Exercise Sheet Solutions 7

Compiler Design Lab
Prof. Dr. Sebastian Hack
Fabian Ritter, B.Sc.

This is a solution proposal. It does not necessarily describe complete solutions but gives the initial ideas to solve the theoretical exercises and presents the results.

Exercise 7.1 Interval Analysis: Overflows and Soundness

The interval analysis as we have seen it in the lecture is only defined for programs that use mathematical integers ($\mathbb{Z}$) rather than fixed-width machine integers ($\mathbb{B}^n$ for e.g. $n = 32$). The abstract transformers for this analysis are obviously unsound if the analyzed programming language specifies integers to wrap on overflow (e.g. Java's integers and C's unsigned integers).

Elaborate on whether the defined abstract transformers are sound for C's signed integer arithmetic (where overflows result in undefined behavior), i.e. argue whether compiler transformations based on information computed with these transformers would preserve the semantics of the program. What if we want to use the analysis to verify properties of a program rather than to justify compiler transformations?

Solution:

Compiler optimizations Whenever no overflow occurs, interval arithmetic on mathematical integers and on (signed) machine integers coincides. Therefore, if we require that the transformations that our compiler performs are valid in a program that computes in mathematical integers, we will not miscompile our machine integer program for overflow-free executions. As the behavior of signed integer overflow is undefined in C, it is literally not possible for a compiler to miscompile a program for executions with signed integer overflows. Hence, we can use our interval analysis for mathematical integers to soundly justify program transformations.

Verification This question becomes more philosophical to answer if we are concerned with verifying that systems that involve C programs (e.g. a running processor) preserve certain properties. One way to argue could be: In this case, a program execution having undefined behavior means that the system is allowed to do anything, especially also to violate any properties of interest. Therefore, we would need to make sure that if any execution might contain signed integer overflows, our analysis does not claim to verify any property of the system. The abstract transformers for our mathematical interval domain as presented in the lecture do not ensure this, hence they are unsound for system verification purposes.

Exercise 7.2 Interval Analysis: Division

In the lecture, we defined the abstract division operation $[l_1, u_1] /^{\#} [l_2, u_2] = [a, b]$ for our interval analysis as follows:

$$\begin{aligned} a &= l_1/l_2 \sqcap l_1/u_2 \sqcap u_1/l_2 \sqcap u_1/u_2 \\ b &= l_1/l_2 \sqcup l_1/u_2 \sqcup u_1/l_2 \sqcup u_1/u_2 \end{aligned} \qquad \text{if } 0 \notin [l_2, u_2]$$

and

$$[a, b] = [-\infty, +\infty] \qquad \text{if } 0 \in [l_2, u_2]$$

This can be imprecise for cases where the divisor interval contains 0 if we decide to conclude from the occurence of a division a/b that b cannot have the value 0 in any valid execution.

Improve the abstract interval division such that it is also precise for this case.

Solution: If we do not assume division by zero to result in ∞, the extremal results of the interval division occur for the divisors -1 and 1 (rather than for the bounds of the argument intervals). Therefore, we replace the second case of the defintion with the following:

$$\begin{aligned} a &= l_1/l_2 \sqcap l_1/u_2 \sqcap u_1/l_2 \sqcap u_1/u_2 \sqcap l_1 \sqcap -u_1 \\ b &= l_1/l_2 \sqcup l_1/u_2 \sqcup u_1/l_2 \sqcup u_1/u_2 \sqcup -l_1 \sqcup u_1 \end{aligned} \qquad \text{if } \{-1, 0, 1\} \in [l_2, u_2]$$

Further,

$$[l_1, u_1]/[0, u_2] = [l_1, u_1]/[1, u_2]$$

and

$$[l_1, u_1]/[l_2, 0] = [l_1, u_1]/[u_2, -1]$$

Finally, we define $[l_1, u_1]/[0, 0]$ to be $\bot$.

