

Exercise sheet 9 for 9 April 2003

2003-04-02

Do exercises 9.1, 9.2, 9.3, and 9.4. Hand in the solutions.

Exercise 9.1 The continuation-based micro-C compiler (file `imp/contcomp.sml`) still generates clumsy code in some cases. For instance, the statement (file `imp/ex16.c`):

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

is compiled to this machine code:

```
GETBP, LDI, IFZERO L2,
GOTO L1
L2: 1111, PRINTI, INCSP ~1,
L1: CSTI 2222, PRINTI
```

which could be optimized to this by inverting the conditional jump:

```
GETBP, LDI, IFNZRO L1,
L2: 1111, PRINTI, INCSP ~1,
L1: CSTI 2222, PRINTI
```

Improve the compiler to recognize this situation. It must recognize that it is about to generate code of these forms:

```
IFZERO L2, GOTO L1, L2: ....
IFNZRO L2, GOTO L1, L2: ....
```

where a conditional jump just skips over an unconditional jump. Instead it should generate code such as this:

```
IFNZRO L1, L2: ....
IFZERO L1, L2: ....
```

Exercise 9.2 Improve code generation in the continuation-based micro-C compiler so that a less-than comparison with constant arguments is compiled to its truth value. For instance, `11 < 22` should compile to the same code as `true`, and `22 < 11` should compile to the same code as `false`. This can be done by a small extension of the `addCST` function.

Further improve the code generation so that all comparisons with constant arguments are compiled to the same code as `true` (e.g. `11 <= 22` and `11 != 22` and `22 > 11` and `22 >= 11`) or `false`.

Check that `if (11 <= 22) print 33;` compiles to code that unconditionally executes `print 33` without performing any test or jump at all.

Exercise 9.3 Extend the micro-C abstract syntax (file `imp/Absyn.sml`) with conditional expressions `Cond(e1, e2, e3)`, corresponding to the C/C++/Java/C# concrete syntax:

```
e1 ? e2 : e3
```

The expression `Cond(e1, e2, e3)` must evaluate `e1`, and if the result is non-zero, must evaluate `e2`, otherwise `e3`. (If you want to extend also the lexer and parser to accept this new syntax, then note that `?` and `:` are right associative. But implementing them in the lexer and parser is not strictly necessary for this exercise).

Schematically, the conditional expression should be compiled to the code shown below:

```

    <e1code>
    IFZERO L1
    <e2code>
    GOTO L2
L1: <e3code>
L2:

```

Extend the continuation-based micro-C compiler (file `imp/contcomp.sml`) to compile conditional expressions to stack machine code. Your compiler should optimize code while generating it. Check that your compiler compiles the following kinds of examples to code that works properly:

```
true ? 1111 : 2222
```

```
false ? 1111 : 2222
```

The abstract syntax for the first expression is `Cond(Cst(CstI 1), Cst(CstI 1111), Cst(CstI 2222))`. Unless you have implemented conditional expressions (`e1 ? e2 : e3`) in the lexer and parser, the simplest way to experiment with this is to invoke the `cExpr` expression compilation function directly:

```
cExpr (Cond(Cst(CstI 1), Cst(CstI 1111), Cst(CstI 2222))) (Env.empty, 0) [];
```

Do not waste too much effort trying to get your compiler to optimize away everything that is not needed. This seems impossible without traversing and modifying already generated code.

Exercise 9.4 As in the lecture, note that

```

e1 && e2  is equivalent to  (e1 ? e2 : 0)
e1 || e2  is equivalent to  (e1 ? 1 : e2)

```

Implement the sequential logical operators (`&&` and `||`) this way in your extended compiler from Exercise 9.3. Test this approach on file `imp/ex13.c` and possibly other examples. How does the code quality compare to the complicated compilation of `&&` and `||`?

Exercise 9.5 Improve the compilation of assignment expressions that are really just increment operations, such as these

```

i = i + 1
a[i] = a[i] + 1

```

It is easiest to recognize such cases in the abstract syntax, not by looking at the code continuation.

Exercise 9.6 Try to make sense of the code generated by the continuation-based compiler for the n -queens program in file `imp/ex11.c`. Draw a flowchart of the compiled program.

Exercise 9.7 Implement a post-optimizer for stack machine symbolic code as generated by the micro-C compilers. This should be a function:

```
optimize : instr list -> instr list
```

where `instr` is defined in `imp/Machine.sml`. The idea is that `optimize` should improve the code using the local bytecode equivalences shown in the lecture. Also, it may delete code that is unreachable (code that cannot be executed). Function `optimize` should be *correct*: the code it produces must behave the same as the original code, when executed on the stack machine in `imp/Machine.java`.

The function would have to make two passes over the code. In the *first pass* it computes a set of all reachable labels. A label, and the instructions following it, is reachable if (1) it can be reached from the beginning of the code (instruction 0), or (2) there is a jump or call (`GOTO`, `IFZERO`, `IFNZRO`, `CALL`, `TCALL`) to it from a reachable instruction.

In the *second pass* it can go through the instruction sequences at reachable labels (only), and simplify them using the bytecode equivalences.

Note that simplification of `[CSTI 1, IFZERO L1]` may cause label `L1` to become unreachable. Also, deletion of the code labelled `L1` in `[CST 0, GOTO L2, Label L1, ..., Label L2, ADD]` may enable further local simplifications. Hence the computation of reachable labels and simplification of code may have to be repeated until no more simplifications can be made.