Creating Formally Verified Components for Layered Assurance with an LLVM to ACL2 Translator[*]

David S. Hardin[†]
Advanced Technology Center
Rockwell Collins
Cedar Rapids, IA, USA
dshardin@rockwellcollins.com

Jedidiah R. McClurg[‡]
Department of Computer Science
University of Colorado
Boulder, CO, USA
jedidiah.mcclurg@colorado.edu

Jennifer A. Davis
Advanced Technology Center
Rockwell Collins
Cedar Rapids, IA, USA
jadavis4@rockwellcollins.com

ABSTRACT

In our current work, we need to create a library of formally verified software component models from code that has been compiled (or decompiled) using the Low-Level Virtual Machine (LLVM) intermediate form; these components, in turn, are to be assembled into subsystems whose top-level assurance relies on the assurance of the individual components. Thus, we have undertaken a project to build a translator from LLVM to the applicative subset of Common Lisp accepted by the ACL2 theorem prover. Our translator produces executable ACL2 specifications featuring tail recursion, as well as in-place updates via ACL2's single-threaded object (stobj) mechanism. This allows us to efficiently support validation of our models by executing production tests for the original artifacts against those models. Unfortunately, features that make a formal model executable are often at odds with efficient reasoning. Thus, we also present a technique for reasoning about tail-recursive ACL2 functions that execute in-place, utilizing a formally proven "bridge" to primitive-recursive versions of those functions operating on lists.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*

General Terms

Languages, Verification

[*]Approved for Public Release, Distribution Unlimited

[†]The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[‡]The work described herein was performed during a co-op session at Rockwell Collins.

.

Keywords

Formal verification, Theorem proving, ACL2, LLVM

1. INTRODUCTION

> *"Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful."* [2] – George Box, British Statistician

Layered assurance for software often requires the creation of a library of assured software component models, starting with code that lacks a formal pedigree. These assured components can then be assembled into subsystems whose top-level assurance relies on the assurance of the individual components. In our current work, we need to create such a library of formally verified software component models from code that has been compiled (or decompiled) using the Low-Level Virtual Machine (LLVM) intermediate form [14]. Thus, we have undertaken a project to build a translator from LLVM to the applicative subset of Common Lisp [11] accepted by the ACL2 theorem prover [9], and perform verification of the component model using ACL2's automated reasoning capabilities.

Formal verification cannot proceed without a model of the artifact to be analyzed, but as George Box notes above, all models are necessarily approximations. The "trick" with modelling, then, is to capture the essence of the artifact under analysis, at least with respect to the properties that one wishes to verify.

But, how do we assure ourselves that our formal models have sufficient fidelity to "the real world" in order for our analyses to be valid? One significant step is to automate the translation of the artifact from its "native" form to a formal specification. This minimizes the possibility of human error, and, in our experience, makes one think carefully about what details of the source artifact are most important to capture. Another, complementary, way to increase confidence is to produce an executable formal model, and validate it by running production tests for the original artifact on that model. Unfortunately, features that make a formal model executable (tail recursion, in-place state updates) often make reasoning difficult.

In the present work, we are particularly concerned with establishing functional correctness properties for code that has been compiled (or decompiled) using LLVM. LLVM is the intermediate form for many common compilers, including

the clang compiler used by Mac OS X and iOS developers. LLVM supports a number of language frontends, and LLVM code generation targets exist for a wide variety of machines, including both CPUs and GPUs.

LLVM is a register-based intermediate in Static Single Assignment (SSA) form [5]. As such, LLVM supports any number of registers, each of which is only assigned once, statically (dynamically, of course, a given register can be assigned any number of times). Appel has observed that "SSA form is a kind of functional programming" [1]; this observation, in turn, inspired us to build a translator from LLVM to the applicative subset of Common Lisp accepted by the ACL2 theorem prover. Our translator produces an executable ACL2 specification that is able to efficiently support validation via testing, as the generated ACL2 code features tail recursion, as well as in-place updates via ACL2's single-threaded object (stobj) mechanism. In order to ease the process of proving properties about these translated functions, we have also developed a technique for reasoning about tail-recursive ACL2 functions that execute in-place, utilizing a formally proven "bridge" to primitive-recursive versions of those functions operating on lists; this technique is discussed in Section 4.

