Chapter 11

Analysis and optimisation

In chapter 8, we briefly mentioned optimisations without going into detail about how they are done. We will remedy this in this chapter.

An optimisation, generally, is about recognising instructions that form a specific pattern that can be replaced by a smaller or faster pattern of new instructions. In the simplest case, the pattern is just a short sequence of instructions that can be replaced by a another short sequence of instructions. In chapter 8, we replaced sequences of intermediate-code instructions by sequences of machine-code instructions, but the same idea can be applied to replacing sequences of machine-code instructions by sequences of machine-code instructions or sequences of intermediate-code instructions by other sequences of intermediate-code instructions. This kind of optimisation is called *peephole optimisation*, because we look at the code through a small hole that just allows us to see short sequences of instructions.

Another thing to note about the patterns we used in chapter 8 is that they sometimes required some of the variables involved to have no subsequent uses. This is a non-local property that requires looking at an arbitrarily large context of the instructions, sometimes the entire procedure or function in which the instructions appear. A variable having possible subsequent uses is called *live*. In chapter 9 we looked at how *liveness analysis* could determine which variables are live at each instruction. We used this to determine interference between variables, but the same information can also be used to enable optimisations, such as replacing a sequence of instructions by a simpler sequence.

Many optimisations are like this: We have a pattern of instructions and some requirements about the context in which the pattern appears. These contextual requirements are often found by analyses similar to liveness analysis, collectively called *data-flow analyses*.

We will now look at optimisations that can be enabled by such analyses. We will later present a few other types of optimisations.

11.1 Data-flow analysis

As the name indicates, data-flow analysis attempts to discover how information flows through a program. We already discussed liveness analysis, where the information that a variable is live flows through the program in the opposite order of the flow of values: A value flows from an assignment to a variable to its uses, but liveness information flows from a use of a variable back to its assignments. Liveness analysis is, hence, called a backwards analysis. In other analyses information flows in the same order as values. For example, an analysis might try to approximate the set of possible values that a variable can hold, and here the flow is naturally from assignments to uses.

The liveness analysis presented in chapter 9 consisted of four things:

1. Information about which instructions can follow others, *i.e.*, the successors of each instruction.
2. For each instruction *gen* and *kill* sets that describe how data-flow information is created and destroyed by the instruction.
3. Equations that define *in* and *out* sets by describing how data-flow information flow between instructions.
4. Initialisation of the *in* and *out* sets for a fixed-point iteration that solve the data-flow equations.

We will use the same template for other data-flow analyses, but the details might differ. For example:

1. Forwards analyses require information about the predecessors of an instruction instead of its successors.
2. Where liveness analysis uses sets of variables, other analyses might use sets of instructions or sets of variable/value pairs.
3. The equations for *in* and *out* sets might differ. For example, they may use intersection instead of union to combine information from several successors or predecessors of an instruction.
4. Where liveness analysis initialises all *in* and *out* sets to empty sets (except for the *out* set of the last instruction in the function, which is initialised to the set of variables live at the exit of the function), other analyses might initialise the sets to, for example, the set of all instructions in the function. If we want the minimal solution to the equations, we initialise with empty sets, but if the want the maximal solution to the equations, we initialise with the set of all

relevant values. See appendix A for more details about minimal and maximal solutions to set equations.

We will in the following sections show some examples of optimisations and the data-flow analyses required to find the contextual information required to determine if the optimisation is applicable. As the optimisations we show are not specific to any particular processor, we will show them in terms of intermediate code, but the same principles can be applied to analysis and optimisations of machine-code.

11.2 Common subexpression elimination

After translation to intermediate code, there might be several occurrences of the same calculations, even when this is not the case in the source program. For example, the assignment `a[i] := a[i]+1` can in many languages not be simplified at the source level, but the assignment might be translated to the following intermediate-code sequence:

```
t1 := 4*i
t2 := a+t1
t3 := M[t2]
t4 := t3+1
t5 := 4*i
t6 := a+t5
M[t6] := t4
```

Note that both the multiplication by 4 and the addition of `a` is repeated, but that while we can see that the expressions `a+t1` and `a+t5` have the same value, they are not textually identical.

Our ultimate goal is to eliminate both of these redundancies, but we will start by a simpler analysis that only catches the second occurrence of `4*i` and then discuss how it can be extended to also eliminate `a+t5`.

11.2.1 Available assignments

If we want to replace an expression by a variable that holds the value of the expression, we need to keep track of which expressions we have available and which variables hold their values. So we want at each program point a set of pairs of variables and expressions. Since each such pair originates in an assignment of the expression to the variable, we can, equivalently, use a set of assignment instructions that occur in the program. This makes things slightly simpler later on. We call this analysis *available assignments*

It is clear that information flows from assignment forwards, so for each instruction in the program, the *in* set is the set of available assignments before the instruction is executed and the *out* set is the set of assignments available afterwards. The *gen* and *kill* sets should, hence, describe which new assignments become available and which are no longer available. An assignment makes itself available, *unless* the variable on the left-hand side also occurs on the right-hand side (because the assignment would make a new occurrence of the expression have a different value). All other assignments where the variable on the left-hand side of the instruction occurs (on either side) are invalidated: If the variable occurs on the left-hand side on another assignment, the variable no longer holds the value of the expression on the right-hand side, and if the variable occurs in an expression, the expression changes value, so the left-hand-side variable no longer holds the current value of the expression. Figure 11.1 shows the *gen* and *kill* sets for each kind of instruction in the intermediate language.

