

Chapter 12

A locally optimizing compiler

In this chapter we shall see that thinking in continuations is beneficial also when compiling micro-C to stack machine code. Generating stack machine code backwards may seem silly, but it enables the compiler to inspect the code that will consume the result of the code being generated. This permits the compiler to perform many optimizations (code improvement) easily.

12.1 What files are provided for this chapter

In addition to the micro-C files mentioned in Section 7.1, the following file is provided:

File	Contents
MicroC/Contcomp.fs	compile micro-C backwards

12.2 Generating optimized code backwards

In Chapter 8 we compiled micro-C programs to abstract machine code for a stack machine, but the code quality was poor, with many jumps to jumps, addition of zero, tests of constants, and so on.

Here we present a simple optimizing compiler that optimizes the code on the fly, while generating it. The compiler does not rely on advanced program analysis or program transformation. Instead it combines local optimizations (so-called peephole optimizations) with backwards code generation.

In backwards code generation, one uses a ‘compile-time continuation’ to represent the instructions following the code currently being generated. The compile-time continuation is simply a list of the instructions that will follow the current one. At run-time, those instructions represent the continuation of the code currently being generated: that continuation will consume any result produced (on the stack) by the current code.

Using this approach, a one-pass compiler:

- can optimize the compilation of logical connectives (such as `!`, `&&` and `||`) into efficient control flow code;
- can generate code for a logical expression `e1 && e2` that is adapted to its context of use:

- will the logical expression’s value be bound to a variable:

```
b = e1 && e2
```

- or will be used as the condition in an if- or while-statement:

```
if (e1 && e2) ...;
```

- can avoid generating jumps to jumps in most cases;
- can eliminate some dead code (instructions that cannot be executed);
- can recognize tail calls and compile them as jumps (instruction `TCALL`) instead of proper function calls (instruction `CALL`), so that a tail-recursive function will execute in constant space.

Such optimizations might be called backwards optimizations: they exploit information about the ‘future’ of an expression: the use of its value. Forwards optimizations, on the other hand, would exploit information about the ‘past’ of an expression: its value. A forwards optimization may for instance exploit that a variable has a particular constant value, and use that value to simplify expressions in which the variable is used (constant propagation). This is possible only to a very limited extent in backwards code generation.

12.3 Backwards compilation functions

In the old forwards compiler from Chapter 8, the compilation function `cExpr` for micro-C expressions had the type

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

In the backwards compiler, it has this type instead:

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

The only change is that an additional argument of type `instr list`, that is, list of instructions, has been added; this is the code continuation `C`. All other compilation functions (`cStmt`, `cAccess`, `cExprs`, and so on, listed in Figure 8.4) are modified similarly.

To see how the code continuation is used, consider the compilation of simple expressions such as constants `CstI i` and unary (one-argument) primitives `Priml("!", e1)`.

In the old forwards compiler, code fragments are generated as instruction lists and are concatenated together using the append operator (`@`):

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr list =
  match e with
  | ...
  | CstI i          -> [CSTI i]
  | Priml(ope, e1) ->
    cExpr e1 varEnv funEnv
    @ (match ope with
      | "!"        -> [NOT]
      | "printi"   -> [PRINTI]
      | "printc"   -> [PRINTC]
      | _          -> raise (Failure "unknown primitive 1"))
  | ...
```

For instance, the expression `!false`, which is `Priml("!", CstI 0)` in abstract syntax, is compiled to `[CSTI 0] @ [NOT]`, that is, `[CSTI 0; NOT]`.

In a backwards (continuation-based) compiler, the corresponding compiler fragment would look like this:

```
and cExpr (e : expr) varEnv funEnv (C : instr list) : instr list =
  match e with
  | ...
  | CstI i          -> CSTI i :: C
  | Priml(ope, e1) ->
    cExpr e1 varEnv funEnv
    (match ope with
     | "!"        -> addNOT C
     | "printi"   -> PRINTI :: C
     | "printc"   -> PRINTC :: C
     | _          -> failwith "unknown primitive 1")
  | ...
```

So the new instructions generated are simply stuck onto the front of the code `C` already generated. This in itself achieves nothing, except that it avoids using the append function `@` on the generated instruction lists, which can be costly. The code generated for `!false` is `CSTI 0 :: [NOT]` with `is [CSTI 0; NOT]` as before.

12.3.1 Optimizing expression code while generating it

Now that the code continuation `C` is available, we can use it to optimize (improve) the generated code. For instance, when the first instruction in `C` (which is the next instruction to be executed at run-time) is `NOT`, then there is no point in generating the instruction `CSTI 0`; the `NOT` will immediately turn the zero into a one. Instead we should generate the constant `CSTI 1`, and throw away the `NOT` instruction. We can easily modify the expression compiler `cExpr` to recognize such special situations, and generate optimized code:

```
and cExpr (e : expr) varEnv funEnv (C : instr list) : instr list =
  match e with
  | ...
  | CstI i -> match (i, C) with
              | (0, NOT :: C1) -> CSTI 1 :: C1
              | (_, NOT :: C1) -> CSTI 0 :: C1
              | _                -> CSTI i :: C
  | ...
```

With this scheme, the code generated for `!false` will be `[CSTI 1]`, which is shorter and faster.

In practice, we introduce an auxiliary function `addCST` to take care of these optimizations, both to avoid cluttering up the main functions, and because constants (`CSTI`) are generated in several places in the compiler:

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

The `addCST` function is defined by straightforward pattern matching:

```
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
| (_, 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

```

Note in particular that instead of generating [CSTI 0; IFZERO lab] this will generate an unconditional jump [GOTO lab]. This optimization turns out to be very useful in conjunction with other optimizations.

The auxiliary functions addNOT, addINCSP, and addGOTO generate NOT, INCSP, and GOTO instructions, inspecting the code continuation C to optimize the code if possible.

An attractive property of these local optimizations is that one can easily see that they are correct. Their correctness depends only on some simple code equivalences for the abstract stack machine, which are quite easily proven by considering the state transitions of the abstract machine shown in Figure 8.1.

Concretely, the function addCST above embodies these instruction sequence equivalences:

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 ($m + 1$)	when $m < 0$
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 $n \neq 0$

Additional equivalences are used in other optimizing code-generating functions (addNOT, makeINCSP, addINCSP, addGOTO):

NOT, NOT	has the same meaning as	<code><empty></code> (see Note)
NOT, IFZERO <i>a</i>	has the same meaning as	IFNZRO <i>a</i>
NOT, IFNZRO <i>a</i>	has the same meaning as	IFZERO <i>a</i>
INCSP 0	has the same meaning as	<code><empty></code>
INCSP <i>m</i> ₁ , INCSP <i>m</i> ₂	has the same meaning as	INCSP (<i>m</i> ₁ + <i>m</i> ₂)
INCSP <i>m</i> ₁ , RET <i>m</i> ₂	has the same meaning as	RET (<i>m</i> ₂ - <i>m</i> ₁)

Note: The NOT, NOT equivalence holds when the resulting value is used as a boolean value: that is, when no distinction is made between 1 and other non-zero values. The code generated by our compiler satisfies this requirement, so it is safe to use the optimization.

12.3.2 The old compilation of jumps

To see how the code continuation is used when optimizing jumps (instructions GOTO, IFZERO, IFNZRO), consider the compilation of a conditional statement:

```
if (e) stmt1 else stmt2
```

The old forwards compiler (file `MicroC/Comp.fs`) used this compilation scheme:

```
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]
```

The above compiler fragment generates various code pieces (instruction lists) and concatenates them to form code such as this:

```
<e> IFZERO L1
<stmt1> GOTO L2
L1: <stmt2>
L2:
```

where `<e>` denotes the code generated for expression `e`, and similarly for the statements.

A plain backwards compiler would generate exactly the same code, but do it backwards, by sticking new instructions in front of the instruction list `C`, that is, the compile-time continuation:

```
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 :: C)))

```

12.3.3 Optimizing a jump while generating it

The continuation-based compiler fragment above unconditionally generates new labels and jumps. But if the instruction after the if-statement is GOTO L3, then it would wastefully generate a jump to a jump:

```

<e> IFZERO L1
<stmt1> GOTO L2
L1: <stmt2>
L2: GOTO L3

```

One should much rather generate GOTO L3 than the GOTO L2 which leads directly to a new jump. (Jumps slow down pipelined processors considerably because they cause instruction pipeline stalls. So-called branch prediction logic in modern processors mitigates this effect to some degree, but still it is better to avoid excess jumps.) Thus instead of mindlessly generating a new label labend and a GOTO, we call an auxiliary function `makeJump` that checks whether the first instruction of the code continuation `C` is a GOTO (or a return RET or a label) and generates a suitable jump instruction `jumpend`, adding a label to `C` if necessary, giving `C1`:

```
let (jumpend, C1) = makeJump C
```

The `makeJump` function is easily written using pattern matching. If `C` begins with a return instruction RET (possibly below a label), then `jumpend` is RET; if `C` begins with label lab or GOTO lab, then `jumpend` is GOTO lab; otherwise, we invent a new label lab and then `jumpend` is GOTO lab:

```

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)

```

Similarly, we need to stick a label in front of `<stmt2>` above only if there is no label (or GOTO) already, so we use a function `addLabel` to return a label labelse, possibly sticking it in front of `<stmt2>`:

210 *Backwards compilation functions*

```
let (labelse, C2) = addLabel (cStmt stmt2 varEnv funEnv C1)
```

Note that C1 (that is, C possibly preceded by a label) is the code continuation of stmt2.

The function `addLabel` uses pattern matching on the code continuation C to decide whether a label needs to be added. If C begins with a `GOTO lab` or `label lab`, we can just reuse lab; otherwise we must invent a new label:

```
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)
```

Finally, when compiling an if-statement with no else-branch:

```
if (e)  
  stmt
```

we do not want to get code like this, with a jump to the next instruction:

```
<e> IFZERO L1  
<stmt1> GOTO L2  
L1:  
L2:
```

to avoid this, we introduce a function `addJump` which recognizes this situation and avoids generating the `GOTO`.

Putting everything together, we have this optimizing compilation scheme for an if-statement `If(e, stmt1, stmt2)`:

```
let (jumpend, C1) = makeJump C  
let (labelse, C2) = addLabel (cStmt stmt2 varEnv funEnv C1)  
in cExpr e varEnv funEnv (IFZERO labelse  
  :: cStmt stmt1 varEnv funEnv (addJump jumpend C2))
```

This gives a flavour of the optimizations performed for if-statements. Below we show how additional optimizations for constants improve the compilation of logical expressions.

12.3.4 Optimizing logical expression code

As in the old forwards compiler (file `MicroC/Comp.fs`) logical non-strict connectives such as `&&` and `||` are compiled to conditional jumps, not to special instructions that manipulate boolean values.

Consider the example program in file `MicroC/ex13.c`. It prints the leap years between 1890 and the year `n` entered on the command line:

```
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;
    }
}
```

The non-optimizing forwards compiler generates this code for the while-loop:

```
GOTO L3
L2: GETBP; 1; ADD; GETBP; 1; ADD; LDI; 1; ADD; STI; INCSP -1;    y=y+1
    GETBP; 1; ADD; LDI; 4; MOD; 0; EQ; IFZERO L9;              y%4==0
    GETBP; 1; ADD; LDI; 100; MOD; 0; EQ; NOT; GOTO L8;         y%100!=0
L9: 0;
L8: IFNZRO L7; GETBP; 1; ADD; LDI; 400; MOD; 0; EQ; GOTO L6;   y%400==0
L7: 1;
L6: IFZERO L4; GETBP; 1; ADD; LDI; PRINTI; INCSP -1; GOTO L5;  print y
L4: INCSP 0;
L5: INCSP 0;
L3: GETBP; 1; ADD; LDI; GETBP; 0; ADD; LDI; LT; IFNZRO L2;    y<n
```

The above code has many deficiencies:

- an occurrence of `0; ADD` could be deleted, because $x + 0$ equals x
- `INCSP 0` could be deleted (twice)
- two occurrences of `0; EQ` could be replaced by `NOT`, or the subsequent test could be inverted
- if `L9` is reached, the jump to `L7` at `L8` will not be taken; so instead of going to `L9` one could go straight to the code following `IFNZRO L7`
- similarly, if `L7` is reached, the jump to `L4` at `L6` will not be taken; so instead of going to `L7` one could go straight to the code following `IFZERO L4`

212 *Backwards compilation functions*

- instead of executing GOTO L8 followed by IFNZRO L7 one could execute IFNZRO L7 right away.

The optimizing backwards compiler solves all those problems, and generates this code:

```
GOTO L3;
L2: GETBP; 1; ADD; GETBP; 1; ADD; LDI; 1; ADD; STI; INCSP -1;   y=y+1
    GETBP; 1; ADD; LDI; 4; MOD; IFNZRO L5;                   y%4==0
    GETBP; 1; ADD; LDI; 100; MOD; IFNZRO L4;                 y%100!=0
L5: GETBP; 1; ADD; LDI; 400; MOD; IFNZRO L3;                 y%400==0
L4: GETBP; 1; ADD; LDI; PRINTI; INCSP -1;                   print y
L3: GETBP; 1; ADD; LDI; GETBP; LDI; LT; IFNZRO L2;           y<n
```

To see that the compilation of a logical expression adapts itself to the context of use, contrast this with the compilation of the function `leapyear` (file `MicroC/ex22.c`):

```
int leapyear(int y) {
    return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
}
```

which returns the value of the logical expression instead of using it in a conditional:

```
L2: GETBP; LDI; 4; MOD; IFNZRO L6;                           y%4==0
    GETBP; LDI; 100; MOD; IFNZRO L5;                         y%100!=0
L6: GETBP; LDI; 400; MOD; NOT; RET 1;                         y%400==0
L5: 1; RET 1                                                 true
```

The code between L2 and L6 is essentially the same as before, but the code following L6 is different: it leaves a (boolean) value on the stack top and returns from the function.

12.3.5 Eliminating dead code

Instructions that cannot be executed are called dead code. For instance, the instructions immediately after an unconditional jump (GOTO or RET) cannot be executed, unless they are preceded by a label. We can eliminate dead code by throwing away all instructions after an unconditional jump, up to the first label after that instruction (function `deadcode`). This means that instructions following an infinite loop are thrown away (file `MicroC/ex7.c`):

```

while (1) {
    i = i + 1;
}
print 999999;

```

The following code is generated for the loop:

```
L2: GETBP; GETBP; LDI; 1; ADD; STI; INCSP -1; GOTO L2
```

where the statement `print 999999` has been thrown away, and the loop conditional has been turned into an unconditional `GOTO`.

12.3.6 Optimizing tail calls

As we have seen before, a tail call `f(...)` occurring in function `g` is a call that is the last action of the calling function `g`. That is, when the called function `f` returns, function `g` will do nothing more before it too returns.

In code for the abstract stack machine from Section 8.2, a tail call can be recognized as a call to `f` that is immediately followed by a return from `g`: a `CALL` instruction followed by a `RET` instruction. For example, consider this program with a tail call from `main` to `main` (file `MicroC/ex12.c`):

```

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

```

The code generated by the old forwards compiler is

```

L1: GETBP; 0; ADD; LDI; IFZERO L2;                if (n)
    GETBP; 0; ADD; LDI; 1; SUB; CALL (1,L1); RET 1; GOTO L3;    main(n-1)
L2: 17; RET 1;                                    17
L3: INCSP 0; RET 0

```

The tail call is apparent as `CALL(1, L0); RET`. Moreover, the `GOTO L3` and the code following label `L3` are *unreachable*: those code fragments cannot be executed, but that's less important.

When function `f` is called by a tail call in `g`:

```

void g(...) {
    ... f(...) ...
}

```

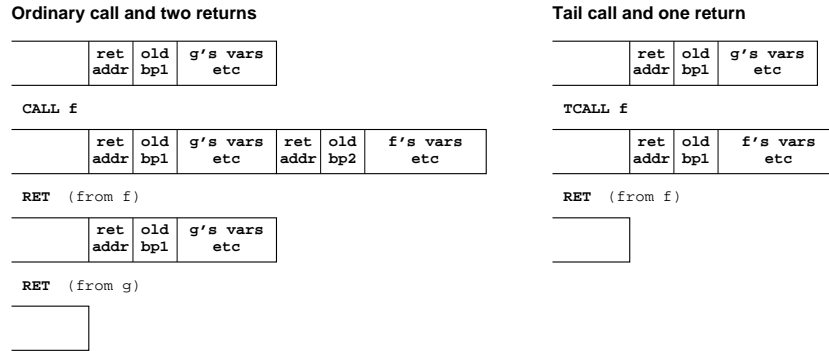


Figure 12.1: Example replacement of a call and a return by a tail call.

then if the call to f ever returns to g , it is necessarily because of a `RET` instruction in f , so two `RET` instructions will be executed in sequence:

```
CALL m f; ... RET k; RET n
```

The tail call instruction `TCALL` of our stack machine has been designed so that the above sequence of executed instructions is equivalent to this sequence of instructions:

```
TCALL m n f; ... RET k
```

This equivalence is illustrated by an example in Figure 12.1. More formally, Figure 12.2 uses the stack machine rules to show that the equivalence holds between two sequences of executed instructions:

$$\text{CALL } m \ a; \dots \text{RET } k; \text{RET } n \equiv \text{TCALL } m \ n \ a; \dots; \text{RET } k$$

provided function f at address a transforms $s, r_2, b_2, v_1, \dots, v_m$ into $s, r_2, b_2, w_1, \dots, w_k, v$ without using the lower part s of the stack at all.

The new continuation-based compiler uses an auxiliary function `makeCall` to recognize tail calls:

```
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
```

It will compile the above example function `main` to the following abstract machine code, in which the recursive call to `main` has been recognized as a tail call and has been compiled as a `TCALL`:

Stack	Action
Ordinary call and two returns	
$s, r_1, bp_1, u_1, \dots, u_n, v_1, \dots, v_m$	CALL m f (at address $r_2 - 1$)
$\Rightarrow s, r_1, bp_1, u_1, \dots, u_n, r_2, bp_2, v_1, \dots, v_m$	code in the body of f
$\Rightarrow s, r_1, bp_1, u_1, \dots, u_n, r_2, bp_2, w_1, \dots, w_k, v$	RET k
$\Rightarrow s, r_1, bp_1, u_1, \dots, u_n, v$	RET n (at address r_2)
$\Rightarrow s, v$	
Tail call and one return	
$s, r_1, bp_1, u_1, \dots, u_n, v_1, \dots, v_m$	TCALL m n f (at $r_2 - 1$)
$\Rightarrow s, r_1, bp_1, v_1, \dots, v_m$	code in the body of f
$\Rightarrow s, r_1, bp_1, w_1, \dots, w_k, v$	RET k
$\Rightarrow s, v$	

Figure 12.2: A tail call is equivalent to a call followed by return.

```

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

```

Note that the compiler will recognize a tail call only if it is immediately followed by a RET. Thus a tail call inside an if-statement (file `MicroC/ex15.c`), like this one:

```

void main(int n) {
    if (n!=0) {
        print n;
        main(n-1);
    } else
        print 999999;
}

```

is optimized to use the TCALL instruction only if the compiler never generates a GOTO to a RET, but directly generates a RET. Therefore the `makeJump` optimizations made by our continuation-based compiler are important also for efficient implementation of tail calls.

In general, it would be unsound to implement tail calls in C and micro-C by removing the calling function's stack frame and replacing it by the called function's stack frame. In C, an array allocated in a function `g` can be used also in a function `f` called by `g`, as in this program:

```

void g() {
    int a[10];
    a[1] = 117;
}

```

```

    f(1, a);
}

void f(int i, int a[]) {
    print a[i];
}

```

However, that would not work if `g`'s stack frame, which contains the `a` array, were removed and replaced by `f`'s stack frame. Most likely, the contents of array cell `a[1]` would be overwritten, and `f` would print some nonsense. The same problem appears if the calling function passes a pointer that point inside its stack frame to a called function. Note that in Java, in which no array is ever allocated on the stack, and pointers into the stack cannot be created, this problem does not appear. On the other hand, C# has similar problems as C in this respect.

To be on the safe side, a compiler should make the tail call optimization only if the calling function does not pass any pointers or array addresses to the function called by a tail call. The continuation-based compiler in `MicroC/Contcomp.fs` performs this unsound optimization anyway, to show what impact it has. See micro-C example `MicroC/ex21.c`.

12.3.7 Remaining deficiencies of the generated code

There are still some problems with the code generated for conditional statements. For instance, compilation of this statement (file `MicroC/ex16.c`):

```

if (n)
{ }
else
    print 1111;
print 2222;

```

generates this machine code:

```

L1: GETBP; LDI; IFZERO L3;
    GOTO L2;
L3: CSTI 1111; PRINTI; INCSP -1;
L2: CSTI 2222; PRINTI; RET 1

```

which could be optimized by inverting `IFZERO L3` to `IFNZRO L2` and deleting the `GOTO L2`. Similarly, the code generated for certain trivial while-loops is unsatisfactory. We might like the code generated for

```

void main(int n) {
    print 1111;
    while (false) {
        print 2222;
    }
    print 3333;
}

```

to consist only of the `print 1111` and `print 3333` statements, leaving out the `while`-loop completely, since its body will never be executed anyway. Currently, this is not ensured by the compiler. This is not a serious problem: some unreachable code is generated, but it does not slow down the program execution.

12.4 Other optimizations

There are many other kinds of optimizations that an optimizing compiler might perform, but that are not performed by our simple compiler:

- *Constant propagation*: if a variable `x` is set to a constant value, such as 17, and never modified, then every use of `x` can be replaced by the use of the constant 17. This may enable further optimizations if the variable is used in expressions such as

```
x * 3 + 1
```

- *Common subexpression elimination*: if the same (complex) expression is evaluated twice with the same values of all variables, then one could instead evaluate it once, store the result (in a variable or on the stack top), and reuse it. Common subexpressions frequently occur behind the scenes. For instance, the assignment

```
a[i] = a[i] + 1;
```

is compiled to

```

GETBP; aoffset; ADD; LDI; GETBP; ioffset; ADD; LDI; ADD;
GETBP; aoffset; ADD; LDI; GETBP; ioffset; ADD; LDI; ADD; LDI;
1; ADD; STI

```

where the address (lvalue) of the array indexing `a[i]` is evaluated twice. It might be better to compute it once, and store it in the stack. However,

the address to be reused is typically buried under some other stack elements, and our simple stack machine has no instruction to duplicate an element some way down the stack (similar to the JVM's `dup_x1` instruction).

- *Loop invariant computations*: If an expression inside a loop (`for`, `while`) does not depend on any variables modified by execution of the loop body, then the expression may be computed outside the loop (unless evaluation of the expression has a side effect, in which case it must be evaluated inside the loop). For instance, in

```
while (...) {
  a[i] = ...
}
```

part of the array indexing `a[i]` is loop invariant, namely the computation of the array base address:

```
GETBP; aoffset; ADD; LDI
```

so this could be computed once and for all before the loop.

- *Dead code elimination*: if the value of a variable or expression is never used, then the variable or expression may be removed (unless evaluation of the expression has side effects, in which case the expression must be preserved).

12.5 History and literature

The influential programming language textbook by Abelson, Sussman and Sussman [1] hints at the possibility of optimization on the fly in a continuation-based compiler. Xavier Leroy's 1990 report [81] describes optimizing backwards code generation for an abstract machine. This is essentially the machine and code generation technique used in Caml Light, OCaml, and Moscow ML. The same idea is used in Mads Tofte's 1990 Nsukka lecture notes [133], but the representation of the code continuation given there is more complicated and provides fewer opportunities for optimization.

The idea of generating code backwards is probably much older than any of these references.