

## Programming Languages, F2005

### Lecture 9, Wednesday 13 April 2005

- 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

### 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;
- that generates code for a logical expression `e1 && e2` that is adapted to its context of use:
  - will its value be 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) ...`;
- that avoids generating jumps to jumps in most cases;
- that eliminates most dead code, that is, instructions that cannot be executed;
- that recognizes tail calls and compiles them as jumps (instruction `TCALL`).

### 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, INCSP ~1, GOTO L3,
L2: CSTI 44, PRINTI, INCSP ~1,
L3: INCSP 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 uses `@` to append lists of generated instructions:

```
[GOTO labtest, Label labbegin] @ cStmt body env
@ [Label labtest] @ cExpr e env @ [IFNZRO labbegin]
```

### 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 : lenv) (fenv : fenv) : instr list = ...
```

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

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

The argument `C` contains the code, already generated, that follows the code for `e`.

Thus `C` is a compile-time representation of the run-time continuation for expression `e`.

This allows the compiler to perform optimizations.

**Example:** If `e` is `CstI 1`, 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]`.

### The old expression compilation function

```

fun cExpr (e : expr) (env : lenv) (fenv : fenv) : instr list =
  case e of
  ...
  | CstI i      => [CSTI i]
  | Prml(ope, e1) =>
    cExpr e1 env fenv
    @ (case ope of
      "!"      => [NOT]
      "printi" => [PRINTI]
      "printc" => [PRINTC]
      _        => raise Fail "unknown primitive 1")
  | ...

```

### Towards a new expression compilation function: generating the same code in a different way

```

fun cExpr (e : expr) (env : lenv) (fenv : fenv) (C : instr list) : instr list =
  case e of
  ...
  | CstI i      => CSTI i :: C
  | Prml(ope, e1) =>
    cExpr e1 env fenv
    (case ope of
      "!"      => NOT :: C
      "printi" => PRINTI :: C
      "printc" => PRINTC :: C
      _        => raise Fail "unknown primitive 1")
  | ...

```

### Optimizing the code while generating it (imp/contcomp.sml)

Do not blindly generate CSTI, but introduce a function addCST to inspect the continuation code C:

```

fun cExpr (e : expr) (env : lenv) (fenv : fenv) (C : instr list) : instr list =
  case e of
  ...
  | CstI i => addCST i C
  | ...

```

The addCST function generates better code, functionally equivalent for the given code continuation:

```

fun addCST i C =
  case (i, C) of
  (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
  (_, INCSP m  :: C1) => if m < 0 then addINCSP (m+1) C1
                        else CSTI i :: C
  (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

```

Using this technique, we get the code [CSTI 1, PRINTI] from the source print(! 0).

### Example of backwards code generation

Let's compile the expression print(! 0).

In abstract syntax that is Prml("printi", Prml("!", CstI 0)).

Backwards compilation goes like this:

Expression e	Code continuation C
Prml("printi", Prml("!", CstI 0))	[]
Prml("!", CstI 0)	[PRINTI]
CstI 0	[NOT, PRINTI]
	[CSTI 1, NOT, PRINTI]

Observation: When compiling the constant CstI 0, we know that at runtime it will be negated.

The result of that will always be 1. Why not generate CstI 1 outright and avoid the negation?

### Correctness of the optimizations

The optimizations reflect equivalences between machine code sequences:

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)
0, EQ	has the same meaning as	NOT
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
```

This means that `if (!x) print(42);` compiles to

```
GETBP, LDI, IFNZRO L2, CSTI 42, PRINTI, ..., L2:
```

rather than

```
GETBP, LDI, NOT, IFZERO L2, CSTI 42, PRINTI, ..., L2:
```

### 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 :: _ => (lab, C)
  | GOTO lab :: _ => (lab, C)
  | _             => let val lab = newLabel()
                    in (lab, Label lab :: C) end
```

### Optimization of stack pointer updates

These equivalences hold for stack pointer manipulations (INCSP  $m$ ):

---

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

---

They are exploited in this code generation function:

```
fun addINCSP m1 C : instr list =
  case C of
    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
```

This means that the body of `void main(int x) { int y; }` gets compiled to

```
CSTI 0, RET 1
```

rather than

```
CSTI 0, INCSP ~1, RET 0
```

### 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
    RET m      :: _ => (RET m, C)
  | Label lab :: RET m :: _ => (RET m, C)
  | Label lab  :: _ => (GOTO lab, C)
  | GOTO lab   :: _ => (GOTO lab, C)
  | _         => let val lab = newLabel()
                in (GOTO lab, Label lab :: C) end
```

This and the previous optimization mean that `if (x) if (y) print(42);` gets compiled to

```
x, IFZERO L2, y, IFZERO L3, CSTI 42, PRINTI, RET 2,
L3: RET 1, L2: RET 1
```

rather than

```
x, IFZERO L2, y, IFZERO L4, CSTI 42, PRINTI, INCSP ~1, GOTO L5,
L4: INCSP 0, L5: GOTO L3,
L2: INCSP 0, L3: INCSP 0, RET 1
```

### The compilation of statements, part 1: if-statements

The old compilation scheme:

```
if (e)
  stmt1
else
  stmt2
      <ecode>
      IFZERO L1;
      <stmt1code>
      GOTO L2;
      L1: <stmt2code>
      L2:
```

The new compilation scheme:

```
fun cStmt stmt (env : lenv) (fenv : fenv) (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 fenv C1)
    in
      cExpr e env fenv (IFZERO labelse
        :: cStmt stmt1 env fenv (addJump jumpend C2))
    end
  | ...
```

Using makeJump propagates a subsequent jump or RET into the true-branch.