Note that a copy instruction $x := y$ does not generate any available assignment, as nothing is gained by replacing an occurrence of y by x. Note, also, that any store of a value to memory kills all instructions that load from memory. We need to make this rather conservative assumption because we do not know where in memory loads and stores go.

The next step is to define the equations for *in* and *out* sets:

$$out[i] = gen[i] \cup (in[i] \setminus kill[i]) \tag{11.1}$$

$$in[i] = \bigcap_{j \in pred[i]} out[j] \tag{11.2}$$

As mentioned above, the assignments that are available after an instruction (*i.e.*, in the *out* set) are those that are generated by the instruction and those that were available before, except for those that are killed by the instruction. The available assignments before an instruction (*i.e.*, in the *in* set) are those that are available at all predecessors, so we take the intersection of the sets available at the predecessors.

The predecessors $pred[i]$ of instruction i is $\{i-1\}$, except when i is a `LABEL` instruction, in which case we also add all `GOTO` and `IF` instructions that can jump to i, or when i is the first instruction in the program (*i.e.*, when $i = 1$), in which case $i-1$ is not in the predecessors of i.

We need to initialise the *in* and *out* sets for the fixed-point iteration. We want to find the maximal solution to the equations: Consider a loop where an assignment is available before the loop and no variable in the assignment is changed inside the loop. We would want the assignment to be available inside the loop, but if we initialise the *out* set of the jump from the end of the loop to its beginning to the empty set, the intersection of the assignments available before the loop and those available at the end of the loop will be empty, and remain that way throughout the

Instruction i	$gen[i]$	$kill[i]$
`LABEL` l	$\emptyset$	$\emptyset$
$x := y$	$\emptyset$	$assg(x)$
$x := k$	$\{x := k\}$	$assg(x)$
$x :=$ **unop** y where $x \neq y$	$\{x :=$ **unop** $y\}$	$assg(x)$
$x :=$ **unop** x	$\emptyset$	$assg(x)$
$x :=$ **unop** k	$\{x :=$ **unop** $k\}$	$assg(x)$
$x := y$ **binop** z where $x \neq y$ and $x \neq z$	$\{x := y$ **binop** $z\}$	$assg(x)$
$x := y$ **binop** z where $x = y$ or $x = z$	$\emptyset$	$assg(x)$
$x := y$ **binop** k where $x \neq y$	$\{x := y$ **binop** $k\}$	$assg(x)$
$x := x$ **binop** k	$\emptyset$	$assg(x)$
$x := M[y]$ where $x \neq y$	$\{x := M[y]\}$	$assg(x)$
$x := M[x]$	$\emptyset$	$assg(x)$
$x := M[k]$	$\{x := M[k]\}$	$assg(x)$
$M[x]$:= y	$\emptyset$	$loads$
$M[k]$:= y	$\emptyset$	$loads$
`GOTO` l	$\emptyset$	$\emptyset$
`IF` x **relop** y `THEN` l_t `ELSE` l_f	$\emptyset$	$\emptyset$
$x :=$ `CALL` $f(args)$	$\emptyset$	$assg(x)$

where $assg(x)$ is the set of all assignments in which x occurs on either left-hand or right-hand side, and $loads$ is the set of all assignments of the form $x := M[\cdot]$.

Figure 11.1: Gen and kill sets for available assignments

```
1:  i := 0
2:  a := n * 3
3:  IF i < a THEN loop ELSE end
4:  LABEL loop
5:  b := i * 4
6:  c := p + b
7:  d := M[c]
8:  e := d * 2
9:  f := i * 4
10: g := p + f
11: M[g] := e
12: i := i + 1
13: a := n * 3
14: IF i < a THEN loop ELSE end
15: LABEL end
```

Figure 11.2: Example program for available-assignments analysis

iteration. So, instead we initialise the *in* and *out* sets for all instructions except the first to the set of all assignments in the program (so we find the maximal solution). The *in* set for the first instruction remains empty, as no assignments is available at the beginning of the program.

When an analysis takes the intersection of the values for the predecessors, we will always initialise sets (except for the first instruction) to the largest possible, so we do not get overly conservative results for loops.

11.2.2 Example of available-assignments analysis

Figure 11.2 shows a program that doubles elements of an array p

Figure 11.3 shows *pred*, *gen* and *kill* sets for each instruction in this program. We represent an assignment by the number of the assignment instruction, so *gen* and *kill* sets are sets of numbers. We will, however, identify identical assignments with the same number, so both the assignment in instruction 2 and the assignment in instruction 13 are identified with the number 2, as can be seen in the *gen* set of instruction 13.

