Computing Verified Machine Address Bounds during Symbolic Exploration of Code

J Strother Moore

Abstract When operational semantics is used as the basis for mechanized verification of machine code programs it is often necessary for the theorem prover to determine whether one expression denoting a machine address is unequal to another. For example, this problem arises when trying to determine whether a read at the address given by expression a is affected by an earlier write at the address given by b. If it can be determined that a and b are definitely unequal, the write does not affect the read. Such address expressions are typically composed of "machine arithmetic function symbols" such as `+`, `*`, `mod`, `ash`, `logand`, `logxor`, etc., as well as numeric constants and values read from other addresses. In this paper we present an abstract interpreter for machine address expressions that attempts to produce a bounded natural number interval guaranteed to contain the value of the expression. The interpreter has been proved correct by the ACL2 theorem prover and is one of several key technologies used to do fast symbolic execution of machine code programs with respect to a formal operational semantics. We discuss the interpreter, what has been proved about it by ACL2, and how it is used in symbolic reasoning about machine code.

1 Preface

One might ask why a paper on the ACL2 project is included in the volume marking the 20^{th} and 25^{th} anniversaries of the European ProCoS project. ProCoS was in part inspired by the successful effort at Computational Logic, Inc. (CLI), first published in 1989, to verify a system "stack," from a gate-level description of a microprocessor, through an assembler, linker, loader, two compilers, and an operating system, to several applications. All were

Department of Computer Science, University of Texas, Austin, TX, USA; email: moore@cs.utexas.edu

verified using the Nqthm [5] theorem prover and their correctness results were designed to compose so that each level relieved the preconditions of the level below. The result was a mechanically checked theorem of the form: under certain very specific preconditions on the resources available and the inputs, the application programs (when compiled, linked, and loaded) run correctly on the hardware. The only unverified assumptions were the ones at the bottom: the fabrication of the gate-level description was faithful to the design and the physical gates behave as logically specified [1].

But the CLI stack inspired more than ProCoS. It was one of several Nqthm projects in the late 1980s and early 1990s involving models of commercial interest. See for example the work on the C String Library as compiled by `gcc` for the Motorola 68020 [6]. These projects stressed Nqthm in ways we had not seen before: its capacity, efficiency, and convenience as a practical functional programming language. Thus was born, in 1989, ACL2: A Computational Logic for Applicative Common Lisp [7, 13, 12, 11] ACL2 was a reimplementation of Nqthm in an applicative subset of Common Lisp [19]. But while the logic of Nqthm was a "homegrown" dialect of pure Lisp, the logic of ACL2 is applicative Common Lisp, a fast, efficient, widely supported ANSI standard programming language.

ACL2 has since been used in many industrial projects and is in use regularly at several companies involved with microprocessor design. For a good illustration of how ACL2 can be used in industry, see [18].

2 Introduction

Operational semantics has long been used to formalize and mechanically verify properties of machine code programs. Examples of the Edinburgh Pure Lisp Theorem Prover, Nqthm and ACL2 being used to prove functional correctness of code under formal operational semantics may be found in numerous publications [2, 1, 6, 21, 16, 17, 20, 10].

In such applications, terms in the logic are used to represent machine states, transition functions define the effects of individual instructions, these instruction-specific transition functions are then wrapped up into a "big switch" single-step function that applies the transition function dictated by the opcode of the next instruction, and finally the single-step function is wrapped up into a recursive iterated step function for giving semantics to whole programs. Typically the program being analyzed is stored in the state, either encoded numerically in memory or symbolically in some "execute only" state component. Theorems are then posed, typically, as implications asserting that if the initial state has some property then the "final" state produced by the iterated step function has some related property. These theorems are typically proved by induction but the "heavy lifting" in the proof is done by a rewriting strategy that explores the various paths through the program and

composes and simplifies the individual state transitions. The rewriting strategy is just deductive implementation of *symbolic evaluation* which we sometimes also call *code walking*. The basic idea of symbolic evaluation is to start with a symbolic state expression containing a concrete program counter and program code but containing variables in some state components (e.g., memory locations holding program data). Hypotheses typically constrain these variables. To symbolically step that state: retrieve the instruction at the program counter, instantiate the transition function with that instruction, simplify the resulting state (rearranging expressions representing the contents of various registers and memory locations, testing them, and producing an `IF`-expression with new states with known program counters), and repeat on all the new states until some condition is satisfied.[1]

We sometimes refer to these proofs as *code proofs* because they can establish properties of explicit machine code.

Fundamental to this approach to semantics are the terms denoting reads and writes to the memory of a state because every transition requires manipulating the memory. In this work we focus on a byte addressed memory and use these terms for read and write:

$\mathtt{R}(a, n, st)$: returns the natural number obtained by reading n bytes starting at address a in the memory of state st
$\mathtt{!R}(a, n, v, st)$: returns the new state obtained by writing n bytes of natural number v into the memory of st starting at address a

We call a an *address* and n an *extent*. `R` and `!R` use the Little Endian convention to represent natural numbers as sequences of bytes. If an integer is supplied for v above, its twos complement representation – a natural number – is used. For example, `!R` writes the least significant byte of the binary representation of v into address a and writes the more significant bytes into the higher addresses.

