

- Some shortcomings of the old micro-C compiler**
- Clumsy code is generated, especially for `if`- and `while`-conditions (file `tmp/exl9.c`):
- ```
void main(int x) {
    if (x == 0) print 33; else print 44;
}
```
- This produces the following code:
- ```
GETBP, C$TI 0, ADD, LDI, C$TI 0, EQ, I$ZERO L2,
C$TI 33, PRINT, INCS$P ~1, GOTO L3,
L2: C$TI 44, PRINT, INCS$P ~1,
L3: INCS$P 0, RET 0
```
- We shall improve the compiler to produce this instead:
- ```
GETBP, LDI, I$NZRO L2,
C$TI 33, PRINT, RET 1,
L2: C$TI 44, PRINT, RET 1
```
- Tail-calls are not executed in constant space (example `tmp/exl2.c`), and therefore run out of stack space.
  - The compiler itself inefficiently uses `@` to append lists of generated instructions:
 

```
[GOTO labtest, Label labbegin] @ c$tm body env
@ [Label labtest] @ c$expr e env @ [I$NZRO 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 c$expr (e : expr) (env : l$nv) (fenv : f$nv) (C : instr list) = ...
: instr list = ...
```
- The new (backwards, continuation-based) compiler takes an extra argument, a code continuation:
- ```
fun c$expr (e : expr) (env : l$nv) (fenv : f$nv) (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 `C$TI 1`, then we should generate `C$TI 1`. However, if `C` starts with `NOT`, then we can instead throw away the `NOT` and generate `C$TI 0`. The resulting code will be `[C$TI 0]` instead of `[C$TI 1, NOT]`.

- Programming Languages, F2004**  
**Lecture 9, Wednesday 14 April 2004**
- 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 `C$TI 0`, `ADD` from the generated program, etc. Then we could run the simplifier in a second pass after the compiler, but this is somewhat inelegant. Instead we shall study a one-pass compiler:
- that optimizes the compilation of logical connectives (such as `!`, `&&` and `||`) into efficient control flow code;
  - that generates code for a logical expression `e1 && e2` that is adapted to its context of use:
    - will its value be bound to a variable, as in `b = e1 && e2`;
    - or will its value be used in the condition in an `if`- or `while`-statement, as in `if (e1 && e2) ...`;
  - that avoids generating jumps to jumps in most cases;
  - that eliminates most dead code, that is, instructions that cannot be executed;
  - that recognizes tail calls and compiles them as jumps (instruction `T$CALL`).



**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
  | _ => (Lab, C)
  | Label Lab :: _ => (Lab, C)
  | _ => let val Lab = newLabel()
         in (Lab, Label Lab :: C) end
  end

```

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

The old compilation scheme:

```

<code>
if (e)
  stmt1
  IFZERO l1!
<stmt1code>
else
  stmt2
  GOTO l2!
<stmt2code>
end

```

The new compilation scheme:

```

fun stmt stmt (env : Env) (lenv : Env) (C : Instr list) : Instr list =
  case stmt of
  | If(e, stmt1, stmt2) =>
    let val (jumpend, C1) = makeJump C
        val (labelse, C2) = addLabel (stmt stmt2 env lenv C1)
    in
      expr e env lenv (IFZERO labelse
        :: stmt stmt1 env lenv (addJump jumpend C2))
    end
  | ...
  end

```

Using makeJump propagates a subsequent jump or RET into the true-branch, as in jmp/exl9.c.

**Optimization of stack pointer updates**

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

|                                             |                         |                                          |
|---------------------------------------------|-------------------------|------------------------------------------|
| INCSP 0                                     | has the same meaning as | (empty)                                  |
| INCSP m <sub>1</sub> , INCSP m <sub>2</sub> | has the same meaning as | INCSP (m <sub>1</sub> + m <sub>2</sub> ) |
| INCSP m <sub>1</sub> , RET m <sub>2</sub>   | has the same meaning as | RET (m <sub>2</sub> - m <sub>1</sub> )   |

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
  | RET m2
  | Label Lab :: _ => RET m2
  | _ => makeINCSP m1 C
  end

```

**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, C)
  | Label Lab
  | GOTO Lab, C)
  | _ => (GOTO Lab, C)
  | GOTO Lab, C)
  | _ => let val Lab = newLabel()
         in (GOTO Lab, Label Lab :: C) end
  end