Note that each assignment kills itself, but since it also (in most cases) generates itself, the net effect is to remove all conflicting assignments (including itself) and then adding itself. Assignment 12 ($i := i + 1$) does not generate itself, since i also occurs on the right-hand side. Note, also, that the write to memory in instruction 11 kills instruction 7, as this loads from memory.

i	$pred[i]$	$gen[i]$	$kill[i]$
1		1	1,5,9,12
2	1	2	2
3	2		
4	3,14		
5	4	5	5,6
6	5	6	6,7
7	6	7	7,8
8	7	8	8
9	8	9	9,10
10	9	10	10
11	10		7
12	11		1,5,9,12
13	12	2	2
14	13		
15	3,14		

Figure 11.3: *pred*, *gen* and *kill* for the program in figure 11.2

For the fixed-point iteration we initialise the *in* set of instruction 1 to the empty set and all other *in* and *out* sets to the set of all assignments. We need only the assignments that are actually generated by some instructions, *i.e.*, $\{1,2,5,6,7,8,9,10\}$. We then iterate equations 11.2 and 11.1 as assignments until we reach a fixed-point. Since information flow is forwards, we process the instructions by increasing number and calculate $in[i]$ before $out[i]$. The iteration is shown in figure 11.4. For space reasons, the table is shown sideways and the final iteration (which is identical to iteration 2) is not shown.

11.2.3 Using available assignment analysis for common subexpression elimination

If instruction i is of the form $x := e$ for some expression e and $in[i]$ contains an assignment $y := e$, then we can replace $x := e$ by $x := y$.

If we apply this idea to the program in figure 11.2, we see that at instruction 9 ($f := i*4$), we have assignment 5 ($b := i*4$) available, so we can replace instruction 9 by ($f := b$). At instruction 13 ($a := n*3$), we have assignment 2 ($a := n*3$) available, so we can replace instruction 13 by $a := a$, which we can omit entirely as it is a no-operation. The optimised program is shown in figure 11.5.
Note that while we could eliminate identical expressions, we could not eliminate

i	Initialisation		Iteration 1		Iteration 2	
	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$
1		1,2,5,6,7,8,9,10		1		1
2	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1	1,2	1	1,2
3	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2	1,2	1,2	1,2
4	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2	1,2	2	2
5	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2	1,2,5	2	2,5
6	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5	1,2,5,6	2,5	2,5,6
7	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6	1,2,5,6,7	2,5,6	2,5,6,7
8	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7	1,2,5,6,7,8	2,5,6,7	2,5,6,7,8
9	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7,8	1,2,5,6,7,8,9	2,5,6,7,8	2,5,6,7,8,9
10	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9	1,2,5,6,7,8,9,10	2,5,6,7,8,9	2,5,6,7,8,9,10
11	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,8,9,10	2,5,6,7,8,9,10	2,5,6,8,9,10
12	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,8,9,10	2,6,8,10	2,5,6,8,9,10	2,6,8,10
13	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	2,6,8,10	2,6,8,10	2,6,8,10	2,6,8,10
14	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	2,6,8,10	2,6,8,10	2,6,8,10	2,6,8,10
15	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	2	2	2	2

Figure 11.4: Fixed-point iteration for available-assignment analysis

```
 1: i := 0
 2: a := n * 3
 3: IF i < a THEN loop ELSE end
 4: LABEL loop
 5: b := i * 4
 6: c := p + b
 7: d := M[c]
 8: e := d * 2
 9: f := b
10: g := p + f
11: M[g] := e
12: i := i + 1
14: IF i < a THEN loop ELSE end
15: LABEL end
```

Figure 11.5: The program in figure 11.2 after common subexpression elimination.

the expression $p+f$ in instruction 10 even though it has the same value as the right-hand side of the available assignment 6 ($c := p+b$), since $b = f$. A way of achieving this is to first replace all uses of f by b (which we can do since the only assignment to f is $f := b$) and then repeat common subexpression elimination on the resulting program. If a large expression has two occurrences, we might have to repeat this a large number of times to get the optimal result. An alternative is to keep track of sets of variables that have the same value (a technique called *value numbering*), which allows large common subexpressions to be eliminated in one pass.

Another limitation of the available assignment analysis is when two different predecessors to an instruction have the same expression available but in different variables, *e.g.*, if one predecessor of instruction i has the available assignments $\{x := a+b\}$ and the other predecessor has the available assignments $\{y := a+b\}$. These have an empty intersection, so the analysis would have no available assignments at the entry of instruction i. It is possible to make common subexpression elimination make the expression $a+b$ available in this situation, but this makes analysis and transformation more complex.

Suggested exercises: 11.1.

11.3 Jump-to-jump elimination

When we have an instruction sequence like

```
LABEL l1
GOTO l2
```

we would like to replace all jumps to l_1 by jumps to l_2. However, there might be chains of such jumps, *e.g.*,

```
LABEL l1
GOTO l2
...
LABEL l2
GOTO l3
...
LABEL l3
GOTO l4
```

We want, in this example, to replace a jump to l_1 with a jump to l_4 directly. To do this, we make a data-flow analysis that for each jump finds its ultimate destination. The analysis is a backwards analysis, as information flows from a label back to the instruction that jumps to it. The *gen* and *kill* sets are quite simple:

instruction	*gen*	*kill*
`LABEL` l	$\{l\}$	$\emptyset$
`GOTO` l	$\emptyset$	$\emptyset$
`IF` c `THEN` l_1 `ELSE` l_2	$\emptyset$	$\emptyset$
any other	$\emptyset$	the set of all labels

The equations for *in* and *out* are:

$$in[i] = \begin{cases} gen[i] \setminus kill[i] & \text{if } out[i] \text{ is empty} \\ out[i] \setminus kill[i] & \text{if } out[i] \text{ is non-empty} \end{cases} \tag{11.3}$$

$$out[i] = \bigcap_{j \in succ[i]} in[j] \tag{11.4}$$

We initialise all *out* sets to the set of all labels, except the *out* set of the last instruction (which has no successors), which is initialised to empty.

Note that, after the fixed-point iteration, $in[i]$ can only have more than one element if i is part of or jumps to a loop consisting entirely of `GOTO` instructions and labels. Such infinite loops can occur in valid programs (*e.g.*, as an idle-loop that waits for an interrupt), so we allow this.

When we have reached a fixed-point, we can do the following optimisations:

- At `GOTO` i, we can replace i by any element in $in[i]$.

- If instruction i is `LABEL` l and $l \notin in[i]$, there will be no jumps to l after the optimisation, so we can remove instruction i and all following instructions up to just before the next label. This is called *dead code elimination*.

- At `IF` c `THEN` i `ELSE` j, we can replace i by any element in $in[i]$ and j by any element in $in[j]$. If $in[i] = in[j]$, the test is redundant, and we can replace `IF` c `THEN` i `ELSE` j by `GOTO` l, where l is any element in $in[i]$.

Suggested exercises: 11.2.

11.4 Index-check elimination

When a language requires bounds-checking of array accesses, the compiler must insert tests before each array access to verify that the index is in range. Index-check elimination aims to remove these checks when they are redundant.

To find this, we collect for each program point a set of inequalities that hold at this point. If a bounds check is implied by these inequalities, we can eliminate it.

We will use the conditions in `IF-THEN-ELSE` instructions as our source for inequalities. Note that, since bounds checks are translated into such instructions, this set of conditions includes all bounds checks.

Conditions all have the form x **relop** p, where x is a variable and p is either a constant or a variable. We only care about inequalities, *i.e.*, conditions of the form $p < q$ or $p \leq q$, where p and q are either variables or constants (but they can not both be constants). Each condition that occurs in the program is translated into a set of inequalities of this form. For example $x = y$ is translated into $x \leq y$ and $y \leq x$. We also generate inequalities for the negation of each condition in the program, so a condition like $x < 10$ generates the inequalities $x < 10$ and $10 \leq x$. A condition $x = y$ generates no inequalities for its negation, as $x \neq y$ can not be expressed as a conjunction of inequalities. This gives us a universe Q of inequalities that we work on. At each point in the program, we want to find which of these inequalities are guaranteed to hold at this point, *i.e.*, a subset of Q.

The idea is that, after executing the instruction `IF` c `THEN` l_1 `ELSE` l_2, the conditions derived from c will be true at l_1 and the conditions derived from the negation of c will be true at l_2, *assuming* there are no other jumps to l_1 or l_2. To ensure this assumption, we insert extra labels and jumps in the program: each instruction of the form `IF` c `THEN` l_1 `ELSE` l_2 is replaced by the sequence

$$in[i] = \begin{cases} \bigcap_{j \in pred[i]} in[j] \\ \qquad \text{if } pred[i] \text{ has more than one element} \\ in[pred[i]] \cup when(c) \\ \qquad \text{if } pred[i] \text{ is } \texttt{IF}\ c\ \texttt{THEN}\ i\ \texttt{ELSE}\ j \\ in[pred[i]] \cup whennot(c) \\ \qquad \text{if } pred[i] \text{ is } \texttt{IF}\ c\ \texttt{THEN}\ j\ \texttt{ELSE}\ i \\ (in[pred[i]] \setminus conds(Q,x)) \cup equal(Q,x,p) \\ \qquad \text{if } pred[i] \text{ is of the form } x := p \\ in[pred[i]] \setminus upper(Q,x) \\ \qquad \text{if } pred[i] \text{ is of the form } x := x + k \text{ where } k \geq 0 \\ in[pred[i]] \setminus lower(Q,x) \\ \qquad \text{if } pred[i] \text{ is of the form } x := x - k \text{ where } k \geq 0 \\ in[pred[i]] \setminus conds(Q,x) \\ \qquad \text{if } pred[i] \text{ is of a form } x := e \text{ not covered above} \\ in[pred[i]] \\ \qquad otherwise \end{cases}$$

Figure 11.6: Equations for index-check elimination

```
IF c THEN t ELSE f
LABEL t
GOTO l1
LABEL f
GOTO l2
```

where t and f are new labels that do not occur anywhere else.

This way, if a label has more than one predecessor, these will all be `GOTO` statements (that do not alter the set of valid inequalities). We can later remove the added code by jump-to-jump elimination as described in section 11.3.

We do not use *out* sets, but write the equations for *in* sets in terms of the *in* sets of the predecessors and the type of instruction of the predecessor, as shown in figure 11.6. Note that information flows forwards: from predecessors to successors. We use the following auxiliary definitions:

- $when(c)$ is the set of inequalities implied by the condition c. These will be elements of Q (the universe of inequalities), since Q was constructed to include all these.

