
  ...
  | Block stms =>
    ...
    let fun pass1 [] (_, fdepth) = ([], fdepth)
        | pass1 (s1::sr) env
          let val res1 as (_, env1) = bStmtordec s1 env
              val (res1 :: rest, fdepthr) = pass1 sr env1
              in (rest :: rest, fdepthr) end
          val (stmsback, fdepthend) = pass1 stms env
              fun pass2 []
                | pass2 ((BDec code, env) :: sr) C = code @ pass2 sr C
                | pass2 ((BStmt stmt, env) :: sr) C =
                  cStmt stmt env (pass2 sr C)
              in pass2 stmsback (addINCSP(#2 env - fdepthend) C) end
        Return NONE =>
          Return (#2 env - 1) :: deaddcode C
        Return (SOME e) =>
          Return (RET (#2 env) :: deaddcode C)
      cExpr e env (RET (#2 env) :: deaddcode C)
```

Eliminating dead code

Dead code is code that cannot be executed, such as the addition:

```
GOTO L1, 1, ADD, L1: ...
```

Code that follows an unconditional GOTO or RETURN is dead, up until the next label.

Dead code typically arises because of constant conditions in if or while; for example imp/ex7.c.

Dead code takes up space and costs nothing at runtime; but its removal can enable new optimizations.

This function removes locally dead code:

```
fun deaddcode C =
  case C of
  [] => []
  | Label lab :: _ => C
  | _ :: C1 => deaddcode C1
```

It is used when generating a GOTO or RET:

```
fun addJump jump C =
  let val C1 = deaddcode C
      in
        case (jump, C1) of
          (GOTO lab1, Label lab2 :: _) => if lab1=lab2 then C1
          else GOTO lab1 :: C1
          | _ => jump :: C1
        end;
      | _
```

Recognizing tail calls

Example imp/ex12.c.

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

The tail call main(n-1) is recognizable as a CALL immediately followed by RETURN:

```
L0: GETBP, 0, ADD, DI, IFZERO L1,
GETBP, 0, ADD, DI, 1, SUB, CALL(1, L0), RET 1, GOTO L2,
L1: 17, RET 1,
L2: INCSP 0, RET 0
```

if (n)
 main(n-1)
17

The CALL will create a new stack frame, so main will run out of stack space for large n.

Executing tail calls in the stack machine

Consider a call from g to f:

```
fun g x = ... f e1 e2 ...
```

Ordinary call and two returns

ret	old	g's vars
addr	bp	etc

CALL f

ret	old	g's vars	ret	old	f's vars
addr	bp	etc	addr	bp	etc

RET (from f)

ret	old	g's vars
addr	bp	etc

RET (from g)

Tail call and one return

ret	old	g's vars
addr	bp	etc

TCALL f

ret	old	f's vars
addr	bp	etc

RET (from f)

Effect on code size and run time

Loop executing 20 million times (file imp/ex8.c and imp/prog1):

Compiler	Code size (w)	Running time (s)
Direct	32	18.4
Backwards	22	10.9
Hand-coded	13	3.9

The *n*-queen problem (file imp/ex11.c) for *n* = 11:

Compiler	Code size (w)	Running time (s)
Direct	616	8.3
Backwards	549	7.2

Compiling tail calls

Function makeCall generates a call to an *m*-argument function at label *lab*.

If the CALL is followed by RET, or by Label and RET, then it is turned into a TCALL:

```
fun makeCall m lab C : instr list =
  case C of
    RET n          <<< C1 => TCALL(m, n, lab) :: C1
  | Label _ :: RET n :: _ => TCALL(m, n, lab) :: C
  | _               => CALL(m, lab) :: C
```

The new compiler will generate this code for imp/ex12.c:

```
L1: GETBP, LDI, FZERO L2,
    GETBP, LDI, 1, SUB, TCALL(1, 1, L1),
L2: 17, RET 1
                                     IF (n)
                                     main(n-1)
                                     17
```

The TCALL will not create a new stack frame, so main can be executed for arbitrarily large *n*.

Compiling logical expressions to control flow

The value of a logical expression (`m == 0 && n == 0`) may be used in two different ways:

- Its value may be assigned to a variable, or returned from a function:

```
b = (m == 0 && n == 0);
```

Here, 0 or 1 should be assigned to *b*.

- Its value may be used in an if- or while-condition:

```
if (m == 0 && n == 0) ...;
```

Here there is no need to first generate a 0 or 1 and then test it using IFZERO or IFNZRO.

The old forwards compiler compiles a logical expression `x == 0` to the same code regardless of its use.

The new backwards compiler compiles it differently depending on the context; see imp/ex20.c:

```
void main(int m, int n) {
  int b;
  if (m == 0 && n == 0)
    print 1111;
  else
    print 2222;
  b = (m == 0 && n == 0);
}
```

