Reasoning About LLVM Code Using Codewalker

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

Rockwell Collins

Objectives

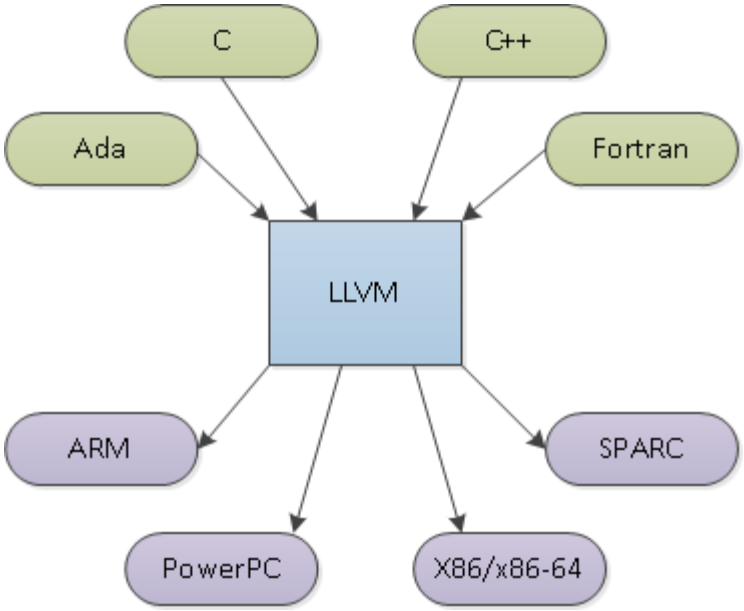
- Reason about machine code generated from high-level languages
  - Eliminate need to trust compiler frontends by reasoning about compiler intermediate forms
- Exercise the ACL2 theorem prover, and the integrated Codewalker facility, to prove properties of binary programs
  - Highly automated proof system — minimal user interaction
  - High-speed, executable specifications — can be used for validation testing
  - "Pluggable" Instruction Set definitions

Motivating Work

- J Moore's Codewalker (released 2015), inspired by Magnus Myreen's "decompilation into logic" work
- Magnus Myreen's Decompilation into Logic (2008 - present)
  - Imperative machine code (PPC, x86, ARM) -> HOL4
  - Extracts functional behavior of imperative code
  - Assures decompilation process is sound
- Andrew Appel (Princeton) observed that "SSA is functional programming" (1998)
- Our previous work to create an LLVM-to-ACL2 translator (ACL2-14 paper)
  - An untrusted translator, written in OCaml
  - Can we use Codewalker to produce a higher-fidelity reasoning environment for LLVM code?

LLVM

- LLVM is the intermediate form for many common compilers, including clang
- LLVM code generation targets exist for a variety of machines
- LLVM is a register-based intermediate in Static Single Assignment (SSA) form (each variable is assigned exactly once, statically)



Example — C Source

```
unsigned long occurrences(unsigned long val, unsigned int n,
                          unsigned long *array) {
  unsigned long num_occur = 0;
  unsigned int j = 0;
  for (j = 0; j < n; j++) {
    if (array[j] == val) num_occur++;
  }
  return num_occur;
}
```

- We can produce LLVM from C source (for LLVM 3.6.0) as follows:

```
clang -O1 -S -emit-llvm occurrences.c
```