2. THE ACL2 SYSTEM

We utilize the ACL2 theorem proving system for much of our verification work, as it best presents a single model for formal analysis and simulation. ACL2 provides a highly automated theorem proving environment for machine-checked formal analysis. An additional feature of ACL2, single-threaded objects, adds to its strength as a vehicle for reasoning about fixed-size data structures, as will be detailed in a future section.

ACL2 source code inherits a number of Lisp peculiarities, and so we offer a brief tutorial so that the reader may be able to better read the ACL2 code that appears in this paper. Lisp is a prefix language; the operator always appears before the operands. Lisp defines a function with `defun`, e.g.

```
(defun funname (parm1 parm2 parm3)
  (<body>))
```

and function invocation proceeds as `(funname x y z)`. Multiway conditionals use the `cond` form. `let` binds variables to values within a function body. Many Lisp functions operate on lists, the fundamental Lisp data structure; the most basic of these are obscurely named `car` (first element of a list), and `cdr` (rest of the list, not including the first element). `nth` returns the nth element of a list; `update-nth` returns a new list with the nth element replaced. `take` returns the first n elements of a list; `nthcdr` returns a list containing all but the first n elements. `nil` designates the empty list, and also is boolean "false" (`t` is true). Lisp predicate names are traditionally given a suffix of "p"; thus, `endp` is a function that returns `t` if the end of a list has been reached (and returns `nil` otherwise).

3. TOOLCHAIN OVERVIEW

Our translation toolchain architecture is shown in Figure 1. The left side of the figure depicts a typical compiler frontend producing LLVM intermediate code. LLVM output can be produced either as a binary "bitcode" (.bc) file, or as text (.ll file). We chose to parse the text form, producing an abstract syntax tree (AST) representation of the LLVM program. Our translator then converts the AST to ACL2 source. The ACL2 source file can then be admitted into an ACL2 session, along with conjectures that one wishes to prove about the code, which ACL2 processes mostly automatically. In addition to proving theorems about the translated LLVM code, ACL2 can also be used to execute test vectors at reasonable speed.

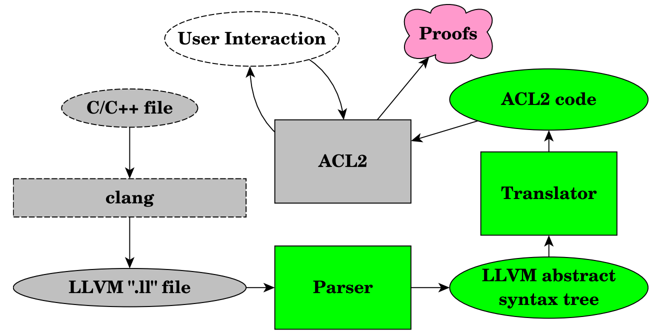

Figure 1: LLVM-to-ACL2 translation toolchain.

The translator is written in OCaml [6], and successfully parses all 5000+ legal `.ll` files in the LLVM source distribution. The translator produces the AST from the input, removes aliases, extracts functions from labelled basic blocks, constructs parameter lists, determines declaration order, then generates the ACL2 code for each function.

3.1 An Example

As an example, consider the following C source code that computes the sum of the first n elements of an array, plus an initial value:

```
long sumarr(unsigned int n, long sum, long *array) {
  unsigned int j = 0;
  for (j = 0; j < n; j++) {
    sum += array[j];
  }
  return sum;
}
```

This is admittedly a very simple example, and fails to exercise many of the more advanced features of our translator and proof framework. However, its relative simplicity allows us to narrate a complete translation and verification within the confines of this paper.

We produce the LLVM code for this function by invoking `clang` as follows: `clang -O4 -S -emit-llvm sumarr.c`. The generated LLVM code for clang version 4.2 (which supports LLVM 3.2) is excerpted below:

```
define i64 @sumarr(i32 %n, i64 %sum, 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 ]
  %.01 = phi i64 [ %4, %.lr.ph ], [ %sum, %0 ]
  %2 = getelementptr i64* %array, i64 %indvars.iv
```

```
  %3 = load i64* %2, align 8, !tbaa !0
  %4 = add nsw i64 %3, %.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:
  %.0.lcssa = phi i64 [ %sum, %0 ], [ %4, %.lr.ph ]
  ret i64 %.0.lcssa
}
```

Observe that LLVM output is similar to assembly code, with labels and low-level opcodes like `br` (branch), `icmp` (integer compare) and `load` (load from memory). Registers are prepended with the "%" character, and are given sometimes-meaningful names. Consistent with the SSA philosophy, no register appears on the left hand side of an assignment ("=") more than once. A peculiar feature of LLVM code is the `phi` instruction, which provides register renaming at a branch target. We will use the `phi` in our ACL2 translation to match formal to actual parameters, as will be detailed later.

Keeping in mind that SSA is functional programming, we can begin to translate this LLVM output to ACL2. First, each label becomes its own function, so we produce `defun` forms for `@sumarr`, `.lr.ph`, and `._crit_edge`. We further observe that the latter function is a trivial leaf function that can be inlined into its callers. The formal parameters for the remaining functions can be determined by consulting the left hand side of the `phi` functions; thus, `.lr.ph` includes `indvars.iv` and `%.01` in its parameter list. We also need to identify parameters that are read, but not modified — `%n` and `%array`. Thus, our developing loop function can be defined as `(defun .lr.ph (%.01 %.indvars.iv %n %array)...)`.

We are left, then, with the question of how to translate memory and memory transactions. Typically in ACL2, a machine state data structure is declared, and passed as a parameter to all functions that read and/or write elements of the state. If a given function updates the state, the modified state must be returned. Obviously, for a large state, functional update of the state can become quite expensive. Thus, the ACL2 developers have created a special kind of data structure that maintains functional semantics, but is implemented using in-place update operations "under the hood".

3.2 ACL2 Single-Threaded Objects

ACL2 enforces restrictions on the declaration and use of specially-declared structures called single-threaded objects, or stobjs [3]. From the perspective of the ACL2 logic, a stobj is just an ordinary ACL2 object, and can be reasoned about in the usual way. Ordinary ACL2 functions are employed to "access" and "update" stobj fields (defined in terms of the list operators `nth` and `update-nth`). However, ACL2 enforces strict syntactic rules on stobjs to ensure that "old" states of a stobj are guaranteed not to exist. This property means that ACL2 can provide destructive implementation for stobjs, allowing stobj operations to execute quickly. In short, an ACL2 single-threaded object combines a functional semantics about which we can readily reason, utilizing ACL2's powerful heuristics, with a relatively high-speed imperative implementation that more closely follows "normal" programming practice.

3.3 Completing the Translation

Our translator emits ACL2 code defining a single-threaded object for memory, declared as a Lisp array of unsigned bytes. It additionally defines ACL2 functions to load and store 8, 16, 32, and 64 byte quantities, both signed and unsigned, and supports both little-endian and big-endian encoding. The translator ensures that this state stobj, `st`, is passed to all translated functions. Since the `sumarr` example does not modify memory, `st` does not need to be returned. The translated functions for `sumarr` are depicted in Figure 2. Referring to the figure, `@sumarr_0` is a "driver" function that does some initial parameter checking before invoking `@sumarr_%.lr.ph`, which implements the loop. Within the latter function, `getelementptr` computes the address of a given element of the array, as does the LLVM instruction of the same name, and `load-i64l` reads a 64-bit signed integer in little-endian order from the address computed by `getelementpr`.

Some additional ACL2 features should be mentioned at this point. First, inside the standard Lisp `(declare...)` form in Figure 2 one will note some ACL2-specific declarations. `:measure` provides a measure predicate to be used in termination analysis; this measure should decrease for every recursive call of the function, and ACL2's termination analysis machinery will prove that it does. (NB: `nfix` casts its input to a natural number.) `:stobjs` declares `st` as a single-threaded object, and `:guard` restricts the "types" of the input parameters to the function — if the guards are violated during execution, the function will not be invoked. However, guards are *not* a part of the ACL2 logic; thus, one will note that these guard conditions are also explicitly checked in the opening conditional of the generated function. But, these individual predicates are surrounded by `mbt`, which signals that these checks need not be made during execution. (ACL2 guards are a complex subject; the curious reader is referred to the ACL2 documentation [12].)