- $whennot(c)$ is the set of inequalities implied by the negation of the condition c. These will, likewise, be elements of Q.

- $conds(Q,x)$ is the set of inequalities from Q that involve x.
- $equal(Q,x,p)$, where p is a variable or a constant, is the set of inequalities from Q that are implied by the equality $x = p$. For example, if $Q = \{x < 10, 10 \leq x, 0 < x, x \leq 0\}$ then $equal(Q,x,7) = \{x < 10, 0 < x\}$.
- $upper(Q,x)$ is the set of inequalities from Q that have the form $p < x$ or $p \leq x$, where p is a variable or a constant.
- $lower(Q,x)$ is the set of inequalities from Q have the form $x < p$ or $x \leq p$, where p is a variable or a constant.

Normally, any assignment to a variable invalidates all inequalities involving that variable, but we have made some exceptions: If we assign a constant or variable to a variable, we check all the possible inequalities and add those that are implied by the assignment. Also, if x increases, we invalidate all inequalities that bound x from above but keep those that bound x from below, and if x decreases, we invalidate the inequalities that bound x from below but keep those that bound x from above. We can add more special cases to make the analysis more precise, but the above are sufficient for the most common cases.

We initialise all *in* sets to Q, except the *in* set for the first instruction, which is initialised to the empty set.

After the data-flow analysis reaches a fixed-point, the inequalities in $in[i]$ are guaranteed to hold at instruction i. So, if we have an instruction i of the form `IF` c `THEN` l_t `ELSE` l_f and c is implied by the inequalities in $in[i]$, we can replace the instruction by `GOTO` l_t. If the negation of c is implied by the inequalities in $in[i]$, we can replace the instruction by `GOTO` l_f.

We illustrate the analysis by an example. Consider the following for-loop and assume that the array `a` is declared to go from 0 to 10.

```
for i:=0 to 9 do
  a[i] := 0;
```

This loop can be translated (with index check) into the intermediate code shown in figure 11.7.

The set Q of possible inequalities in the program are derived from the conditions in the three `IF-THEN-ELSE` instructions and their negations, *i.e.*, $Q = \{i \leq 9, 9 < i, i < 0, 0 \leq i, 10 < i, i \leq 10\}$.

We leave the fixed-point iteration and check elimination as an exercise to the reader, but note that the assignment $i := 0$ in instruction 1 implies the inequalities $\{i \leq 9, 0 \leq i, i \leq 10\}$ and that the assignment $i := i + 1$ in instruction 11 preserves $0 \leq i$ but invalidates $i \leq 9$ and $i \leq 10$.

```
 1: i := 0
 2: LABEL for1
 3: IF i ≤ 9 THEN for2 ELSE for3
 4: LABEL for2
 5: IF i < 0 THEN error ELSE ok1
 6: LABEL ok1
 7: IF i > 10 THEN error ELSE ok2
 8: LABEL ok2
 9: t := i * 4
10: t := a + t
11: M[t] := 0
12: i := i + 1
13: GOTO for1
14: LABEL for3
```

Figure 11.7: Intermediate code for for-loop with index check

Suggested exercises: 11.3.

11.5 Limitations of data-flow analyses

All of the data-flow analyses we have seen above are approximations: They will not always accurately reflect what happens at runtime: The index-check analysis may fail to remove a redundant index check, and the available assignment analysis may say an assignment is unavailable when, in fact, it is available.

In all cases, the approximations err on the safe side: It is better to miss an opportunity for optimisation than to make an incorrect optimisation. For liveness analysis, this means that if you are in doubt about a variable being live, you had better say that it is, as assuming it dead might cause its value to be overwritten. When available assignment analysis is used for common subexpression elimination, saying that an assigning is available when it is not may make the optimisation replace an expression by a variable that does not always hold the same value as the expression, so it is better to leave an assignment out of the set if you are in doubt.

It can be shown that no compile-time analysis that seeks to uncover nontrivial information about the run-time behaviour of programs can ever be completely exact. You can make more and more complex analyses that get closer and closer to the exact result, but there will always be programs where the analysis is not precise. So a compiler writer will have to be satisfied with analyses that find most cases where an optimisation can be applied, but misses some.

11.6 Loop optimisations

Since many programs spend most of their time in loops, it is worthwhile to study optimisations specific for loops.

11.6.1 Code hoisting

One such optimisation is recognising computations that are repeated in every iteration of the loop without changing the values involved, *i.e.,* loop-invariant computations. We want to lift such computations outside the loop, so they are performed only once. This is called *code hoisting*.

We saw an example of this in section 11.2.3, where calculation of $n*3$ was done once before the loop and subsequent re-computations were replaced by a reference to the variable a that holds the value of $n*3$ computed before the loop. However, it is only because there was an explicit computation of $n*3$ before the loop, that we could avoid re-computation inside the loop: Otherwise, the occurrence of $n*3$ inside the loop would not have any available assignment that can replace the calculation.

So our aim is to move or copy loop-invariant assignments to before the loop, so their result can be reused inside the loop. Moving a computation to before the loop may, however, cause it to be computed even when the loop is not entered. In addition to causing unnecessary computation (which goes against the wish for optimisation), such computations can cause errors when the precondition (the loop condition) is not satisfied. For example, if the invariant computation is a memory access, the address may be valid only if the loop is entered.