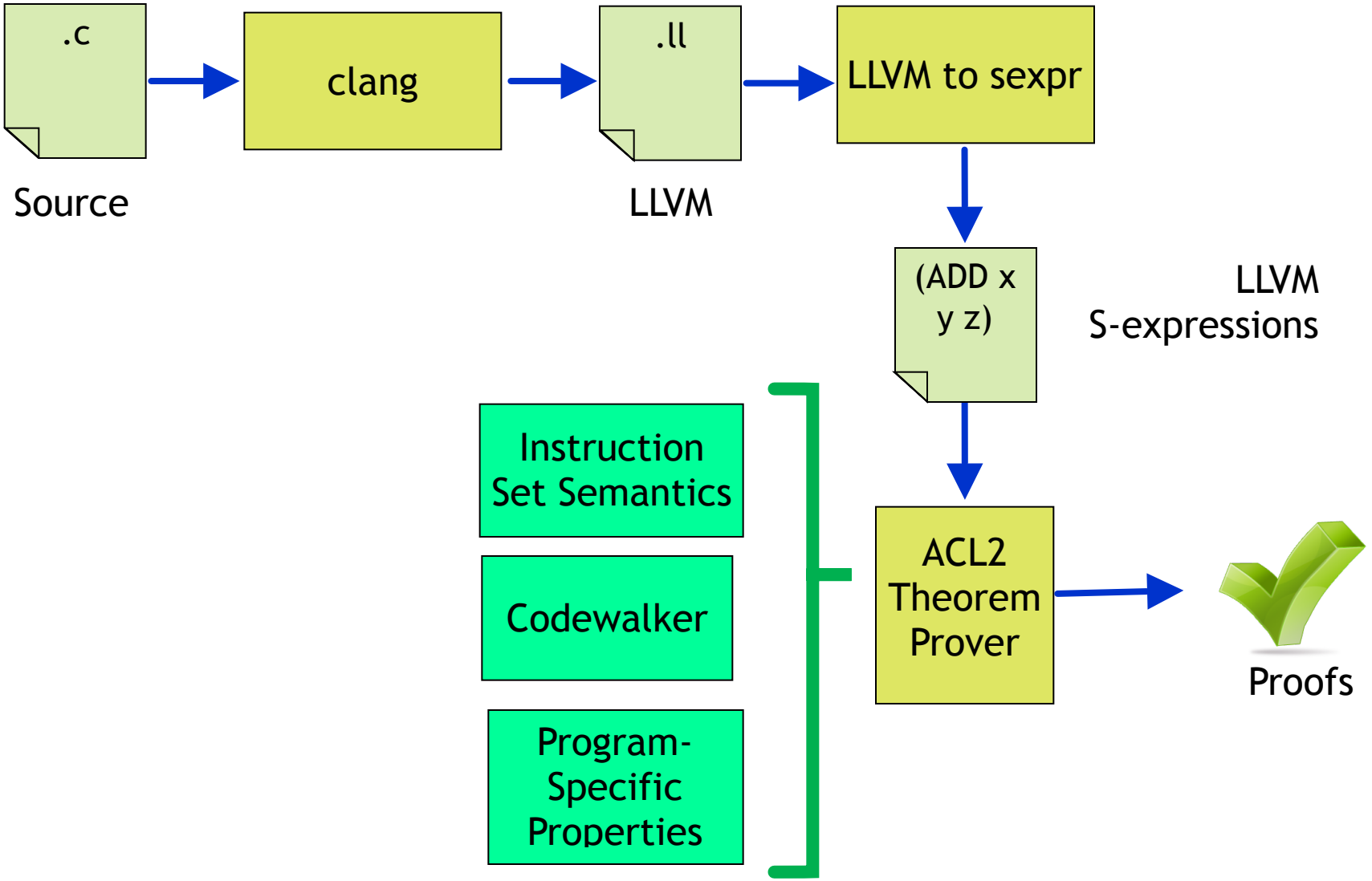
Example: Generated LLVM from clang

```
  define i64 @occurrences(i64 %val, i32 %n, i64* %array) {
  %1 = icmp eq i32 %n, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:
  %indvars.iv = phi i64 [ %indvars.iv.next, %.lr.ph ], [ 0, %0 ]
  %num_occur.01 = phi i64 [ %.num_occur.0, %.lr.ph ], [ 0, %0 ]
  %2 = getelementptr inbounds i64* %array, i64 %indvars.iv
  %3 = load i64* %2, align 8, !tbaa !1
  %4 = icmp eq i64 %3, %val
  %5 = zext i1 %4 to i64
  %.num_occur.0 = add i64 %5, %num_occur.01
  %indvars.iv.next = add i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %n
  br i1 %exitcond, label %._crit_edge, label %.lr.ph

._crit_edge:
  %num_occur.0.lcssa = phi i64 [ 0, %0 ], [ %.num_occur.0, %.lr.ph ]
  ret i64 %num_occur.0.lcssa
}
```

Toolchain Architecture



Converting LLVM Assembly to ACL2 S-Expressions

- Conversion needed so that LLVM programs can both be reasoned about, as well as executed within the ACL2 environment
- Conversion currently done by hand; could be automated by using some of the previous translator work
- Fixed-width type information ignored; all computations performed on arbitrary-precision data
  - Didn't want to tackle both Codewalker *and* modular arithmetic complexity at the same time
- No "syntactic sugar" to distinguish register numbers from numeric constants; use added (CONST n) instruction and a simple stack, featuring a (POPTO r) operator, to introduce numeric constants

Converting LLVM to ACL2 (cont'd.)

- Phi instructions at the beginning of blocks declare what values certain registers should have, based on the identity of the previous executing block
  - Think of phi instructions as "comes from" information
- Needed to make the declarative phi instructions operational so we could successfully intepret LLVM programs in ACL2
- Decided to introduce register-to-register transfer capability via the previously added stack, using a (PUSH r1)/(POPTO r2) pair
- We place these register-to-register transfers at the end of blocks that branch to other blocks containing phi instructions
- Using the stack instructions makes it clear that these instructions were added to the original LLVM code
- Many other possible solutions

Example LLVM Code in S-Expression Form

```
;;    reg[2] contains val
;;    reg[1] contains n
;;    reg[0] contains array base address
<PREAMBLE CODE ELIDED>
;; .lr.ph:                                          ; preds = %0, %.lr.ph
   (GETELPTR 7 0 5)     ; 8   reg[7] <- mem address of arr[index]
   (LOAD 8 7)           ; 9   reg[8] <- mem[reg[7]] = arr[index]
   (EQ 9 8 2)           ; 10  reg[8] == val?
   (ADD 10 6 9)         ; 11
   (CONST 1)            ; 12
   (POPTO 11)           ; 13
   (ADD 12 5 11)        ; 14
   (EQ 13 12 1)         ; 15  reg[13] <- reg[12] == n?
   (PUSH 12)            ; 16
   (POPTO 5)            ; 17  phi(j), j <- j+1
   (PUSH 10)            ; 18
   (POPTO 6)            ; 19  phi(num_occur)
   (BR 13 1 -12)        ; 20  loop back to .lr.ph if j+1 < n
;;._crit_edge:                                      ; preds = %.lr.ph, %0
   (PUSH 6)             ; 21  push num_occur on stack
   (HALT)               ; 22
```

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

ACL2 Single-Threaded Objects (stobjs)

- ACL2 enforces strict syntactic rules on stobjs to ensure that "old" states of a stobj are guaranteed not to exist
  - This means that ACL2 can provide destructive implementation for stobjs, allowing stobj operations to execute quickly
- An ACL2 single-threaded object thus combines:
  - A functional semantics about which we can reason
  - A relatively high-speed implementation that we can use for model validation, i.e. by executing tests that run on the machine being modelled
- stobjs introduce some complications for reasoning, but the ACL2 community has been diligently working on mitigating these issues

Machine Interpreter

- A top-level machine interpreter whose state is modelled as a stobj is normally written in ACL2 as follows, where LL2 is the name of our LLVM subset:

```
(defun LL2(s n)
  (declare (xargs :stobjs (s)))
  (if (zp n)
      s
      (let ((s (step s)))
        (LL2 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 (ADD a b c): increment the pc, and set the value
;; of the first local to the sum of the second and third locals.

(defun execute-ADD (inst s)
  (declare (xargs :stobjs (s)))
  (let* ((s (!loi (arg1 inst)
                  (+ (loi (arg2 inst) s) (loi (arg3 inst) s)) s))
         (s (!pc (+ 1 (pc s)) s)))
    s))
```

- where `(pc s)` returns the value of the program counter stored in the state `s`, `(loi n s)` returns the value of the nth local variable (or register — both terms are used) stored in s, and `(arg<i> inst)` returns the value of the ith operand of the current instruction;
- `(!pc v s)` sets the value of the program counter to `v`, and `(!loi j x s)` sets the value of the jth local to `x`. These latter two functions update the state `s`.

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
- Three main Codewalker API's utilized in our work:
  - **`def-model-api`**
  - **`def-semantics`**
  - **`def-projection`**

def-model-api

- **`def-model-api`** instructs Codewalker on the basics of the machine model:
  - The name of the machine interpreter function
  - The name of the state variable, and whether it is a stobj
  - How to access and update the registers
  - Basic machine types
  - How to access and update the program counter
  - Subroutine linkage (not yet implemented)

def-semantics

- **`def-semantics`** is the main workhorse of the Codewalker facility. It instructs Codewalker to create "semantic functions" for given regions of machine code.
- Semantic functions summarize the actions of given machine code segments on machine state
  - Semantic functions are generated by symbolic simulation of the machine previously described to **`def-model-api`**.
  - Importantly, the machine interpreter function is not mentioned in the generated semantic functions
- **`def-semantics`** also generates a 'clock' function that prescribes the number of instruction steps that the interpreter must execute in order to cover the user-specified range of program counters
- Finally, **`def-semantics`** generates a theorem that the semantic function correctly summarizes the changes to the machine state produced by executing the machine interpreter for the number of steps indicated by the generated clock function.

def-semantics Example

```
(def-semantics
  :init-pc 8
  :focus-regionp (lambda (pc) (and (<= 8 pc) (<= pc 9)))
  :root-name frag
  :hyps+ ((occurrences-programp s) (program-inv s)
          (<= (+ (nth 0 (rd :locals s)) (nth 1 (rd :locals s))) (len (rd :memory s)))))
```

This generates a semantic function, a clock function, and a correctness theorem for the code segment of occurrences-program between PC=8 and PC=9, inclusive:

```
(DEFUN SEM-FRAG-8 (S)
  (DECLARE (XARGS :NON-EXECUTABLE T :MODE :LOGIC))
  (DECLARE (XARGS :STOBJS (S)))
  (PROG2$ (ACL2::THROW-NONEXEC-ERROR 'SEM-FRAG-8 (LIST S))
   (IF (AND (HYPS S) (OCCURRENCES-PROGRAMP S)(PROGRAM-INV S)
            (<= (+ (NTH 0 (RD :LOCALS S)) (NTH 1 (RD :LOCALS S))) (LEN (RD :MEMORY S))))
       (WR :PC 10
           (WR :LOCALS
               (UPDATE-NTH 7 (+ (NTH 0 (RD :LOCALS S)) (NTH 5 (RD :LOCALS S)))
                           (UPDATE-NTH 8 (NTH (+ (NTH 0 (RD :LOCALS S)) (NTH 5 (RD :LOCALS S)))
                                             (RD :MEMORY S))
                                       (RD :LOCALS S)))
               S))
     S)))
```

def-semantics Example (cont'd.)

Generated Clock Function:

```
(DEFUN CLK-FRAG-8 (S) […]
  (IF (AND (HYPS S) (OCCURRENCES-PROGRAMP S) (PROGRAM-INV S)
           (<= (+ (NTH 0 (RD :LOCALS S)) (NTH 1 (RD :LOCALS S))) (LEN (RD :MEMORY S))))
      2 0)))
```

Generated Correctness Theorem:

```
(DEFTHM SEM-FRAG-8—CORRECT
  (IMPLIES
    (AND (HYPS S)
         (OCCURRENCES-PROGRAMP S)
         (PROGRAM-INV S)
         (<= (+ (NTH 0 (RD :LOCALS S)) (NTH 1 (RD :LOCALS S))) (LEN (RD :MEMORY S))
         (EQUAL (RD :PC S) 8))
    (EQUAL (LL2 S (CLK-FRAG-8 S))
           (SEM-FRAG-8 S))))
```

where LL2 is the machine interpreter for a subset of LLVM that we previously defined.

Note that, since semantic functions accept a single machine state parameter, and return machine state, individual semantic functions may be joined by simple functional composition to produce an overall semantic function for an entire subroutine

def-semantics, cont'd.

- The example just given is very simple, in the interest of presenting the output of def-semantics on single slides
- For the occurrences program, we invoke def-semantics twice:
  - once to generate a semantic function for the preamble code before the loop
  - once for the loop and postlude code
- We then compose these two generated semantic functions to produce a semantic function for the entire occurrences program.

def-projection

- **`def-semantics`** eliminates the machine interpreter from the description of what a segment of machine code accomplishes, making that segment of code easier to reason about
- **`def-projection`** takes a semantic function generated by def-semantics, and allows us to 'project out' a function that computes the final value of some machine state component, thus eliminating any explicit mention of the machine state from the function and making reasoning even easier
  - **`def-projection`** only works for computations that don't depend on state
  - Thus, we didn't use def-projection for the occurrences program
- If the projection is successful, **`def-projection`** produces a theorem stating that the output of the generated projection function, given appropriate machine state component inputs, is equal to the projected component of the machine state returned by the semantic function

Results: Proof

We were able to prove that the LLVM occurrences program implemented a list-based non-tail-recursive occurlist function, using techniques described in an ACL2-13 paper:

```
(defun occurlist (val lst)
  (declare (xargs :guard (and (integerp val) (integer-listp lst))))
  (if (endp lst) 0
    (+ (if (= val (car lst)) 1 0)
       (occurlist val (cdr lst)))))
```

Final Correctness Theorem:

```
(defthm l12-running-occurrences-code-=-occurlist
  (implies
   (and (hyps s) (program-inv s) (occurrences-programp s)
        (<= (+ (nth 0 (rd :locals s)) (nth 1 (rd :locals s))) (len (rd :memory s)))
        (= (nth 1 (rd :locals s)) (len (rd :memory s)))
        (equal (rd :pc s) 0))
   (= (nth 6 (rd :locals (l12 (l12 s (clk-preamble-0 s))
                              (clk-loop-8 (l12 s (clk-preamble-0 s))))))
      (occurlist (nth 2 (rd :locals s)) (rd :memory s)))) [hints elided])
```

We also proved similar correctness results for a number of other small C programs.

Results: Execution

- We are able to execute LLVM programs on concrete inputs utilizing the LL2 interpreter
- Since the LL2 state is a stobj:
  - State updates are done in-place, and thus are inexpensive
  - it is easy to set up an initial state using a sequence of state-modifying commands, such as `!memi`
- LL2 is tail-recursive, so it won't blow up the stack
- Measured performance on a 2012 MacBook Pro: 226,000 LLVM instructions per second
  - One-tenth the performance of our previous translation approach, which generated Lisp functions for each LLVM subroutine
  - Performance is adequate for basic validation testing

Codewalker Issues

- Codewalker does not help with the creation of loop invariants; these must be provided manually
- One cannot yet instruct Codewalker to generate relocatable semantic functions
- Codewalker does not yet handle subroutine linkages; this is future work
- We have been unable to get Codewalker to successfully process a code region with nested loops; this work is ongoing
  - Myreen's system also has difficulty with nested loops
- Codewalker is still a bit "touchy"; it is easy to overlook some necessary input predicate that will cause Codewalker to fail
  - Being an early release, Codewalker's failure messages are currently not very helpful
- We have not yet employed Codewalker on a machine model with fixed-width registers; the added complexity of modular arithmetic may present difficulties
- ***Issues aside, Codewalker is a very promising new capability for the experienced ACL2 user!***

Conclusion

We successfully utilized Codewalker to prove key properties about small LLVM programs, generated from C source by the clang compiler.

Verification:

- Codewalker enables highly automated formal proofs of correctness for LLVM programs
- 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 LLVM code interpreter, enabling basic validation testing.