Development of a Translator from LLVM to ACL2

David Hardin, Jennifer Davis, David Greve, and Jedidiah McClurg

July 2014
Introduction

- **Research objectives:**
  - Reason about machine code generated from high-level languages
    - Eliminate need to trust compiler frontends by reasoning about a compiler intermediate representation
  - Reason about small fragments of assembly code

- **We wish to create a library of formally verified software component models for layered assurance.**

- **In our current work, the components are in LLVM intermediate form.**

- **We wish to translate them to a theorem proving language, such as ACL2.**

- **Thus, we built an LLVM-to-ACL2 translator.**
Motivating Work

- Jianzhou Zhao (U Penn) et al. produced several different formalizations of operational semantics for LLVM in Coq. (2012)
  - Intention to produce a verified LLVM compiler

- Magnus Myreen’s (Cambridge) “decompilation into logic” work (2009 - 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)
  - Inspired us to build a translator from LLVM to ACL2
LLLVM

- 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).
- An LLVM program consists of a list of entities:
  - There are eight types including:
    - function declarations
    - function definitions
- Our software component models are created from code that has been compiled into the LLVM intermediate form.
ACL2

- A Computational Logic for Applicative Common Lisp (ACL2)
- Highly automated theorem proving system
- Functional language with admission criteria
- Executable subset of language
- Rich set of legal identifiers (@foo, %.06, @bar_%0)
- Side-effect free subset of Lisp, so it inherits Lisp peculiarities
  - Function definition: (defun funname (parm1 parm2 parm3) (<body>))
  - Function invocation: (funname x y z)
  - let binds variables to values within a function body
  - Multiway conditionals use the cond form
  - Lists are the fundamental data structure
  - ACL2 supports integers and rationals
  - Lisp predicate names are traditionally given a suffix of “p”
LLVM-to-ACL2 Translation Toolchain

User Interaction

C/C++ file

clang

LLVM "ll" file

Parser

ACL2 theorem prover

Translator

ACL2 code

LLVM abstract syntax tree

Proofs
Example — C Source

```c
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 3.2) as follows:

```
clang -O4 -S -emit-llvm occurrences.c
```
Example — LLVM

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
}
Translation Snippet

- Each block within an LLVM function contains a list of instructions in SSA form with type information.
- Hence we can readily convert a list of instructions into an appropriate `let*` construct.

```llvm
%2 = getelementptr inbounds i64* %array, i64 %indvars.iv
%3 = load i64* %2, align 8, !tbaa !1
%4 = icmp eq i64 %3, %val

(let*
 ( (%2 (+ %array (_30_gep (list %indvars_dot_iv)))))
 ( (%3 (i64_frombytes (loadbytes *i64_size* %2 st))))
 ( (%4 (icmp= %3 %val)))
 ...))
```
Main Translator Algorithm

Translator

Get Function Names → Remove Aliases → Promote Blocks to Functions → Translate to ACL2

- LLVM AST
- ACL2 Code
Remove Aliases

- Aliases allow new names (e.g., @foo1 and @foo2) to be used for globals and functions.

```assembly
@bar = global i32 123
@foo1 = alias i32* @bar
@foo2 = alias i32* @bar
```

- We eliminate these.
Main Translator Algorithm

Translator

Get Function Names → Remove Aliases → Promote Blocks to Functions → Translate to ACL2

LLVM AST → ACL2 Code
Promote Blocks to Functions

- As we have seen, LLVM functions often contain inner blocks and branch instructions.

- Each of these blocks is pulled out as a new function.

- For each block, the phi instructions denote variables that become parameters for that new function.

\[
%.06 = \text{phi} \ i64 \ [ \ %4, \ %.lr.ph \ ], \ [ \ %\text{sum}, \ %0 \ ]
\]

- The phi instructions also tell us the parameter values that must be used at the new function’s call site(s).
Dealing with Order of Declarations

- ACL2 requires functions and constants to be defined before they are used.

- We do a topological sort on each of the call/dependency graphs.

Figure: Topological Sort on the Call Graph
Main Translator Algorithm

Translator

Get Function Names → Remove Aliases → Promote Blocks to Functions → Translate to ACL2

LLVM AST → ACL2 Code
Translate to ACL2

- Function declaration → ACL2 function stub

- Function definition with instruction list → def::un construct with a nested let*-bound expression
  - def::un improves upon defun by providing input and output “type” signatures

- Memory → an association list of address-value pairs, where values are 8 bits wide. Zero values are not explicitly stored.

- State is represented using a defstructure form.

- Floating-point number → corresponding rational number, e.g.:
  \[ 1.234 \rightarrow (+ 1 (/2 10) (/3 100) (/4 1000)) \]
**Significant Translator Limitations**

- Exceptions are not supported (not used by C code).
- Indirect call instructions are not supported.
- Variable-length argument lists are not supported.
Translating Loops

- LLVM loops are translated into recursive functions in ACL2.

- Automatically generating measures to admit these recursive functions is, in general, implausible.

- We utilize `def::ung`, found in `coi/defung/defung.lisp`, to admit the function without the need of a measure.
  - The `def::ung` macro generates a companion domain predicate which, when true, ensures that the function terminates.
  - We thus exchange the need to identify a measure for the need to reason about the function’s domain predicate during subsequent proofs.

- We also isolate the potentially complex main body of these functions from the recursive call, producing a non-recursive “step” function that is called at each recursion.
Loop body Example — ACL2

(def::un occurrences_step_0 (done %num_occur_dot_01 %indvars_dot_iv
    %array %n %val st)
  (declare (xargs :signature ((natp i64_p i64_p _30_p i32_p i64_p stp)
    natp i64_p i64_p _30_p i32_p i64_p stp)))

  (let*
    ((%2 (+ %array (_30_gep (list %indvars_dot_iv))))
     (%3 (i64_frombytes (loadbytes *i64_size* %2 st)))
     (%4 (icmp= %3 %val))
     (%5 (zext %4 1 64))
     (%_dot_num_occur_dot_0
      (bits (+ %5 %num_occur_dot_01) 63 0))
     (%indvars_dot_iv_dot_next
      (bits (+ %indvars_dot_iv (bits 1 63 0)) 63 0))
     (%lftr_dot_wideiv (bits %indvars_dot_iv_dot_next 31 0))
     (%exitcond (icmp= %lftr_dot_wideiv %n))
     (mvlist %exitcond %_dot_num_occur_dot_0 %indvars_dot_iv_dot_next
        %array %n %val st))))
Concrete Execution

• Since the translator produces a set of executable ACL2 functions, we can validate the translated code by running it against concrete inputs, e.g.:

```lisp
(def::un occurrences-test1 ()
  (declare (xargs :signature (() natp)))
  (let*
    ((myst (st 0 #xffff0000 #xffff0000 nil))
     (myst (update-mem
            (wr.n 8 #x8038 399
             (wr.n 8 #x8030 234 ...
             (wr.n 8 #x8000 (1- (expt 2 64)) (mem myst)...) myst))))
    (retval (occurrences 399 8 #x8000 myst))))
```

• Performance on benchmarks similar to the above is approximately 2.4 million LLVM instructions per second on an ordinary laptop computer.
Reasoning about Translated Functions

- In order to reason about translated functions in ACL2, we need to first perform abstraction on the data types.
  - For our example, we need to abstract the input array to a Lisp list:

```lisp
(def::ung liftlist (done j array n st)
  (declare (xargs :signature ((natp natp natp natp natp stp)
                               nat-listp)))
  (if (= done 1) nil
      (let* ((ptr (+ array (* j 8)))
              (val (wfrombytes 8 (loadbytes 8 ptr st)))
              (j (bits (1+ j) 63 0))
              (done (if (= (bits j 31 0) n) 1 0)))
    (cons val (liftlist done j array n st)))))
```

- This “lift” function is written in a way that closely reflects the behavior of the underlying LLVM implementation.
Reasoning about Translated Functions (cont’d.)

• The list-based specification of occurrences is fairly conventional:

(def::un occurlist (val list)
  (declare (xargs :signature ((natp nat-listp) natp)))
  (if (endp list)
      0
      (+ (if (= val (car list)) 1 0)
          (occurlist val (cdr list)))))

• We now wish to prove that the translated occurrences function operating over an array in memory produces a result equal to the occurlist function operating over a (lifted) list.
Reasoning about Translated Functions (cont’d.)

- The final equivalence theorem is stated as follows:

```lisp
(def::un occurrences_spec (val n array st)
    (declare (xargs :signature ((natp natp natp stp) natp)))
    (if (zp n) 0
        (bits (occurlist val (liftlist 0 0 array n st)) 63 0)))
```

```lisp
(defthm occurrences_equiv–thm
    (implies (and (stp st) (natp n) (natp array) (bvecp val 64))
        (equal (retval (occurrences val n array st))
            (occurrences_spec val n array st))))
```

- This proof requires us to reason about the domain predicates introduced by `def::ung`, among other complications. See the paper for details.
Future Work

- Stack analysis and data structure analysis
- LLVM DataLayout directives (global endianness, alignment/padding specification)
- LLVM intrinsic functions
- Variable-length argument lists
- Attempt to eliminate cycles in the call graph by code rewrites when possible (rather than just blindly emitting mutual-recursion)
Conclusion

• We built an LLVM-to-ACL2 translator
  – Successfully parses all 5000+ test cases in the LLVM regression suite
  – Produces executable, tail-recursive ACL2 specifications
  – Provides relatively high-speed concrete execution
  – Enables proofs of correctness about LLVM programs

• Our growing regression test suite contains examples with global variables, pointers, arrays, and strings.

• Active development on the translator continues in a number of areas to improve its completeness and robustness, as well as improve our ability to reason about LLVM programs.