Finally, observe that the translated function for the LLVM loop conveniently becomes a tail-recursive function in ACL2. Tail-recursive functions are very nice for execution, as Lisp compilers know to optimize a tail call into a jump; thus, an arbitrary number of recursive tail calls can be made without exhausting the stack (by contrast, recursive calls that are not tail-recursive can "blow up" the stack after a few thousand frames.) However, tail-recursive functions are not so convenient for reasoning, as will be addressed in the next section.

4. FORMAL ANALYSIS OF TAIL-RECURSIVE FUNCTIONS OPERATING ON SINGLE-THREADED OBJECTS

As noted by several researchers (e.g. [7]), reasoning about functions on stobjs is more difficult than performing proofs about traditional ACL2 functions on lists which utilize primitive recursion. This difficulty is compounded by the fact that in order to scale to data structures containing millions of elements, recursive functions must be tail-recursive (this would be the case whether we used stobjs or not). Previous work has described a preliminary method to deal with these issues, at least for the case of functions that operate over large stobj arrays [8]. With this method, we have been able to show, for a number of such functions, that a tail-recursive, stobj-based function that "marches" from lower

```
(defun @sumarr_%.lr.ph (%.01 %indvars.iv %n %array st)
  (declare (xargs :measure (nfix (- (nfix %n) (nfix %indvars.iv)))
                  :stobjs st
                  :guard (and (integerp %.01) (natp %indvars.iv) (natp %n)
                              (natp %array) (< %indvars.iv %n))))
  (if (not (and (mbt (integerp %.01)) (mbt (natp %indvars.iv)) (mbt (natp %n))
                (mbt (natp %array)) (mbt (< %indvars.iv %n)))) %.01
    (let ((%2 (getelementptr %array %indvars.iv 8)))
      (let ((%3 (load-i64l %2 st)))
        (let ((%4 (ifix (+ %3 %.01))))
          (let ((%indvars.iv.next (nfix (+ %indvars.iv 1))))
            (let ((%exitcond (if (= %indvars.iv.next %n) 1 0)))
              (if (= %exitcond 1) %4
                (@sumarr_%.lr.ph %4 %indvars.iv.next %n %array st))))))))))

(defun @sumarr_0 (%n %sum %array st)
  (declare (xargs :stobjs st
                  :guard (and (natp %n) (integerp %sum) (natp %array))))
  (if (not (and (mbt (natp %n)) (mbt (integerp %sum)) (mbt (natp %array))))
      (ifix %sum)
    (let ((%1 (if (= %n 0) 1 (ifix %sum))))
      (if (= %1 1) (ifix %sum) (ifix (@sumarr_%.lr.ph %sum 0 %n %array st))))))
```

Figure 2: Translation to ACL2.

array indices to upper ones is equivalent to a primitive recursive version of that function operating over a simple list. This technique, which has been named Hardin's Bridge[1], relates a traditional imperative loop operating on an array of values to a primitive recursion operating on a list.

As depicted in Figure 3, the process of building this bridge begins by translating an imperative loop, which operates using `op` on a data array `d`, into a tail-recursive function (call it `x-tail`) operating on a stobj `st` containing an array field, also named `d`. This tail-recursive function is invoked as `(x-tail j res st)`, where `res` is an accumulator, and the index `j` counts up from 0 to the size of the data array, `*SZ*`. `x-tail` is then shown to be equivalent to a non-tail-recursive function `x-iter` that also operates over the stobj `st`. This function is invoked as `(x-iter k res st)`, where index `k` counts down from `*SZ*` toward zero. The equivalence of these two functions is established in a manner similar to the `defiteration` capability found in `centaur/misc/iter.lisp` in the ACL2 distributed books.