```

**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.

The fun stmt (env : lenv) (fenv : fenv) (C : instr list) : instr list =

```

fun stmt of
  case stmt of
    Block stmts =>
      let fun pass1 [ (, fdepth) = ([], fdepth)
          | pass1 (s1::sr) env =
            let val res1 as (, env1) = bstmtoordc s1 env
                in (res1, fdepthr) = pass1 sr env1
                val (resr, fdepthr) = pass1 sr env1
                in (res1, fdepthr) end
            fun pass2 []
              | 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 stmts bck (addINCSP(#2 env - fdepthend) C) end
          | Return NONE =>
            Return NONE
          | Return (#2 env - 1) :: deadcode C
            | Return (SOME e) =>
              cdxpr e env fenv (RET (#2 env) :: deadcode C)
  end

```

**Recognizing tail calls**

Function main in example tmp/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,
    GETBP, 0, ADD, LDI, 1, SUB, CALL(L1, L0), RET 1, GOTO L2,
L1: 17, RET 1,
L2: INCSP 0, RET 0

```

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

**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 stmt (env : lenv) (fenv : fenv) (C : instr list) : instr list =
  case stmt of
    while(e, body) =>
      let val labbegin = newLabel()
          val (jumpst, C1) =
            makedump (cdxpr e env fenv (IFNZRO labbegin :: C))
          addjump jumpst (Label labbegin :: cstmt body env fenv C1)
          end
      | ...
  end

```

Using makedump 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 tmp/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 tmp/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
    | _ =>
      let fun addjump jump C =
            let val C1 = deadcode C
                in (* jump is GOTO or RET *)
            end;
          case (jump, C1) of
            (GOTO lab1, Label lab2 :: _ :: C1) => if lab1=lab2 then C1
            else GOTO lab1 :: C1
          | _ => jump :: C1
      end;

```

In Java, arrays and objects are on the heap and the tail call optimization is sound (except for security).  
 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 general, the tail call optimization is unsound in C; see example tmp/ex21.c.

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

```

L1: GETBP, LDI, IPZERO L2,
    GETBP, LDI, 1, SUB, TCALL(L1, 1, L1),
L2: L7, RET L1
    if (n)
        main(n-1)
    L7
    
```

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

```

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

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

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

**Compiling tail calls**

```

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

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

The old forwards compiler generates the same code for a logical expression x == 0 regardless of its use.

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

```

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

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

Here, 0 or 1 should be assigned to b.

```

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

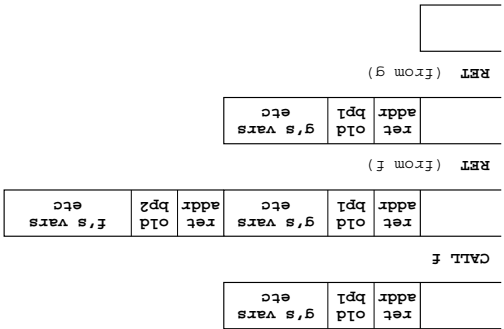
- Its value may be assigned to a variable, or returned from a function:

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

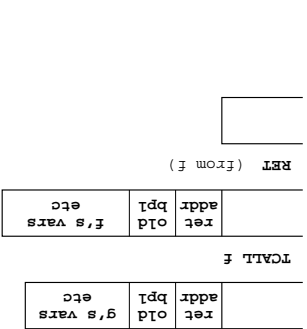
**Compiling logical expressions to control flow**

**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**



**Tail call and one return**



**Effect of compiler optimizations on code size and run time**

Loop executing 20 million times (file tmp/ex8.c and tmp/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 tmp/ex11.c) for n = 11:

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

**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 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 =
  ...
  | AndAlso(e1, e2) =>
    ...
    IFZERO lab :: - =>
      cexpr e1 env fenv (IFZERO lab :: C2)
    end
    | - =>
      let val (jumpend, C1) = maketump C
          val (labfalse, C2) = addlabel (addCST 0 C1)
          in
            cexpr e1 env fenv (IFZERO labfalse (addjump jumpend C2))
          end
    | OrElse(e1, e2) => ... dual to AndAlso ...
    
```