A common solution to this problem is to unroll the loop once: A loop of the form (using C-like syntax):

```
while (cond) {
  body
}
```

is transformed to

```
if (cond) then {
  body
  while (cond) {
    body
  }
}
```

Similarly, a test-at-bottom loop of the form

```
do
  body
while (cond)
```

can be unrolled to

```
body
while (cond) {
  body
}
```

Now, we can safely calculate the invariant parts in the first copy of the body and reuse the results in the loop. If the compiler does common subexpression elimination, this unrolling is all that is required to do code hoisting – assuming the unrolling is done before common-subexpression elimination. Unrolling of loops is most easily done at source-code level (*i.e.*, on the abstract syntax tree), so this is no problem. This unrolling will, of course, increase the size of the compiled program, so it should be done with care if the loop body is large.

11.6.2 Memory prefetching

If a loop goes through a large array, it is likely that parts of the array will not be in the cache of the processor. Since access to non-cached memory is *much* slower than access to cached memory, we would like to avoid this.

Many modern processors have *memory prefetch instructions* that tell the processor to load the contents of an address into cache, but unlike a normal load, a memory prefetch does not cause errors if the address is invalid, and it returns immediately without waiting for the load to complete. So a way to ensure that an array element is in the cache is to issue a prefetch of the array element well in advance of its use, but not so well in advance that it is likely that it will be evicted from the cache between the prefetch and the use. Given modern cache sizes and timings, 25 to 10000 cycles ahead of the use is a reasonable time for prefetching – less than 25 increases the risk that the prefetch is not completed before the use, and more than 10000 increases the chance that the value will be evicted from the cache before use.

A prefetch instruction usually loads an entire cache line, which is typically four or eight words, so we do not have to explicitly prefetch every array element – every fourth element is enough.

So, assume we have a loop that adds the elements of an array:

```
sum = 0;
for (i=0; i<100000; i++)
  sum += a[i];
}
```

we can rewrite this to

```
sum = 0;
for (i=0; i<100000; i++) {
  if (i&3 == 0) prefetch a[i+32];
  sum += a[i];
}
```

where `prefetch a[i+32]` prefetches the element of `a` that is 32 places after the current element. The number 32 is rather arbitrary, but makes the number of cycles between prefetch and use lie in the interval mentioned above. Note that we used the test `i&3==0`, which is equivalent to `i%4==0`, but somewhat faster.

We do not have to worry about prefetching past the end of the array – prefetching will never cause runtime errors, so at worst we prefetch something that we do not need.

While this transformation adds a test (that takes time), the potential savings by having all array elements in cache before use are much larger. The overhead of testing can be reduced by unrolling the loop body:

```
sum = 0;
for (i=0; i<100000; i++) {
  prefetch a[i+32];
  sum += a[i];
  i++;
  sum += a[i];
  i++;
  sum += a[i];
  i++;
  sum += a[i];
}
```

This should, of course, only be done if the loop body is small. We have exploited that the number of iterations is a multiple of 4, so the exit test is not needed at every increment of `i`. If we do not know this, the exit test must be replicated after each increase of `i`, like shown here:

```
sum = 0;
for (i=0; i<n; i++) {
  prefetch a[i+32];
  sum += a[i];
  if (++i < n) {
    sum += a[i];
    if (++i < n) {
      sum += a[i];
      if (++i < n) {
        sum += a[i];
      }
    }
  }
}
```

In a nested loop that accesses a multi-dimensional array, you can prefetch the next row while processing the current. For example, the loop

```
sum = 0;
for (i=0; i<1000; i++)
  for (j=0; j<1000; j++)
    sum += a[i][j];
  }
}
```

can be transformed to

```
sum = 0;
for (i=0; i<1000; i++)
  for (j=0; j<1000; j++)
    if (j&3 == 0) prefetch a[i+1][j];
    sum += a[i][j];
  }
}
```

Again, we can unroll the body of the inner loop to reduce the overhead.

11.7 Optimisations for function calls

Modern coding styles use frequent function (or method) calls, so optimising function calls is as worthwhile as optimising loops.

Basically, optimisation of function calls attempt to reduce the overhead associated with call sequences, prologues and epilogues (see chapter 10). We will see a few ways of doing this below.

11.7.1 Inlining

Inlining means replacing a function call by a copy of the body of the function, with some glue code to replace parameter and result passing.

If we have a call (using C-style syntax)

$$x \texttt{ = f(}exp_1\texttt{,...,}exp_n\texttt{);}$$

and the function `f` is defined as

$$\begin{array}{l} type_0 \texttt{ f(}type_1\ x_1\texttt{,...,}type_n\ x_n\texttt{)} \\ \texttt{\{} \\ \quad body \\ \quad \texttt{return(}exp\texttt{);} \\ \texttt{\}} \end{array}$$

where *body* represents the body of the function (apart from the `return` statement) and the shown `return` statement is the only exit point of the function, we can replace the call by the block

$$\begin{array}{l} \texttt{\{} \\ \quad type_1\ x_1 \texttt{ = } exp_1\texttt{;} \\ \quad \texttt{...} \\ \quad type_n\ x_n \texttt{ = } exp_n\texttt{;} \\ \quad body \\ \quad x \texttt{ = } exp\texttt{;} \\ \texttt{\}} \end{array}$$

Note that if, say, exp_n refers to a variable with the same name as x_1, it will refer to a different instance of x_1 after the transformation. So we need to avoid variables in the inlined function shadowing variables used in the function call statement. We can achieve this by renaming the variables in the inlined function to new, previously unused, names before it is inlined.

