Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.820 Foundations of Program Analysis

Problem Set 5 Out: November 6, 2015
Due: November 25, 2015 at 5:00 PM

For this homework you will implement an analysis based on abstract interpretation that will verify complex routines involving pointer manipulations and will attempt to certify that they cannot make any illegal memory accesses. In this assignment, you will be introduced to the concept of Heap abstractions; that is, abstract domains that can be used to reason about the heap. In this exercise, we will do a very simplified form of heap analysis based on predicate abstraction and interval domains. You will be working with the same language you used for PSet 4, extended with memory allocation and pointer reading and writing.

Concrete semantics. For this exercise we will use a very simple model of the semantics of memory allocation and usage in C. In this model, like in real C, a call to $malloc(k)$ results in the allocation of a buffer of size k. Unlike C, however, we will assume that k is the size in words (as opposed to bytes) and that all values will be exactly one word, so no need to worry about alignment. In our model, each pointer can be broken into two parts: a base address pointing to the beginning of a buffer, and an offset into that buffer. Pointer arithmetic can be used to change the offset, but it cannot be used to make the pointer point to a different buffer. For simplicity, our model will not deal with deallocation.

The heap will be represented with two functions: $h_d : addr \times offset \rightarrow value$ maps a base address and an offset to the value stored at that address, and the function $h_s : addr \rightarrow size$ maps a base address to the size of the buffer that starts at that address. In addition to the heap, we use an environment σ to track local variables. The semantics of our language are fairly straightforward; the main new features compared to the semantics we have seen in class are those expressions and statements that use the heap. Expressions in our language take a heap and a local environment and produce a value, which can be either an integer value or an (addr,offset) pair. For example, when dereferencing an expression e , the expression must evaluate to a pointer whose offset is smaller than the size of buffer it points to.

$$\frac{\langle e,\ (\sigma,\ h_d,\ h_s)\rangle \rightarrow (l,t) \quad 0 \leq t < h_s(l) \quad h_d(l,t) = v}{\langle *e\ ,\ (\sigma,\ h_d,\ h_s)\rangle \rightarrow v}$$

As we said before, pointer arithmetic can only be used to add integer offsets to pointers

$$\frac{\langle e_p,\ (\sigma,\ h_d,\ h_s)\rangle \rightarrow (l,t) \quad \langle e_i,\ (\sigma,\ h_d,\ h_s)\rangle \rightarrow n}{\langle e_p +\ e_i\ ,\ (\sigma,\ h_d,\ h_s)\rangle \rightarrow (l,\ t+n)}$$

Since $malloc(k)$ can modify the heap, we won't treat it as an expression; instead, we treat $x := malloc(k)$ as a special kind of statement. Notice that malloc initializes the entire buffer to zero.

$$\frac{\langle e,\ (\sigma,\ h_d,\ h_s)\rangle \to k \qquad l \textit{ is a fresh address}}{\langle x := \mathit{malloc}(e)\ ,\ (\sigma,\ h_d,\ h_s)\rangle \to (\sigma[x \to (l,0)],\ h_d[(l,0) \to 0,\ \ldots,\ (l,(k-1)) \to 0],\ h_s[l \to k])}$$

The other new kind of statement is assignment into memory

$$\frac{\langle e_1,\ (\sigma,\ h_d,\ h_s)\rangle \to (l,t) \qquad 0 \le t < h_s\,(l) \qquad \langle e_2,\ (\sigma,\ h_d,\ h_s)\rangle \to v}{\langle *e_1 := e_2\ ,\ (\sigma,\ h_d,\ h_s)\rangle \to (\sigma,\ h_d\,[(l,t) \to v]\,,\ h_s)}$$

With this semantic model, we are now ready to define our abstract domain. There are a lot of things we are not modeling, such as the different widths of different data types or various alignment issues, but this simple model will suffice to allow us to reason about interesting properties of heap manipulating programs.

Problem 1 — Setting up the abstract domain (25 points)

In order to do abstract interpretation, we need to define a lattice of abstract values. The high-level idea behind our abstraction is that we want to use a one abstract address to represent all the buffers created at the same allocation site. In addition to that, we need an abstract address to represent buffers allocated outside the scope of the current code, and one to represent null. As for integer values, we will use a simple flat lattice of integers to allow us to check for accesses that may exceed the buffer size.

