Lesson 4

Solutions

Compiler Design I (Komplatorönteknik I) 2018

DISCLAIMER: In the code listings of the following exercises we use a syntax that closely resembles C. However the C language does not allow by-reference parameter passing (which some parts of Exercise 1 assume).

1 Parameter passing

Suppose we have the following program:

```c
int foo(int a) {
    int b = a++;
    return b * a
}

int bar(int b) {
    return foo(b);
}

int main() {
    int c;
    c = 6;
    return bar(c);
}
```

Assume that the compiler allocates all the variables in the stack.

The single arguments of foo and bar can be passed either by-value or by-reference. For each of the four possible combinations (i.e. both arguments by-value, foo’s argument by-value & bar’s argument by-reference, etc.) describe:

1. What kind of data should the compiler put in the argument slots of the activation records for the calls on lines 7 and 13?
2. What kind of assembly code will be necessary to retrieve the value of variable a on line 2?
Answer

The aim of this exercise is to show how compilers handle calling-by-value and calling-by-reference.

First of all it is important to understand that the code for the called function is generated once and should be able to retrieve the value of the passed argument regardless of what has happened before. Other calls to each function might also be present and should work with the same code.

Therefore, if the argument is passed by value, it is always the value that should be in the activation record. Similarly, if the argument is passed by reference, its value should be retrievable with a single “dereference”.

With the previous in mind, to retrieve the value (question 1.2):

- If foo uses call-by-value:
  \[\text{lw}\ \$a0,\ \text{OFFSET}(\$FP)\]

- If foo uses call-by-reference:
  \[\text{lw}\ \$t0,\ \text{OFFSET}(\$FP)\]
  \[\text{lw}\ \$a0,\ 0(\$t0)\]

(Here OFFSET is the specific offset for the slot of the argument.)

With these in mind, for question 1.1, the contents of the argument slots of the activation records will be:

- foo uses call-by-value & bar uses call-by-value:
  Both slots will contain actual values (i.e. the numeric representation of 6).

- foo uses call-by-value & bar uses call-by-reference:
  The slot for the call on line 13 contains the address of the variable c, whereas the slot for the call on line 7 contains the actual value retrieved from that address before the call (i.e 6).

- foo uses call-by-reference & bar uses call-by-value:
  The slot for the call on line 13 contains the value of the variable c at the time of the call (i.e. 6), whereas the slot for the call on line 7 contains the address of the previous slot.

- foo uses call-by-reference & bar uses call-by-reference:
  The slot for the call on line 13 contains the address of the variable c, and the same address is copied in the slot for the call on line 7. The compiler knows that the function received an argument by reference and has to pass it on by reference, therefore it copies the original reference.

1If we allow the first argument to be passed directly in a register (e.g. $a0), then the value or reference is what is stored in the register. The code in this case is:

- If foo uses call-by-value:
  No code needed, the value is already in the register.

- If foo uses call-by-reference:
  \[\text{lw}\ \$a0,\ 0(\$a0)\]
2 Code generation

Suppose that we want to generate code for the expression:

\[
\text{cond} \quad \begin{cases} 
\langle p_1 \rangle &\Rightarrow \langle e_1 \rangle; \\
\langle p_2 \rangle &\Rightarrow \langle e_2 \rangle; \\
\vdots \\
\langle p_n \rangle &\Rightarrow \langle e_n \rangle; \\
1 &\Rightarrow \langle e_{n+1} \rangle 
\end{cases}
\]

dnoc

The evaluation of a \text{cond} expression begins with the evaluation of the predicate \langle p_1 \rangle (if it exists, \( n \) can also be zero). If \langle p_1 \rangle evaluates to a non-zero value, then \langle e_1 \rangle is evaluated, and the evaluation of the \text{cond} expression is complete. If \langle p_1 \rangle evaluates to zero, then \langle p_2 \rangle is evaluated, and this process is repeated until one of the predicates evaluates to a non-zero value. The value of the \text{cond} expression is the value of the expression \langle e_i \rangle corresponding to the first predicate \langle p_i \rangle that evaluates to a non-zero value. If all the predicates evaluate to zero, then the value of the \text{cond} expression is \langle e_{n+1} \rangle.

