TDT4205 Compilers
Exercise 6

Stian Hvatum (hvatum)
MTDT

April 12, 2012

Contents

PART 1 Theory

Task 1.1 Optimization

1. Easy and common

One easy and common optimization done by compilers is common sub-expression elimination. A common sub-expression is a an expression that occurs multiple times either alone or within another expression during the space where the involved variables are not redefined.

```
1 int main(int argc, char** argv)
2 {
3     int a = 5;
4     int b = 10;
5     int c = 8;
6     int d = a + b * c;
7     int e = d - b * c;
8 }
```

At line 6 and 7 we see that $b \cdot c$ occurs at both lines. We can reduce the cost of mulitplying with this code:

```
1 int main(int argc, char** argv)
2 {
3     int a = 5;
4     int b = 10;
5     int c = 8;
6     int tmp = b * c;
7     int d = a + tmp;
8     int e = d - tmp;
9 }
```

We must notice that sometimes it is still cheaper to calculate the sub-expression twice, since pre-calculating it may involve more memory useage and eventually force the processor to use cache or main memory instead of the faster registers.

2. Hard to do

Even for compilers utilizing Data-flow analysis, it is hard[1] to do a complete general and correct calculation of Reaching Definitions. Reaching Definitions is a way to find

[1] Actually the problem is NP-Complete and mathematically undecidable

out which parts of a program that may be executed, and which may not, according to value-dependent branching.

The reason why this is hard to do, is because it is (in general) impossible to determin all possible values each variable may take given its surrounding program. With data-flow analysis, we may determin some of these and maybe also discover tautologies and contradiction, and thus we can eliminate some paths. But since we cannot determin all possible values and paths, some paths will be let though even though they never are executed.

3. Optimalization impact

Copy Progragation

L7: g = f, where f = e from L6.

```
1  a=1
2  b=2
3  c=3
4  d=a+x
5  e=b+c
6
7  g=e
8  g=d+y
9  a=b+c
```

Common subexpression elimination

L5 and L9 both has b + c.

```
1  a=1
2  b=2
3  c=3
4  d=a+x
5  t=b+c
6  e=t
7  f=e
8  g=f
9  g=d+y
10 a=t
```

Contant propagation

a, b and c are all assigned to contants. There are used in L4, L5, L8, L9, and can be replaced with the assigned number.

```
1  a=1
2  b=2
```

```
3  c=3
4  d=1+x
5  e=2+3
6  f=e
7  g=f
8  g=d+y
9  a=2+3
```

We could also used constant folding to eliminate all lines except L1, L4 and L8, where we have unknown variables, given none of the other variables are used after the given code block. If dead code elimination was used, we could eliminated L7, since g is assigned to something else at L8.

Task 1.2 Misc

1. Array bounds checking

To do bounds checking on array lookups and assignments, we need to know how large the array is, and we need to check this lenght against the value of the lookup-variable. This means that we need to store the array lenght somewhere, most likely in the symbol table, and we need to add some extra assembly code around each array lookup and assignment, unless the lookup is a constant or given to be within bounds by external factors.

A high level abstraction of this:

```
1     int array[5];
2
3     /* No check is needed here, since index is constant */
4     array[3] = 2;
5
6    /* Here we need to check the index,
7       and throw an error if out of bounds  */
8
9     if (i >= 0 && i < 5)
10        array[i] = 4;
11    else
12        throw index_out_of_bounds;
```

2. Pointer arithmetics

Live variables When we are looking for live variables, we usually go backwards and check where the last occurence of each variable is used. Since pointers points to data areas, there may be problems related to when that memory area was read. If you write

to a pointers data area, and read another pointers data area, they may refer to the same area, and you cannot declare the first one as dead, even if it is written only (and never "used").

Available expressions Regarding available expressions, we have sort of the same problem as with Live variables. You may have great trouble knowing wherever your expression result is stil valid if your data area has been modified by another pointer pointing to the same location.

3. Graph coloring and register spilling

Register allocation is the problem of assigning registers to those results and variables that is needed the most in the current state. Since you have a given number of registers, and the expression calculated at the given state holds a number of variables you can reduce this to the Graph coloring problem, where the colors are registeres and the areas of the graph are the variables. Each sub-expression has to be calculated without using the same register twice, hence we cannot have two neighbouring areas that share the same color.

Register spilling is the phenomenon that occurs once we don't have enought registers to calculate a problem. When register spilling occures, preformance is greatly reduced. One of the main objectives of register allocation is to minimize the occurences of memory spilling.

4. DAGs

Compilers often represent basic blocks as DAGs because it eases some optimizations, eg.

- **Local common subexpression elimination**
  We will see from the DAG that if two calculations are the same, they will end up in the same node, and thus are equal at that point.
- **Dead code elimination**
  By removing all roots with no attached live variables until there are no more such nodes, we have effectivly removed all dead code from the block.
- **Statement reoder**
  We can reorder statements that does not depend on each other, such that we get more common subexpressions that may be eliminated, eg.
  $a = b + c$;
  $d = b + f + c$;
  becomes
  $a = b + c$;
  $d = b + c + f$;

where we see the common subexpression $b + c$. We must of course be avare of precedence and parentheses when doing such optimizations. The DAG will show us which statements that are dependent on each other.

- **Simplify computation**
  We can do *constant folding* and *strenght reduction* on the expressions within the DAG. Eg. $x^2 = x \cdot x$, $2 \cdot x = x + x$ and $2 + 2 = 4$. The DAG is not strictly required for this, but it is very easy to make these changes on this kind of node-oriented data structures. Using this kind of sustitutions often render some of the used nodes as dead, so that they may be eliminated.