Exercise 7.3 Interval Analysis: Widening

The lecture slides give a simple widening operator for the interval analysis (slide 205 of the corresponding slide set). In this exercise, you will improve on this:

1. Define a widening operator $\nabla : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ for intervals that leads to more precise results than the one from the slides.
2. Argue that it fulfills the requirements of a widening operator:
   - It computes an upper bound: $a \nabla b \sqsupseteq a$ and $a \nabla b \sqsupseteq b$.
   - Any widening sequence $(G^i(\bot))_{i \in \mathbb{N}}$ with $G(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ where $y_i = x_i \nabla f_i(x_1, \ldots, x_n)$ can only increase a finite number of times.
3. Provide an example program and an application of the interval analysis with your widening operator for this program that shows that it computes more precise results than the widening operator from the slides.

Solution:

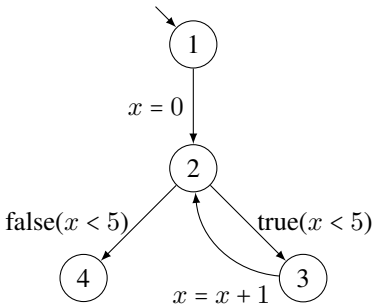
1. We define for $[l, u] = [l_1, u_1] \nabla [l_2, u_2]$:

$$l = \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -42 & \text{if } l_1 > l_2 \wedge l_2 \geq -42 \\ -\infty & \text{otherwise} \end{cases}$$

$$u = \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ 42 & \text{if } u_1 < u_2 \wedge u_2 \leq 42 \\ \infty & \text{otherwise} \end{cases}$$

   (It follows the definition of the one from the lecture except that before immediately jumping to +/-∞, this widening operator goes to +/-42 first.)

2. We argue that it fulfills the requirements of a widening operator:
   - In the notation of the definition, we have to argue that $\max(u_1, u_2) \geq u$ (and, dually, $\min(l_{1,l}\, 2) \leq l$). If $u_1 \geq u_2$, we know that $u = u_1 \geq u_1$ and $u = u_1 \geq u_2$, which shows our claim for this case. In case $u_1 < u_2 \wedge u_2 \leq 42$, we see that $u = 42 \geq u_2$ and $u = 42 \geq u_2 > u_1$, which again shows our claim. For the last case, we know that $u = \infty$ which is greater than or equal to any possible value.
     We can do a dual argument to show that $\min(l_{1,l}\, 2) \leq l$.
   - From the definition of our widening, we can see that any sequence that combines abstract values with the widening can only increase at most 2 times for each bound of each component. Hence, there can be no infinite widening sequence.
3. Consider the following program:



Using our new widening operator, we get the following iteration (tracking only the values of x):

#	1	2	3	4
0	$\bot$	$\bot$	$\bot$	$\bot$
1	$\top$	$[0,0]$	$[0,0]$	$\bot$
2	$\top$	$[0,42]$	$[0,42]$	$[5,42]$
3	$\top$	$[0,+\infty]$	$[0,42]$	$[5,+\infty]$
4	$\top$	$[0,+\infty]$	$[0,42]$	$[5,+\infty]$

So, we can find out that the value of x at program point 2 lies in $[0,42]$. This is clearly an improvement over $[0,\infty]$, which the widening from the lecture would provide.

Exercise 7.4 Global Value Numbering

Reconsider Kildall's Algorithm and the AWZ Algorithm for global value numbering as introduced in the lecture. They describe two different approaches to detect equivalence of program expressions.

1. Give an example for a loop-free program for which Kildall's approach can detect more equivalences than the AWZ algorithm can.
2. Perform both analyses on your example program to show that Kildall's approach really results in more detected equivalences. Make the intermediate steps of your fixed point iteration explicit for both analyses.
3. Describe at which point of the AWZ Algorithm the imprecision arises and explain why this happens.

Solution:

1. Consider the following program:

```
if (...) {
  a = 2;
  x = a + 1;
}
else {
  a = 3;
  x = a + 1;
}
y = a + 1;
... = ... x ... ;
... = ... y ... ;
```

For this program only Kildall's approach can detect that x and a + 1 are identical before executing the assignment to x. So in fact we can replace y = a + 1; by y = x;. We safe one computation of a + 1. A later copy propagation analysis could go even further and also find out that we can safely remove also y = x; and replace all the follwing uses of y by x.