For example, consider the code below (variable declarations have been omitted for convenience):

```
// all variables initially point to something in the outside world = allocation site out
   h = malloc(40);  // allocation site 1
   y = h;
   while(*Nref>0){
      y = y+ 4;
      *y = malloc(40);  // allocation site 2
      y  = *y;
      *Nref = (*Nref) - 1;
   }
```

For this code, our analysis should be able to tell us that the two pointer dereferences of `y` will always be legal, but that the dereferences of `Nref` may fail. The analysis will maintain four abstract locations, one corresponding to allocation site 1 and one to allocation site 2. In addition to those, there will be the abstract location corresponding to the outside world, and the one corresponding to null:

ID	Corresponding concrete values
null	A value which is not a valid address
out	Addresses of objects allocated outside the scope of the code
1	Addresses of buffers allocated at site 1
2	Addresses of buffers allocated at site 2

The set of abstract addresses, including *null* and *out* (which refers to objects allocated outside the scope of the current code) will be called *Alloc* . Note that the abstract addresses in *Alloc* do not form a lattice. In order to construct a lattice from the abstract elements, we will use a powerset construction, so our abstract domain will correspond to subsets of *Alloc*; **T** will be equal to *Alloc* , $\perp$ will be the empty set, and the partial order will be determined by the subset relation. We will use $\mathcal{P}(Alloc)$ to refer to the powerset lattice. Now, for integers (including offsets and buffer lengths), we will use a simple flat lattice like the one used for constant propagation. We will call this lattice $Int = Z \cup \{\top \perp\}$.

As we said in the concrete semantics, pointers actually have two components, a base address and an offset, so they need to be represented by a lattice corresponding to the cross product $\mathcal{E} = \mathcal{P}(Alloc) \times Int$. In principle, integer variables should have values in Int , while pointer variables should have pointers of type *Alloc*. However, we can simplify the implementation of the analysis by using the $\mathcal{E}$ lattice to represent both integers and pointers, eliminating the need to reason about the type of a value. In particular, a constant like 5 would have an abstract value $(\perp, 5)$. This design choice will mean that the analysis will not be able to detect cases where pointers and integers are being mixed illegally, but such cases are best checked by a type system. At the beginning of the program, all variables have the abstract state $(\{out, null\}, \top)$.

The abstract state Now, we need to define the abstract state of the program, and how the abstract state is updated by the execution of the abstract program. The abstract state of the program is going to consist of a $\bar{\sigma}$ which maps variable names to elements in the combined lattice. We will use an abstract heap $\bar{h_d}$ which maps an abstract location and an offset abstract values, and an $\bar{h_s}$ that keeps information about buffer sizes.

$$\bar{\sigma} : var \rightarrow \mathcal{E} \quad \bar{h_d} : (Alloc \times Z) \rightarrow \mathcal{E} \quad \bar{h_s} : (Alloc) \rightarrow Int$$

Using the abstract state, we can define the abstract semantics of expressions and assignments; for example, for variables, constants and object allocation, you have:

$$\frac{\left\langle e \ , \ (\bar{\sigma}, \ \bar{h_d}, \ \bar{h_s}) \right\rangle \rightarrow (a, \ k) \quad mallo\ c_l is\ the\ l^{th}\ allocation\ site}{\left\langle x := mallo\ c_l\,(e)\,, \ \left(\bar{\sigma}, \ \bar{h_d}, \ \bar{h_s}\right) \right\rangle \rightarrow \left(\sigma[x \rightarrow (\{l\}\,, 0)], \ \bar{h_d}[(l,0) \rightarrow (\perp, 0) \sqcup \bar{h_d}\,(l,0)\,, \ldots, (l, n_k) \rightarrow (\perp, 0) \sqcup \bar{h_d}\,(l, n_k)], \ \bar{h_s}[l \rightarrow k \ \sqcup \bar{h_s}\,(l)]\right)}$$

where n_k is the maximum concrete value in $\gamma(k)$. Note that if k is $\top$, then logically speaking, all entries of the form $\bar{h_d}[(l, \ x)]$ for all x will have to be updated. You implementation needs to be able to account for this.