Write a code generation function: \( \text{cgen} \) (\text{cond} \langle p_1 \rangle \Rightarrow \langle e_1 \rangle; \ldots; \langle p_n \rangle \Rightarrow \langle e_n \rangle; 1 \Rightarrow \langle e_{n+1} \rangle\) \text{dnoc}) for this conditional expression.

Answers

\[
\text{cgen} \left( \text{cond} \langle p_1 \rangle \Rightarrow \langle e_1 \rangle; \langle p_{i+1} \rangle \Rightarrow \langle e_{i+1} \rangle; \ldots \langle p_2 \rangle \Rightarrow \langle e_2 \rangle; \langle p_2 \rangle \Rightarrow \langle e_2 \rangle \right) \\
\]

\[
\text{cgen}(\langle p_1 \rangle); /\text{ assumes that the result is stored in a0} \\
\text{emit}(\text{"li }$t0 \text{ 0"}); \\
\text{emit}(\text{"beq }$a0 \%t0 \text{ next"}); \\
\text{cgen}(\langle e_1 \rangle); \\
\text{emit}(\text{"b exiti"}); \\
\text{emit}(\text{"nexti:"}); \\
\text{cgen}(\langle p_{i+1} \rangle \Rightarrow \langle e_{i+1} \rangle; \ldots \langle p_n \rangle \Rightarrow \langle e_n \rangle; 1 \Rightarrow \langle e_{n+1} \rangle); \\
\text{emit}(\text{"exiti:"}); \\
\}
\]
3 Local optimizations

Consider the following basic block, in which all variables are integers and ** denotes exponentiation:

\[
\begin{align*}
a &:= b + c \\
z &:= a ** 2 \\
x &:= 0 * b \\
y &:= b + c \\
w &:= y * y \\
u &:= x + 3 \\
v &:= u + w
\end{align*}
\]

Assume that the only variables that are live at the exit of this block are \( v \) and \( z \). In order, apply the following optimizations to this basic block. Show the result of each transformation.

1. algebraic simplification
2. common sub-expression elimination
3. copy propagation
4. constant folding
5. dead code elimination

When you have completed part 5, the resulting program will still not be optimal. What optimizations, in what order, can you apply to optimize the result of 5 further?

Answer

1) Algebraic optimization: 2) Common sub-expression elimination: 3) Copy propagation:

\[
\begin{align*}
a &:= b + c \\
z &:= a * a \\
x &:= 0 \\
y &:= b + c \\
w &:= y * y \\
u &:= x + 3 \\
v &:= u + w
\end{align*}
\]

4) Constant folding: 5) Dead code elimination:

\[
\begin{align*}
a &:= b + c \\
z &:= a * a \\
x &:= 0 \\
y &:= a \\
w &:= a * a \\
u &:= 3 \\
v &:= u + w
\end{align*}
\]

Doing common sub-expression elimination, copy propagation, constant propagation and dead code elimination one more time, we get the final minimal form:
4 Register allocation

Consider the following program.

\begin{verbatim}
L0: e := 0
    b := 1
    d := 2
L1: a := b + 2
    c := d + 5
    e := e + c
    f := a * a
    if f < c goto L3
L2: e := e + f
    goto L4
L3: e := e + 2
L4: d := d + 4
    b := b - 4
    if b != d goto L1
L5:
\end{verbatim}

This program uses six temporaries, a-f. Assume that the only variable that is live on exit from this program is e. Draw the register interference graph. (Drawing a control-flow graph and computing the sets of live variables at every program point may be helpful.)

\textbf{Answers} First we draw the control flow graph:

The register interference graph is therefore: