CS357 - Assignment 1

Due: October 19, 2015, 23:59

This project is designed to be done by individuals. However, you may work in a group of two if you take on some significant "extra challenges" from the "Extra Credit" section below.

In this assignment you will implement a translator from C programs to SAT formulas that can be used to prove equivalence between two C programs.

Instructions

1. Put together a rudimentary C parser that will work for the examples below. We suggest you don't spend too much time on this (the goal of this homework is to explore how to use SAT, not to write a C compiler). The following alternatives should be easy to set up:

   - pycparser (`https://pypi.python.org/pypi/pycparser`) is a Python library that can parse C code into an AST. This is what we provide in the starter package (see below).
   - CIL (`http://kerneis.github.io/cil/`) is written in ML, and is widely used to prototype program analysis applications for C.

   If you are more familiar with another tool, such as LLVM, go ahead and use it. Or just dive into LEX and YACC if you wish.

2. Build a translator that can convert a fragment of C code (parsed using your front-end from the previous step) to a vector of boolean expressions. This will essentially be a symbolic executor. N-bit values should be represented as vectors of N boolean expressions.

   To execute an operation such as "c = a + b", look up the symbolic values of "a" and "b" in the current symbolic state, and do a symbolic computation of "+" that combines them into a vector of boolean expressions (similar to Alex's description of Saturn). Then update the symbolic state to map "c" to this symbolic value.

   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

You will need to unroll loops. All the loops in the examples below should have relatively small upper bounds on the number of iterations.

3. Using the above code, write code to generate a CNF formula expressing the non-equivalence of two fragments of C code. Write this out in DIMACS format, which is handled by many different SAT solvers.

4. Obtain a SAT solver that reads DIMACS format, such as cryptominisat (`http://www.msoos.org/cryptominisat2/`). Also, obtain another solver that takes DIMACS format to compare results (`satlive.org` is a good source of information about SAT solvers).

5. There is a collection of clever code fragments for bit-level computations at Sean Eron Anderson's Bit Twiddling Hacks page (`http://graphics.stanford.edu/~seander/bithacks.html`). It contains several examples of different ways to compute the same function.

   For the following examples from the above page (and any additional ones you care to attempt), prove that these alternative implementations are equivalent, by solving the CNF formula generated using the code in part 3. Where a naive version is missing, you will need to write one.

   - Compute the minimum (min) of two integers without branching vs.
     Computing the minimum using a ternary expression
   - Compute the maximum (max) of two integers without branching vs.
     Computing the minimum using a ternary expression
   - Conditionally set or clear bits without branching vs.
     Setting or clearing bits with branching
   - Counting bits set, Brian Kernighan's way vs.
     Counting bits set in 14, 24, or 32-bit words using 64-bit instructions vs.
     Counting bits set, in parallel
   - Computing parity the naive way vs.
     Compute parity of a byte using 64-bit multiply and modulus division vs.
     Compute parity of word with a multiply vs.
     Compute parity in parallel
   - Swapping values the obvious way (not here). vs.
     Swapping values with subtraction and addition vs.
     Swapping values with XOR
   - Reverse bits the obvious way vs.
     Reverse the bits in a byte with 3 operations (64-bit multiply and modulus division) vs.
     Reverse the bits in a byte with 4 operations (64-bit multiply, no division) vs
     Reverse the bits in a byte with 7 operations (no 64-bit)

   Unless you're going for extra credit, you can skip any implementations that require features from the "Extra Credit" section below. However, only skip that implementation. If an example has only some implementations that use extra credit features, you still have to prove equivalence between all implementations that don't.

Extra Credit

If you wish to try, feel free to take on any additional challenges, such as: integer multiplication, division and modulus (note that most of the examples that use these operations also require that you handle 'long long', i.e. 64-bit, integer values), arrays/lookup tables, pointers, floating point, inter-procedural analysis, word-level optimizations, additional C examples from elsewhere, abstraction/refinement, extensive comparison of SAT solvers, etc.

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 and run times to solve each of the problems using each SAT solver that you tried (or "time-out" for cases where the solver runs for more than 5 minutes).

Starter Code

We have prepared a starter package for you, which contains the code for all of the above examples in a custom format, along with a Python script to parse the code into ASTs, using a bundled copy of pycparser (you don't need a separate installation of pycparser to use it). You can use this code as the starting point for a Python implementation.

You are free (but not required) to use the starter package. If you do, please report any bugs you find to the course staff, so we can fix them for everyone.

The starter package is located here: `http://web.stanford.edu/class/cs357/handouts/starter-pack.tar.gz`.

Additional Remarks

- If you'd like a more complete reference on bit-blasting, Chapter 6 of Strichman and Kroening's "Decision Procedures" book gives a fairly complete treatment (at least for non-floating point operations).
- Be careful to avoid formula blowup. There are two ways this can be a problem:
  - When you translate C programs to a formula (in memory), make sure that subformulas that occur multiple times are represented as a single shared node. Effectively, this means your formula is a DAG (directed, acyclic graph) rather than a tree.

    You can achieve this by either being careful not to copy nodes and instead reusing the same node, or by using a hash map where you check if a particular node already exists before creating it.
  - When you convert your formulas to CNF and dump them out, take care to only dump each sub-expression once. You can achieve this by introducing intermediate variables as necessary to maintain the sharing.

  If you have a function that traverses the DAG, then make sure that function is memoized. Otherwise, you might end up with an exponential amount of work.
- pycparser (which is used by the starter code) does not differentiate between variables with the same name in different scopes. All examples we provide work without handling this problem, but if you attempt to work with code from elsewhere, be aware of this limitation. You can work around it by manually renaming variables, implementing support for scopes in your code or using a different parser.

- For some of the examples, one or more of the implementations is only expected to work correctly for some subset of the input space (mainly for some specific maximum width of the input argument(s)). You will need to properly encode this into your implementation, to avoid spurious counterexamples.

  The version of the examples in the starter package contains assert() statements that specify these constraints on the relevant examples, so you will need to somehow handle this information during your translation.

- We don't care about the final value of every variable in the examples, only the ones designated as outputs. Also, not all variables need to start unconstrained, only the ones designated as inputs. You will need to include this information in your implementation, and use it appropriately.

  The input format used in the starter package allows you to specify the input and output variables for the implementations of each function. The accompanying parser gives you programmatic access to this information.

- Start early! It is very plausible that you will stumble upon unexpected issues early on. Use Piazza for questions or issues.