We now have a primitive recursion that operates on an array field of a stobj. What we desire, however, is a primitive recursion that operates over a list. One such recursion is as follows:

```
(defun x (res d)
  (if ((endp d) res
    (op (car d) (x res (cdr lst))))))
```

This function can be related to `x-iter` by way of a theorem involving `take`, which as one will recall, returns the first n elements of a list. Additionally, `defthm` is the ACL2 form for stating a conjecture, which the ACL2 system will attempt to prove. Many `defthm` forms include an `implies` operator, which unsurprisingly, is logical implication.

[1]In memory of Scott Hardin, father of the first author: a civil engineer who designed several physical bridges, and a man who valued rigor.

```
(defthm x-iter-take--thm
  (implies (and (stp st) (natp j)
                (integerp res) (<= j *SZ*))
           (= (x-iter j res st)
              (x res (take j (nth *DI* st))))))
```

Here, `(nth *DI* st)` is the way to refer to the entire data array as a list; this is legal in theorems, but is not permitted within function definitions, by ACL2 stobj rules.

If the preconditions are met, then by functional instantiation (setting j = *SZ*),

```
(= (x res (nth *DI* st)) (x-iter *SZ* res st))
```

and, by the earlier equivalence,

```
(= (x-tail 0 res st) (x-iter *SZ* st)
```

so, finally,

```
(equal (x-tail 0 res st) (x res (nth *DI* st)))
```

Once some auxiliary lemmas are proven, ACL2 proves this result automatically.

We can now prove theorems about the array-based iterative loop by reasoning about `x`, which has a much more convenient form from a theorem proving perspective.

We have employed this bridge technique on several predicates and mutators, including an array-based insertion sort employing a nested loop. Returning to our example, we can use the above technique to prove that `@sumarr_%.lr.ph` is equal to the following primitive recursive function:

```
(defun sumlist64 (res lst)
  (declare (xargs :measure (len lst)))
  (cond ((not (true-listp lst)) (ifix res))
        ((endp lst) (ifix res))
        (t (+ (ifix (load-i64ll (take 8 lst)))
      (sumlist64 res (nthcdr 8 lst))))))
```

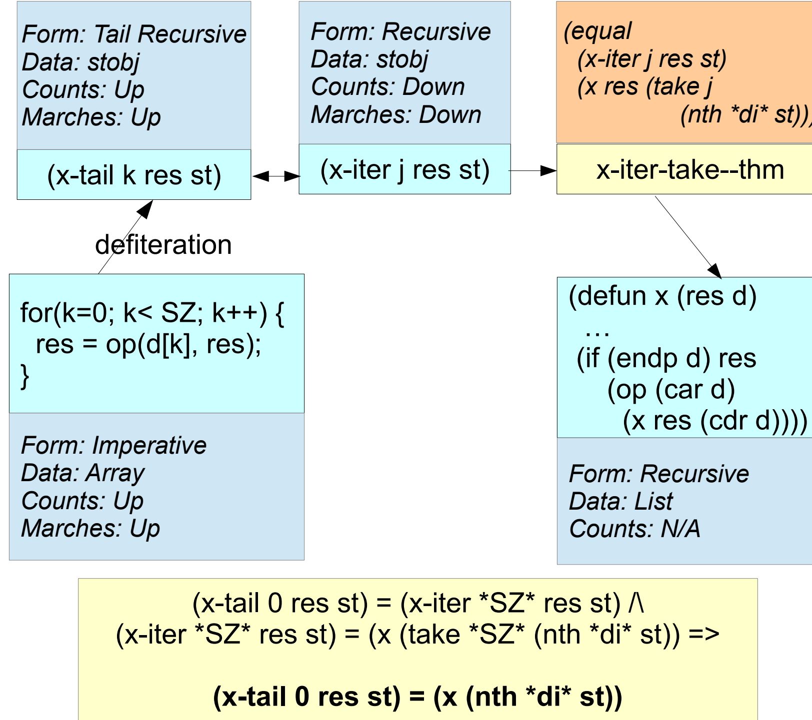

Figure 3: Hardin's Bridge: Relating an imperative loop to a primitive-recursive ACL2 function.

