Reasoning About WebAssembly Code Using Codewalker

David Hardin
Advanced Technology Center
david.hardin@rockwellcollins.com

Rockwell Collins

Objectives

- Reason about machine code generated from high-level languages
  - Eliminate need to trust compiler frontends by reasoning about compiler intermediate forms
- Exercise the ACL2 theorem prover, and the integrated Codewalker facility, to prove properties of low-level programs
  - Highly automated proof system — minimal user interaction
  - High-speed, executable specifications — can be used for validation testing
  - "Pluggable" Instruction Set definitions
- Learn about WebAssembly and how to prove correctness for WebAssembly programs
  - Motivated by previous work on reasoning about LLVM code using Codewalker (ACL2-15 paper)

WebAssembly

- WebAssembly is a new intermediate form for the Internet, under development by Apple, Google, Microsoft, and Mozilla
  - To be supported on WebKit, Chrome, Edge, and Firefox
- Web site: http://webassembly.org; WebAssembly on github
- PLDI 2017 paper, also available on the WebAssembly github:

Bringing the Web up to Speed with WebAssembly

Andreas Haas Andreas Rossberg Derek L. Schuff* Ben L. Titzer
Google GmbH, Germany / *Google Inc, USA
{ahaas,rossberg,dschuff,titzer}@google.com

Michael Holman
Microsoft Inc, USA
michael.holman@microsoft.com

Dan Gohman Luke Wagner Alon Zakai
Mozilla Inc, USA
{sunfishcode,luke,azakai}@mozilla.com

JF Bastien
Apple Inc, USA
jfbastien@apple.com

Rockwell Collins

WebAssembly (cont'd.)

- Stack-based intermediate, similar to JVM and Microsoft IL
- Emphasis on safe execution, portability, speed of JIT'ed code
- Operational semantics in OCaml
- WebAssembly output by LLVM compiler
- Runnable via Javascript API from browsers
- Output formats include binary, as well as s-expression-based representation
- Some Technical Differences relative to the JVM:
  - Instruction set not Java-centric
    - Not as object- and thread-oriented as the JVM
  - Branches are taken relative to the current lexical block
  - Eliminates instructions such as goto that make bytecode verification more challenging
  - Some differences in the stack manipulation instructions

Example: Iterative Factorial Test Case, from WebAssembly github

```
(func (export "fac-iter") (param i64) (result i64)
  (local i64 i64)
  (set_local 1 (get_local 0))
  (set_local 2 (i64.const 1))
  (block
    (loop
      (if
        (i64.eq (get_local 1) (i64.const 0))
        (br 2)     ;; branch out two levels to last instruction
        (block
          (set_local 2 (i64.mul (get_local 1) (get_local 2)))
          (set_local 1 (i64.sub (get_local 1) (i64.const 1)))))
      (br 0)))     ;; branch to beginning of current block
  (get_local 2))
```

Codewalker

- A new facility as of ACL2 7.0 (January 2015), due to J Moore
- Performs “decompilation into logic” of a machine-code program to a series of “semantic functions” that summarize the program’s effect on machine state
- Works with an instruction set description written in the usual ACL2 “machine interpreter” style, as earlier described
- Produces proofs that the generated semantic functions are correct
- Inspired by Magnus Myreen’s Ph.D. thesis (2008)
  - Myreen’s decompiler utilizes the HOL4 theorem prover

- For more details, see **`books/projects/codewalker`** in the ACL2 distribution

Tweaking WebAssembly S-Expressions for Codewalker

- For a first proof-of-concept use of Codewalker to reason about WebAssembly, wanted a more “assembly-code-like” form
  - Closer to JVM-like M1 in the Codewalker distribution

- Particularly didn’t want to deal with the lexical block branch complication
  - Converted to more conventional branch instruction