Note that, unless the body of the inlined function is very small, inlining causes the program to grow in size. Hence, it is common to put a limit on the size of functions that are inlined. What the limit is depends on the desired balance between speed and size. But even when optimising for speed, care should be taken not to inline too much, as large programs can run slower than smal programs due to worse cache behaviour.

Care must be taken if you inline calls recursively: If the inlined body contains a call that is also inlined, and this again contains a call that is inlined, and so on, we might continue inlining forever. So it is common to limit inlining to only one or two levels deep or treat (mutually) recursive functions as special cases.

11.7.2 Tail-call optimisation

A *tail call* is a call that happens just before a return.

As an example, assume we in a function `f` have (using C-style notation) a statement `return(g(x,y));`. Clearly, `f` returns just after `g` returns, and the result of `f` is the result of `g`. We want to combine the call sequence for the call to `g` with the epilogue of `f`. We call this *tail-call optimisation*.

We will exploit the following observations:

- No variables in `f` are live after the call to `g` (since there are not any uses of variables after the call), so there will be no need to save and restore caller-saves variables around the call.
- If we can eliminate all of the epilogue of `f` except for the return-jump to `f`'s caller, we can instead make `g` return directly to `f`'s caller, hence skipping `f`'s epilogue entirely.
- If `f`'s frame holds no useful information at the time we call `g` (or we can be sure that `g` does not overwrite any useful information in the frame), we can reuse the frame for `f` as the frame for `g`.

We will look at this in more detail below.

If we assume a simple stack-based caller-saves strategy like the one shown in figures 10.2 and 10.3, the combined call sequence of the call to `g` and the epilogue of `f` becomes:

$$
\begin{array}{l}
M[FP+4*m+4] := R0 \\
\cdots \\
M[FP+4*m+4*(k+1)] := Rk \\
FP := FP + framesize \\
M[FP+4] := a_1 \\
\cdots \\
M[FP+4*n] := a_n \\
M[FP] := returnaddress \\
\texttt{GOTO g} \\
\texttt{LABEL } returnaddress \\
result := M[FP+4] \\
FP := FP - framesize \\
R0 := M[FP+4*m+4] \\
\cdots \\
Rk := M[FP+4*m+4*(k+1)] \\
M[FP+4] := result \\
\texttt{GOTO } M[FP]
\end{array}
$$

Since there are no live variables after the call to g, we can eliminate the saving and restoring of $R0, \ldots, Rk$, yielding the following simplified code:

$$
\begin{array}{l}
FP := FP + framesize \\
M[FP+4] := a_1 \\
\ldots \\
M[FP+4*n] := a_n \\
M[FP] := returnaddress \\
\texttt{GOTO g} \\
\texttt{LABEL } returnaddress \\
result := M[FP+4] \\
FP := FP - framesize \\
M[FP+4] := result \\
\texttt{GOTO } M[FP]
\end{array}
$$

We now see that all that happens after we return is an adjustment of FP, a copy of the result from g's frame to f's frame and a jump to the return address stored in f's frame.

What we now want is to reuse f's frame for g's frame. We do this by not adding and subtracting $framesize$ from FP, so we get the following simplified code:

$$
\begin{array}{l}
M[FP+4] := a_1 \\
\ldots \\
M[FP+4*n] := a_n \\
M[FP] := returnaddress \\
\texttt{GOTO g} \\
\texttt{LABEL } returnaddress \\
result := M[FP+4] \\
M[FP+4] := result \\
\texttt{GOTO } M[FP]
\end{array}
$$

It is immediately evident that the two instructions that copy the result from and to the frame cancel out, so we can simplify further to

$$
\begin{array}{l}
M[FP+4] := a_1 \\
\ldots \\
M[FP+4*n] := a_n \\
M[FP] := returnaddress \\
\texttt{GOTO g} \\
\texttt{LABEL } returnaddress \\
\texttt{GOTO } M[FP]
\end{array}
$$

We also see an unfortunate problem: Just before the call to g, we overwrite f's return address in $M[FP]$ by g's return address, so we will not return correctly to f's

caller (we will, instead, get an infinite loop). However, since all that happens after we return from `g` is a return from `f`, we can make `g` return directly to `f`'s caller. We do this simply by not overwriting `f`'s return address. This makes the instructions after the jump to `g` unreachable, so we can just delete them. This results in the following code:

$$
\begin{array}{l}
M[FP+4] := a_1 \\
\ldots \\
M[FP+4*n] := a_n \\
\texttt{GOTO g}
\end{array}
$$

With this tail-call optimisation, we have eliminated the potential ill effects of reusing `f`'s frame for `g`. Not only have we shortened the combined call sequence for `g` and epilogue for `f` considerably, we have also saved stack space. Functional programming languages rely on this space saving, as they often use recursive tail calls where imperative languages use loops. By doing tail-call optimisation, an arbitrarily long sequence of recursive tail calls can share the same stack frame, so only a constant amount of stack space is used.