`sumlist64` consumes a list of unsigned bytes eight elements at a time, converts those eight elements to a single signed 64-bit integer observing little-endian byte order (via the `load-i64ll` function), then accumulates a sum. Proving properties about `@sumarr_%.lr.ph` can then be accomplished by proving them instead about `sumlist64`, a much simpler function, and one that is of a non-tail-recursive form better-suited for theorem proving.

5. RELATED WORK

Zhao *et al.* [15] produced several different formalizations of operational semantics for LLVM in Coq [4], noting that their intention is to produce a verified LLVM compiler, similar to the verified CompCert compiler due to Leroy [10] (CompCert does not utilize the LLVM intermediate form). As such, their emphasis on formalizing LLVM operational semantics makes sense. We also considered creating an "LLVM interpreter" in ACL2 (Zhao *et al.* utilized the OCaml extraction capability of the Coq environment to produce such an interpreter), resulting in a "shallow embedding", but decided that a translation to ACL2 (thus producing a "deep embedding") would allow us to begin proving properties about LLVM programs with much less effort. Our approach was also influenced by Magnus Myreen's "decompilation into logic" work [13]. Our approach could be characterized as a sort of decompilation into logic, but we do not go to the same lengths as Myreen to assure that the decompilation process is sound. We also have the advantage of starting with a form that is functional, whereas Myreen has tackled the much more difficult problem of decompiling imperative machine code.

6. CONCLUSION AND FUTURE WORK

We have built a translator from the LLVM intermediate form to the applicative subset of Common Lisp accepted by the ACL2 theorem prover. The translator produces an executable ACL2 specification featuring tail recursion, as well as in-place updates via ACL2's single-threaded object (stobj) mechanism, and we have utilized these features in order to validate our translated models via testing. We also presented a technique for reasoning about tail-recursive ACL2 functions that execute in-place, utilizing a formally proven "bridge" to primitive-recursive versions of those functions operating on lists.

Future work should focus on improving the translator, especially in areas of optimizing the ACL2 output, as well as refining the memory model in ACL2 in order to ease automated reasoning. We also need to continue to develop techniques for reasoning about tail-recursive functions and stobjs; some progress has been made during the current effort, but much more work is needed in order to make the process easier.

7. ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful comments. This work was sponsored in part by the United States Department of Defense.

8. REFERENCES

[1] A. W. Appel. SSA is functional programming. In *SIGPLAN Notices*, volume 33, pages 17–20. ACM, April 1998.

[2] G. E. Box and N. R. Draper. *Empiricial Model-Building and Response Surfaces*. John Wiley and Sons, 1987.

[3] R. S. Boyer and J. S. Moore. Single-threaded objects in ACL2. *PADL 2002*, 2002.

[4] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, April 2013. Version 8.4pl21.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *POPL*, volume 13, pages 451–490. ACM, October 1991.

[6] D. Doligez, A. Frisch, J. Garrigue, D. Remy, and J. Vouillon. The OCaml system release 4.00 documentation and users guide. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[7] D. S. Hardin and S. S. Hardin. Efficient, formally verifiable data structures using ACL2 single-threaded objects for high-assurance systems. In S. Ray and D. Russinoff, editors, *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 100 – 105. ACM, 2009.

[8] D. S. Hardin and S. S. Hardin. ACL2 meets the GPU: Formalizing a cuda-based parallelizable all-pairs shortest path algorithm in ACL2. In R. Gamboa and J. Davis, editors, *Proceedings of the 11th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 114, pages 127 – 142. EPTCS, 2013.

[9] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[10] X. Leroy. Formal verification of a realistic compiler. In *Communications of the ACM*, volume 52, pages 107–115, 2009.

[11] LispWorks Ltd. Common Lisp hyperspec. `http://www.lispworks.com/documentation/HyperSpec/Front/index.htm`.

[12] J. Moore and M. Kaufmann. ACL2 Documentation. `http://www.cs.utexas.edu/users/moore/acl2`.

[13] M. O. Myreen, M. J. C. Gordon, and K. L. Slind. Decompilation into logic — improved. In *FMCAD'12*. ACM/IEEE, October 2012.

[14] The LLVM Project. The LLVM compiler infrastructure. `http://llvm.org/`.

[15] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL'12*. ACM, January 2012.