$$\left\langle NULL \ , \ (\bar{\sigma}, \ \bar{h_d}, \ \bar{h_s}) \right\rangle \rightarrow (\{null\}\,, \ 0)$$

$$\left\langle n \ , \ (\bar{\sigma}, \ \bar{h_d}, \ \bar{h_s}) \right\rangle \rightarrow (\perp, \ n)$$

$$\frac{\langle e_1,\ \ (\bar{\sigma},\ \bar{h_d},\ \bar{h_s})\rangle \to (a_1, n_1) \quad \langle e_2,\ \ (\bar{\sigma},\ \bar{h_d},\ \bar{h_s})\rangle \to (a_2, n_2)}{\langle e_1 + e_2,\ \ (\bar{\sigma},\ \bar{h_d},\ \bar{h_s})\rangle \to (a_1 \sqcup a_2,\ n_1 + n_2)}$$

$$\langle x\ ,\ (\bar{\sigma},\ \bar{h_d},\ \bar{h_s})\rangle \to \bar{\sigma}(x)$$

Notice that the address component of each heap location is initialized to $\perp$ instead of *null*. This allows the analysis to determine, for example, that a location cannot be null. A negative consequence of this decision, however, is that the analysis will not be able to tell that a memory location may be uninitialized, since a location that is initialized on only one branch of a conditional, for example, will just look like it has been fully initialized.

Part a: Answer the following Questions (25 points)

- Write the rule for memory access $(*x)$.
- Write the rule for memory update $*x = e$. Note that with this abstraction, updates cannot be *destructive*. For example, suppose $h_d\,(loc1,\ 5) = (a, b)$; if you have a variable x with abstract value (loc1, 5), and you do a heap update through x (i.e. you write *x = (c,d)), the new values (c,d) cannot destroy the old values (a,b), because $h_d\,(loc1,\ 5) = (a, b)$ tells you something about the values held at position 5 of all the buffers ever allocated by allocation site 1, but you don't know how many buffers there are, or which ones have which values, so the best you can do is to update $h_d\,(loc1,\ 5)$ *to* $(a \sqcup c, b \sqcup d)$.
- Perform the abstract interpretation by hand and show how the abstract state is going to look like at the end of the program below:

```
l = malloc(2); // allocation site 0
p = malloc(10);  // allocation site 1
*l = p;
 t = l+1;
*t = p;
while(*){
  p = p+ 1;
  x  =malloc(10);  // allocation site 2
  *p = x;
  p  = *p;
  *t = p;
}
```

Problem 2 — Implementing the analysis (75+ points)

The starter code for this assignment includes the following files.

- lexer.mll
- parser.mly
- main.ml
- implang.ml
- absint.ml
- cprop.ml

You should be familiar with most of these files from the last pset. The two new files are `absint.ml`, which includes functions for creating control flow graphs from programs and for performing abstract interpretation on them, and `cprop.ml` which implements a very simple analysis based on abstract interpretation for constant propagation. In `main.ml` you will find code that parses a program from stdin, generates a control flow graph, runs the constant propagation analysis on it and then prints the abstract state at the beginning of every basic block.

Your main deliverable is to output a list of statements with each of them labeled with whether or not a memory error can occur at that statement, either because of a null pointer dereference or because of a memory access outside the bounds of an allocated buffer (you don't have to worry about accesses to uninitialized memory locations). The skeleton code we have provided contains a function called 'printResult' that you must use to print all your results. The function currently prints out "YES" for every statement, so you should change it so that it prints NO on those statements that are not guaranteed safe according to your analysis. The grading strategy is as follows: we will run your analysis on a test suite and compare the results with those of our own implementation of the analysis described in this homework. For this we will compute the following numbers:

- C = the number of statements where your analysis conservatively said NO but our analysis said YES.
- S = the number of statements where our analysis conservatively said NO, but your analysis correctly said YES.
- F= the number of statements where your analysis incorrectly said YES.
- N= the total number of statements where our analysis said NO (for normalization purposes).

If your analysis crashes on a test we will treat it as if it had said "YES" to everything on that test. Your total score for this section will be computed as $(3 * (N - F) - C + 2 * S)/(4 * N) * 100$. So the idea is that if your analysis matches exactly ours, you will get the full 75 pts (since F,C and S=0), and if your analysis is perfectly accurate, you cannot get more than 125 pts (since $S < N$). (dont worry, we wont give you negative points). The following are things you can implement to increase the precision of your analysis in order to get those extra 50 points (in order of highest to smallest payout for a given amount of effort):

- Filter your abstract state on branches. In particular, branches of the form `if (t == null)`.

- Keep track of allocation sites that will only execute once so you can do destructive updates on those regions in the abstract heap.
- Implement a more sophisticated abstraction for the integer part of the abstract value (such as an interval domain).

In order to qualify for the extra credit, we ask that you also provide at least 4 tests where the improved precision of the analysis makes a difference.

6.820 Fundamentals of Program Analysis

Fall 2015