In the above, we relied on a pure caller-saves strategy, since the absence of live variables after the call meant that there would be no saving and restoring of caller-saves registers around the call. If a callee-saves strategy or a mixed caller-saves/callee-saves strategy is used, there will still be no saving and restoring around the call, but `f`'s epilogue would restore the callee-saves registers that `f` saved in its own prologue. This makes tail-call optimisation a bit more complicated, but it is normally possible (with a bit of care) to move the restoring of callee-saves registers to *before* the call to `g` instead of waiting until after `g` returns. This allows `g` to overwrite `f`'s frame, which no longer holds any useful information (except the return address, which we explicitly avoid overwriting).

Exercise 11.4 asks you to do such tail-call optimisation for a mixed caller-saves/callee-saves strategy.

11.8 Specialisation

Modern programs consist mainly of calls to predefined library functions (or procedures or methods). Calls to such library functions often fix some of the parameters to constants, as the function is written more generally than needed for the particular call. For example, a function that raises a number to a power is often called with a constant as the power, say, `power(x,5)` which raises `x` to its fifth power. In such cases, the generality of the library function is wasted, and speed can be gained by using a more specific function, say, one that raises its argument to the fifth power.

A possible implementation of the general `power` function (in C) is:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

If we have a call `power(x,5)`, we can replace this by a call `power5(x)` to a specialised function. We now need to add a definition of this specialised function to the program. The most obvious idea would be to take the above code for the `power` function and replace all occurrences of `n` by 5, but this will not work, as `n` changes value inside the body of `power`. What we do instead is to observe the following:

1. The loop condition depends only on `n`.
2. Every change to `n` depends only on `n`.

So it is safe to unroll the loop at compile-time, doing all the computations on `n`, but leaving the computations on `p` and `x` for run-time. This yields the following specialised definition:

```
double power5(double x)
{
  double p=1.0;
  p = p*x;
  x = x*x;
  x = x*x;
  p = p*x;
  return(p);
}
```

Executing `power5(x)` is, obviously, a lot faster than executing `power(x,5)`. Since `power5` is fairly small, we can additionally inline the call, as described in section 11.7.1.

This kind of specialisation may not always be applicable, even if a function call has constant parameters, for example if the call was

`power(3.14159,p)`, where `p` is not a constant, but when the method is applicable, the speedup can be dramatic.

Similar specialisation techniques are used in C++ compilers for compiling templates: When a call specifies template parameters, the definition of the template is specialised with respect to the actual template parameters. Since templates in C++ can be recursively defined, an infinite number of specialised versions might be required. Most C++ compilers put a limit on the recursion level of template instantiations and stop with an error message when this limit is exceeded.

Suggested exercises: 11.5.

11.9 Further reading

We have covered only a small portion of the optimisations that are found in optimising compilers. More examples of optimisations (including value numbering) can be found in advanced compiler textbooks, such as [4, 7, 9, 35].

A detailed treatment of program analysis can be found in [36]. Specialisation techniques like those mentioned in section 11.8 are covered in more detail in [17, 21]. The book [34] has good articles on both program analysis and transformation.

Additionally, the conferences "Compiler Construction" (CC), "Programming Language Design and Implementation" (PLDI) and other programming-language-oriented conferences often present new optimisation techniques, so past proceedings from these is a good source for advanced optimisation methods.

Exercises

Exercise 11.1

In the program in figure 11.2, replace instructions 13 and 14 by

13: $h := n * 3$
14: `IF` $i < h$ `THEN` *loop* `ELSE` *end*

a) Repeat common subexpression elimination on this modified program.

b) Repeat, again, common subexpression elimination on the modified program, but, prior to the fixed-point iteration, initialise all sets to the empty set instead of the set of all assignments.

What differences does this make to the final result of fixed-point iteration, and what consequences do these differences have for the optimisation?

Exercise 11.2

Write a program that has jumps to jumps and perform jump-to-jump optimisation of it as described in section 11.3. Try to make the program cover all the three optimisation cases described at the end of section 11.3.

Exercise 11.3

a) As described in the beginning of section 11.4, add extra labels and gotos for each `IF-THEN-ELSE` in the program in figure 11.7.

b) Do the fixed-point iteration for index-check elimination on the result.

c) Eliminate the redundant tests.

d) Do jump-to-jump elimination as described in section 11.3 on the result to remove the extra labels and gotos introduced in question a.

Exercise 11.4

Section 11.7.2 describes tail-call optimisation for a pure caller-saves strategy. Things become somewhat more complicated when you use a mixed caller-saves/callee-saves strategy.

Using the call sequence from figure 10.10 and the epilogue from figure 10.9, describe how the combined sequence can be rewritten to get some degree of tail-call optimisation. Your main focus should be on reusing stack space, and secondarily on saving time.

Exercise 11.5

Specialise the `power` function in section 11.8 to `n` = 12.