For instance, if (x == 0) print 33; else print 44; compiles to

```
x, IFNZRO "L2", CSTI 33, PRINTI, RET 1,
L2: CSTI 44, PRINTI, RET 1
```

### 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 : lenv) (fenv : fenv) (C : instr list) : instr list =
  case stmt of
  | ...
  | Block stmts =>
    let fun pass1 [] (_, fdepth) = ([], fdepth)
        | pass1 (s1::sr) env =
          let val res1 as (_, env1) = bStmtdordec s1 env
              val (resr, fdepthr) = pass1 sr env1
          in (res1 :: resr, fdepthr) end
        val (stmtsback, fdepthend) = pass1 stmts env
        fun pass2 [] C = C
        | pass2 ((BDec code, env) :: sr) C = code @ pass2 sr C
        | pass2 ((BStmt stmt, env) :: sr) C =
          cStmt stmt env fenv (pass2 sr C)
    in pass2 stmtsback (addINCSP(#2 env - fdepthend) C) end
  | Return NONE =>
    RET (#2 env - 1) :: deadcode C
  | Return (SOME e) =>
    cExpr e env fenv (RET (#2 env) :: deadcode C)
```

### The compilation of statements, part 2: while-loops

The old compilation scheme:

```
while (e)
  body
      GOTO L2;
      L1: <bodycode>
      L2: <ecode>
      IFNZRO L1;
```

The new compilation scheme:

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

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.

### 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 removing it can enable further optimizations.

This function removes locally dead code:

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

It is used when generating a GOTO or RET:

```
fun addJump jump C =
  let val C1 = deadcode 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

Function `main` in example `imp/ex12.c` calls itself by a tail-call:

```
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, LDI, IFZERO L1,          if (n)
    GETBP, 0, ADD, LDI, 1, SUB, CALL(1, L0), RET 1, GOTO L2, main(n-1)
L1: 17, RET 1,                               17
L2: INCSP 0, RET 0
```

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

### 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, IFZERO L2,          if (n)
    GETBP, LDI, 1, SUB, TCALL(1, 1, L1), main(n-1)
L2: 17, RET 1                       17
```

The TCALL will not create a new stack frame, so `main` in `imp/ex12.c` can be executed for arbitrarily large `n`.

In general, the tail call optimization is unsound in C; see example `imp/ex21.c`.

If a function allocates an array on stack and passes it to a function called by a tail call, the array will be overwritten!

In Java, arrays and objects are on the heap and the tail call optimization is sound (except for security).

### 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	bp1	etc

CALL f

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

RET (from f)

ret	old	g's vars
addr	bp1	etc

RET (from g)

--

#### Tail call and one return

ret	old	g's vars
addr	bp1	etc

TCALL f

ret	old	f's vars
addr	bp1	etc

RET (from f)

--

### Effect of compiler optimizations 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  $n$ -queens problem (file `imp/ex11.c`) for  $n = 11$ :

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

### 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 generates the same code for a logical expression `x == 0` 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);
}
```

### A potentially simpler idea that doesn't quite work

The compilation of `e1 && e2` and `e1 || e2` is rather complicated.

Couldn't one just implement `e1 ? e2 : e3` in the compiler, and use these equivalences:

---

---

```
e1 && e2 has the same meaning as e1 ? e2 : false
```

```
e1 || e2 has the same meaning as e1 ? true : e2
```

---

---

The micro-C parser could build the representation `e1 ? e2 : false` for `e1 && e2`, etc.

Then we could delete `Andalso` and `Orelse` from the abstract syntax.

But this does not work well, because

```
true ? 1111 : 2222
```

compiles to

```
    CSTI 1111, GOTO L1,
L2:  CSTI 2222
L1:
```

and it is hard to recognize on-the-fly that the code at `L2` can never be reached.

### Compilation and optimization of composite logical expressions (somewhat complicated)

The logical expression `e1 && e2` is compiled to `<e1>, IFZERO Lf, <e2>, GOTO Le, Lf: 0, Le:`

But if this is followed by `IFZERO lab`, it can be optimized to `<e1>, IFZERO lab, <e2>, IFZERO lab`.

```
fun cExpr (e : expr) (env : lenv) (fenv : fenv) (C : instr list) : instr list =
  case e of
  ...
  | Andalso(e1, e2) =>
    (case C of
    IFZERO lab :: _ =>
      cExpr e1 env fenv (IFZERO lab :: cExpr e2 env fenv C)
    | IFNZRO labthen :: C1 =>
      let val (labelse, C2) = addLabel C1
      in
        cExpr e1 env fenv
        (IFZERO labelse
         :: cExpr e2 env fenv (IFNZRO labthen :: C2))
      end
    | _ =>
      let val (jumpend, C1) = makeJump C
          val (labfalse, C2) = addLabel (addCST 0 C1)
      in
        cExpr e1 env fenv
        (IFZERO labfalse
         :: cExpr e2 env fenv (addJump jumpend C2))
      end
    | Orelse(e1, e2) => ... dual to Andalso ...
```