CS357 - Assignment 2

Due: November 2, 2015, 23:59

In this exercise we will explore the use of an SMT solver to reason about C programs. We will re-implement the first assignment using an SMT solver (rather than a SAT solver), and then extend it to handle pointers and memory. We will explore three different strategies to encode the memory to SMT, and prove that C programs have no memory errors.

Instructions

0. **Getting started**

   (a) Obtain a parser for C. You can reuse your implementation from assignment one, or use the starter package available at http://web.stanford.edu/class/cs357/handouts/starter-pack.tar.gz. As before, we recommend you use pycparser (https://pypi.python.org/pypi/pycparser), which is used in the starter package.

   (b) Obtain Z3 (https://github.com/Z3Prover/z3), the SMT solver we recommend to use for this exercise. Z3 comes with a Python API, which makes it easy to use it together with the starter package. If you have used other SMT solvers before, you are free to try those, too.

1. **Part I: Re-implement assignment 1 using an SMT solver**

   (a) Build a translator that can convert a fragment of C code to an SMT formula. Unlike in the first assignment, use the theory of bitvectors to represent expressions.

   Like in the first assignment, your implementation needs to support at least the following features:

   - arithmetic operations on integers, excluding multiplication, division and modulus
   - integer shift operations
   - bitwise operations on integers
   - 'char' (8-bit) and 'int' (32-bit) integer sizes, both signed and unsigned
   - relational operators on integers (==, <, > etc.)
   - C's implicit integer sign conversion and widening/narrowing rules (e.g., when applying an operator to integer values of different sign and/or width, or when the target of an assignment has different sign and/or width from the assigned value)
   - logical negation
   - 'if' conditions
   - 'for' and 'while' loops
   - integer multiplication (new)

   You will need to unroll loops. All the loops in the examples below have relatively small upper bounds on the number of iterations, so unrolling works fine.

   Since SMT solvers make it much easier to implement various features of C, also implement integer multiplication (which was part of the "extra credit" section in assignment one).

(b) Take all examples from assignment one and prove the equivalence for the different implementations. Make sure you consider all implementations for which your tool supports all operations.

For extra credit, you can also implement other more advanced features.

2. **Part II: Modeling Memory and Handling Pointers**

In this part we will extend the translator to handle memory. We will prove the absence of certain memory-errors such as double-free, out-of-bounds writes and access to uninitialized memory. There are many different ways to encode memory with different trade-offs, and we will consider three concrete options:

- Flat memory model. In this model, the complete memory is modeled as a single array M that maps memory addresses to values. This array contains all of memory, including the stack and the heap. Every variable x is stored at some address a_x, and $M[a_x]$ is used to read or write its value. Furthermore, to support user-allocated structures (through `malloc`), a second array S is used that stores the size of heap-allocated data.
- Burstall model. Similar to the flat model, an array is used to map addresses to values. However, instead of using a single array, an array *per type* is used. That is, we will have an array for *char* values, one for *int* values, and so on.

  This makes the memory model more efficient because the SMT solver does not have to reason about possible aliasing of addresses that point to values of different types. However, if (through casting) pointers to different types do alias, then this will be missed bye the Burstall model. Furthermore, it is not possible to reason soundly about union types.
- Partition model (extra credit). Finally, in this model we again use multiple arrays for memory to make SMT queries more efficient. Here, an aliasing analysis is used to decide which pointers belong to which partition.

You can read more about all three memory models in Section 3 of the paper on *Cascade 2.0* available here: https://cs.nyu.edu/~barrett/pubs/WBW14-abstract.html.

(a) Extend your translator with support for pointers and a memory model. Implement both the flat and the Burstall model, and for extra credit, the partition model.

The goal is to prove the absence of memory-related errors. Consider at least the following errors:

- Pointer dereference. Any dereference of a pointer should be guaranteed to be a valid pointer (non-zero and pointing to allocated).
- Free's are valid. Freeing a memory block should check that the block to be freed has been allocated before (no double frees, or frees of memory that has never been allocated).

(b) We provide a set of examples that you should run through your tool: http://web.stanford.edu/class/cs357/handouts/input_memory.txt Some of the examples are free of errors and your tool should correctly prove this fact. Other examples contain memory-related errors, and your tool must fail to verify them.

Feel free to additionally write your own examples, or use programs from elsewhere.

(c) For all examples, report the name of the example, along with whether your tool was able to prove the absence of errors, as well as the time it took to run your tool. Do this for all memory models that you implemented. Comment on differences between the memory models.

(d) Extra credit: If you want to solve extra challenges, feel free to implement additional features. For instance, you could prove the absence of memory leaks, handle more C features, implement another memory model (as mentioned above), try a different SMT solver, or prove the absence of other classes of errors such as division by zero.

If you want to explore the partition memory model, the paper http://cs.nyu.edu/~barrett/pubs/WBW16-abstract.html provides further information.

Submission Process

Electronically submit your assignment by emailing the following to cs357-aut1516-staff@lists.stanford.edu:

1. A written description of what you did and any interesting issues that came up, including any extra challenges you attempted.
2. A table listing examples from the first homework and run times to solve each of the program equivalence problems using each SMT solver that you tried (or "time-out" for cases where the solver runs for more than 5 minutes).
3. A table listing examples, run times and whether the example contains any errors for all of the memory problems. Do this for all memory models you tried.