COMPILER CONSTRUCTION · WEEK 10 · STATION IV

Semantics as a Formula

Rotor: a bit-precise model of RISC-V in BTOR2. A compiler run backwards, into logic.

THE IDEA

A machine step is a function from state to state. A function on bits is a formula.

The machine's state is the pc, 32 registers and memory: bits. One step reads the instruction at the pc and computes the next state: a function on bits. Every function on finitely many bits is a Boolean formula — the adder of the introduction class is a circuit is a formula.

So write the step down: `next(state) = if instruction is addi then … else if ld then … else …` — one big if-then-else over the fourteen instructions, bit-precise, sixty-four bits wide. That is the semantics of RISC-U as an object a solver can read.

THE BRIDGE

The compiler defined C*'s meaning by translating to RISC-U. Rotor defines RISC-U's meaning by translating to logic. Together: the meaning of a C* program as a formula, with the compiler's output as the intermediate. Syntax became semantics, mechanically.

IN THE TERMINAL

Rotor eats a C* program and writes the machine that runs it.

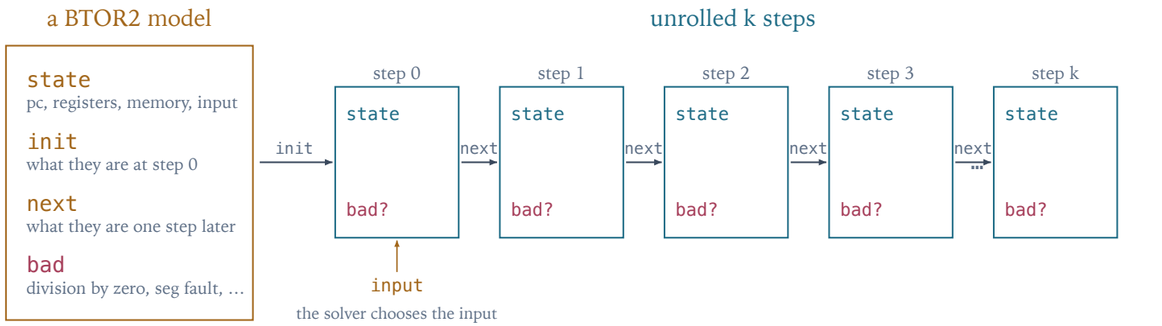
```
$ make rotor
cc … --include selfie.h tools/rotor.c -o rotor
$ ./rotor -c examples/symbolic/division-by-zero-3-35.c
./rotor: 15161 lines of model formulae generated
./rotor: 237537 characters written into examples/symbol
$ ./rotor -c selfie.c - 0
./rotor: … lines of model formulae generated, in about
```

Rotor is built on selfie: it uses starc to compile the program and then generates the model of the machine with that binary loaded. Fifteen thousand lines for fifteen lines of C, because most of the model is the machine, not the program.

Linear in the size of the binary: rotor models all of selfie, forty-three thousand instructions, the same way. And it models gcc's RISC-V binaries too, including compressed instructions, because it models the machine and not the compiler.

BTOR2

Sorts, constants, states, init, next, bad. Every line commented.



satisfiable ⇔ some input reaches a bad state within k steps ⇒ the satisfying assignment is the failing input

unsatisfiable ⇔ no input reaches a bad state within k steps ⇒ a theorem, up to the bound k, and nothing beyond it

size of the unrolled formula: O(k × size of the model) · size of the model: linear in the size of the binary

a model, and what unrolling it means

```
1 sort bitvec 1 ; Boolean
5 sort bitvec 64 ; 64-bit double word
191 state 5 core-0-pc ; program counter
192 init 5 191 190 ; initializing pc
150 state 71 input-buffer ; uninitialized input
20070 next 5 191 20064 ; program counter
36027 bad 36026 core-0-division-by-zero
```

Each line has a number, an operator, a sort, and arguments that are earlier lines: a directed acyclic graph of formulas. `state` declares a variable that persists across steps; `init` fixes it at step 0; `next` says what it is one step later; `bad` names a Boolean that must never be true. The input buffer has no `init`: the solver chooses it.

READING THE MODEL

How `next` for the program counter is built.

Fetch: read the 32-bit word at the pc from the code segment array. Decode: slice out the opcode, funct3, registers, immediate — the decoder of week 5 as slice and concat operations. Then, for each instruction, the new pc: `pc + 4` for most, `pc + imm` for a taken branch or a jump, the register value for `jalr`.

All of it combined by `ite`: if the instruction is beq and the registers are equal then this else that. The same chain for every register and for memory, where a store is a `write` into an array and a load a `read` from it.

COMPARE WITH MIPSTER

The emulator's `execute` is an if-else chain over the same decoded fields, doing the same arithmetic on concrete values. Rotor's `next` is the same chain over symbolic ones. Two transcriptions of one semantics, from one file, and the assignment in week 12 is to use one against the other.

UNROLLING

k copies of the state, chained by `next`: a formula that is satisfiable exactly if a bad state is reachable within k steps.

Make a copy of every state for steps 0 to k. Constrain step 0 by `init`, each step by `next` of the previous, and ask whether any `bad` can be true at any step. Size: k times the model. Satisfying assignment: the input that does it.

That formula goes to a solver next week. Its variables are bits and words and arrays; its size is millions of clauses; and its question is the canonical hard problem.

TWO OTHER USES

Leave the code segment uninitialised as well, and the solver *synthesises* a program that reaches a state. Model two binaries side by side and ask whether their outputs can differ: *equivalence checking*, which is what a compiler test would like to be.

ASSIGNMENT

array-multidimensional.

1. Specify. Declarations `type identifier [ n ] [ m ]` and accesses `a[i][j]`, any number of dimensions.
2. Semantics. Row-major: the address of `a[i][j]` is the base plus 8 · (i · m + j). The symbol table entry must record every dimension. Decide it, write it down.
3. Generate. One multiply and add per dimension, temporaries allocated and freed per index expression. The assertions on temporaries again.
4. `./grader/self.py array-multidimensional`, `self-compile`, and a two-dimensional array somewhere in selfie.c.

KEEP THE TEST PROGRAM

Write a small C* program that exercises your arrays, with an index that can go out of bounds on some input. You will hand it to rotor in two weeks.