- Conversion currently done by hand; could be readily automated

Iterative Factorial Test Case — Slight Tweak

```
;;(func (export "fac-iter") (param i64) (result i64)
;; (local i64 i64)
(get_local 0)     ;; 0
(set_local 1)     ;; 1
(i.const 1)       ;; 2
(set_local 2)     ;; 3
;; (block foo)
;; (loop bar)
(get_local 1)     ;; 4
(i.const 0)       ;; 5
(i.eq)            ;; 6
(jumpt 10)        ;; 7
;; (block baz)
(get_local 1)     ;; 8
(get_local 2)     ;; 9
(i.mul)           ;; 10
(set_local 2)     ;; 11
(get_local 1)     ;; 12
(i.const 1)       ;; 13
(i.sub)           ;; 14
(set_local 1)     ;; 15
;; (end baz)
(jump -12)        ;; 16
;; (end bar)
;; (end foo)
(get_local 2)     ;; 17
(halt)            ;; 18
```

Rockwell Collins

Machine Modeling in ACL2

- We begin by defining a machine state data structure whose components are referenced and/or assigned with each instruction
- Typically, we define machine state elements for the program counter, other fixed-function registers, the register file, data memory, and program memory, aggregating these into a single state variable
    - Register file components and memory locations are usually abstracted as Lisp lists, accessed with **`nth`** and modified with **`update-nth`**
- ACL2 is a purely functional subset of Common Lisp; thus, in order to modify machine state, one must construct a new machine state with the modified components, and return that updated state.
    - For large machine states, this can become expensive (much memory allocation and garbage generation)
- Fortunately, ACL2 also supports *single-threaded objects*, or stobjs, that ameliorate this problem

Machine Interpreter

- A top-level machine interpreter whose state is modelled as a stobj is normally written in ACL2 as follows, where webas is the name of our WebAssembly machine model interpreter:

```
(defun webas(s n)
   (declare (xargs :stobjs (s)))
   (if (zp n)
       s
       (let ((s (step s)))
         (webas s (- n 1)))))
```

- where **`s`** is the machine state, **`(step s)`** is a function that dispatches to an individual instruction function based on the current opcode, and **`zp`** is a standard ACL2 “equals 0” predicate

Instruction Definitions

- Individual instructions are defined as follows:

```
;; Semantics of (I.ADD): increment the pc, pop two items off the
;; arg-stack and push their sum.

(defun execute-I.ADD (inst s)
  (declare (xargs :stobjs (s))
           (ignore inst))
  (let ((u (top (arg-stack s)))
        (v (top (pop (arg-stack s))))
        (arg-stack1 (pop (pop (arg-stack s)))))
   (let* ((s (!arg-stack (push (+ v u) arg-stack1) s))
          (s (!pc (+ 1 (pc s)) s))) s)))
```

- where **(pc s)** returns the value of the program counter stored in the state **s**;
- **(arg-stack s)** returns the argument stack stored in **s**;
- **(!pc v s)** sets the value of the program counter to **v**;
- and **(!arg-stack x s)** sets the argument stack to **x**. These latter two functions update the state **s**.

Proof Results

We were able to prove that the WebAssembly iterative factorial program implements the following non-tail-recursive factorial function:

```
(defun ! (n)
  (if (zp n)
      1
      (* n (! (- n 1)))))
```

Final Correctness Theorem:

```
(defthm reg[2]-of-code-is-!
  (implies (and (hyps s)
                (program1p s)
                (nat-listp (rd :locals s))
                (nat-listp (rd :arg-stack s))
                (equal (rd :pc s) 0))
           (equal (nth 2 (rd :locals (webas s (clk-0 s))))
                  (! (nth 0 (rd :locals s))))))
```

Conclusion

We utilized Codewalker to prove correctness properties about small WebAssembly programs.

No significant results herein; just wanted to learn about WebAssembly and exercise Codewalker on a new instruction set

Verification:

- Codewalker enables automated formal proofs of correctness
- Codewalker provides "pluggable" instruction set definitions
- Verification can occur at the basic block level, thus allowing for incremental progress

Validation:

- ACL2 single-threaded objects allows for reasonably speedy execution of the WebAssembly code interpreter, enabling basic validation testing