2. We assume that there are no further loops surrounding our example program. This means we have no cycles in the control flow graph and also not in the SSA graph.

   Kildall's approach:

   Since the control flow graph on which we calculate partitionings for each program point is non-circular, the fixed point iteration will terminate after visiting each point exactly once if we use a visiting order that respects the execution order of the program (Kildall's approach is a forward analysis!).

   Directly after the condition we have the following partitioning since we know nothing about any equalities at the start:

   $$\{[1], [2], [3], [a], [x], [y], [a+1]\}$$

   Now we enter the then-branch:

   Before a = 2; we have still:

   $$\{[1], [2], [3], [a], [x], [y], [a+1]\}$$

   After a = 2; we have:

   $$\{[1], [3], [a, 2], [x], [y], [a+1]\}$$

   After x = a + 1; we have:

   $$\{[1], [3], [a, 2], [y], [x, a+1]\}$$

   Now we enter the else-branch:

   Before a = 3; we have still:

   $$\{[1], [2], [3], [a], [x], [y], [a+1]\}$$

   After a = 3; we have:

   $$\{[1], [2], [a, 3], [x], [y], [a+1]\}$$

   After x = a + 1; we have:

   $$\{[1], [2], [a, 3], [y], [x, a+1]\}$$

   Now we do the merge between the partitionings coming from the if-branch and the else-branch:

   $$\{[1], [2], [3], [a], [y], [x, a+1]\}$$

   So this means that x and a + 1 are considered equal before executing y = a + 1;. So we can safe one computation and just reuse x.

   AWZ algorithm:

```
if (...) {
  a1 = 2;
  x1 = a1 + 1;
}
else {
  a2 = 3;
  x2 = a2 + 1;
}
a3 = φ3(a1, a2)
x3 = φ3(x1, x2)
y1 = a3 + 1;
... = ... x3 ... ;
... = ... y1 ... ;
```

The analysis used in the AWZ algorithm is not flow sensitive. So we start optimistically with a partitioning assuming for the whole program:

$$\{[1, 2(a_1), 3(a_2), \phi_3(a_1, a_2)(a_3), \phi_3(x_1, x_2)(x_3), a_1 + 1(x_1), a_2 + 1(x_2), a_3 + 1(y_1)]\}$$

The algorithm runs on the SSA graph.

The leaves are constants. The top level symbol of each constant is different from all other top level symbols in the partition. In the same way we directly separate the partitions along the borders of different top level symbols:

$$\{[1], [2(a_1)], [3(a_2)], [\phi_3(a_1, a_2)(a_3), \phi_3(x_1, x_2)(x_3)], [a_1 + 1(x_1), a_2 + 1(x_2), a_3 + 1(y_1)]\}$$

According to our current partitioning we have a_1, a_2 and a_3 in disjoint partitions. This means the first operands of the three addition nodes are all in different partition and therefore the additions also have to be split up according to Herbrand:

$$\{[1], [2(a_1)], [3(a_2)], [\phi_3(a_1, a_2)(a_3), \phi_3(x_1, x_2)(x_3)], [a_1 + 1(x_1)], [a_2 + 1(x_2)], [a_3 + 1(y_1)]\}$$

Now only the ϕ-nodes still have a common partition. Since their first parameters x_1 and a_1 are in different partitions, they also have to be split up:

$$\{[1], [2(a_1)], [3(a_2)], [\phi_3(a_1, a_2)(a_3)], [\phi_3(x_1, x_2)(x_3)], [a_1 + 1(x_1)], [a_2 + 1(x_2)], [a_3 + 1(y_1)]\}$$

3. The problem is here that the algorithm interpretes the ϕ-nodes in a purely syntactical manner. We just see them as a special top level symbol and cannot know that there are essentially other nodes directly referenced by it depending from which preceding block the control arrives.

   If we could look through the ϕs then we could see that coming from the then block a_3 is a_1 and x_3 is $a_1 + 1$. Then we could conclude that $a_1 + 1$ and $a_3 + 1$ are the same. So x_3 is the same as $a_3 + 1$.

   The same argumentation can also be done for looking through the ϕs when coming from the else block.

   The algorithm would have to be extended in an approriate way to support this.