`R` and `!R` enjoy certain properties that are crucial to code proofs. One such property is:

$$a, n, b, m \in \mathbb{Z} \wedge (a + n \leq b) \rightarrow \mathtt{R}(a, n, \mathtt{!R}(b, m, v, st)) = \mathtt{R}(a, n, st).$$

Such a theorem is called a *read-over-write* theorem because it tells us about the results of reading after writing. This particular read-over-write theorem

[1] The process just described is just ordinary mathematical simplification of the iterated step function applied to the initial state. A special case of symbolic evaluation is "symbolic simulation" or "bit blasting" by which we mean a process whereby objects from a given finite set are represented using nested structures whose leaves are Boolean constants and variables. The process computes related objects from definitions or other equations using Boolean decision methods typically based on binary decision diagrams (BDDs) or Boolean satisfiability procedures (SAT). ACL2 supports symbolic simulation, e.g., see the ACL2 online documentation topic GL, but in this paper we are concerned with straightforward simplification.

says the write can be ignored if the read fetches bytes in memory addresses below those written. There are other theorems to deal with overlapping reads and writes and reads above writes. There are analogous *write-over-write* theorems for simplifying state expressions. All are crucial to code proofs.[2]

But what is of concern here is how, in a theorem proving context, we establish such inequalities as $(a + n \leq b)$ when a, n, and b are given by terms produced by symbolic evaluation of machine code. Such hypotheses litter the read-over-write and write-over-write conditional rewrite rules that are heavily used in code proofs. These rules are typically tried many more times than they are successfully applied: given an arbitrary read-over-write expression one must try to establish the hypotheses of each rule to determine whether the read is below, overlapping, or above the write. Furthermore, in typical code proofs, thousands of read-over-write expressions are encountered. Finally, the expressions a and b can become very large.

To put some numbers on the adjectives "heavily used," "large," etc., consider the largest symbolic state encountered while symbolically exploring a machine code implementation of the DES algorithm. The state in question represents the end of one path through the 5,280 instructions in the decryption loop. The normalized state expression contains 2,158,895 function calls, including 58 calls of `!R` to distinct locations and 459,848 calls of `R`. (Repeated writes to the same location are eliminated by the rewriting process.) That state expression also contains 1,698,987 calls of arithmetic/logical functions such as addition, subtraction, multiplication, modulo, and bitwise logical `AND`, exclusive `OR`, shift, etc. The largest value expression written is given by a term involving 147,233 function applications, 31,361 of which are calls of `R` and the rest are calls of arithmetic/logical functions. Values written often become indices into arrays and thus become part of address expressions.

We found it impractical to use ACL2's conventional arithmetic library to answer the address comparison questions that arise while building up such large state expressions. But ACL2 allows the user to extend the rewriter with special-purpose symbolic manipulation programs if those programs — which are written in the ACL2 programming language — are first proved correct by ACL2. So we developed special-purpose programs to answer such questions as "is $(a + n \leq b)$ true?" or more generally, "how do the values of expressions a and b compare?" The core technology is an Abstract Interpreter over Natural Number Intervals called `Ainni`, which takes a term and the context in which it occurs and tries to compute a bounded natural number interval containing all possible values of the term in that context. `Ainni` is purely syntactic — it just walks through the term bounding every subterm — and can be thought of as a verified type-inference mechanism where the types are intervals. `Ainni` was then used to develop a variety of metafunctions for manipulating the

[2] Typical machine state models involve many other state components, their "accessor" and "updater" function symbols, and their analogues to "read-over-write" theorems, etc. But we ignore them in this paper since we are focused on address resolution.

gigantic expressions produced by the symbolic evaluation of machine code sequences containing thousands of instructions.

In section 3 we give some practical information about ACL2 as well as explain ACL2 notation which we often use in place of conventional notation because our techniques involve metafunctions which manipulate the internal ACL2 representation of terms. In section 4 we discuss that representation and metafunctions. In section 5 we introduce ACL2's pre-existing notion of "bounder" functions and a library of elementary bounders. In section 6 we describe the key idea: `Ainni`, our abstract interpreter for machine arithmetic expressions that attempts to produce a bounded interval containing the value of the expression. Also in this section we show the correctness results for `Ainni`. These results have been proved by ACL2 and are necessary if `Ainni` is to be used in verified metafunctions. In section 7 we illustrate calls of `Ainni` and the interpretation of its results. In section 8 we exhibit a metafunction that uses `Ainni` to simplify a certain kind of `MOD` expression. This section shows how a metafunction assembles the results of `Ainni` into a provably correct answer. In section 9 we briefly describe other applications of `Ainni`, including the motivating one for simplifying read-over-write expressions. In section 10 we briefly mention related work. Finally we summarize in section 11 and acknowledge the help of colleagues in section 12.

3 A Little Background on ACL2

In this section we present a little practical background on ACL2, its documentation and user-developed libraries. Then we sketch the syntax of the ACL2 logic and reveal a bit about the implementation of the ACL2 theorem prover in Lisp. We also reveal a bit about the semantics.

ACL2 was initially developed by Robert S. Boyer and the author starting in 1989. However, since the early 1990s it has been extensively further developed, documented, maintained, and distributed by Matt Kaufmann and the author. It is available for free in source code form from the ACL2 home page [14].

When we refer to ":DOC x" we mean the documentation topic x in the online ACL2 documentation, which may be found by visiting the ACL2 home page, clicking on The User's Manuals, then clicking on ACL2+Books Manual and typing x into the "Jump to" box.

In ACL2 parlance, a "book" is a file of definitions and theorems that can be loaded (see :DOC `include-book`) into an ACL2 session to extend the current theory. The actions of the ACL2 rewriter (and other parts of the prover) are influenced by previously proved theorems. Books are often developed with some particular problem domain and proof strategy in mind and when included in a session configure the prover to implement that strategy.

In this paper we refer to several books in the *ACL2 Community Book Repository*. The repository is developed and maintained by the ACL2 user

community. The top of the directory structure may be viewed by visiting GitHub at `https://github.com/acl2/acl2`. A particular file may be found by clicking your way down the directory hierarchy. For example, to find `books/projects/stateman/stateman22.lisp` start on the GitHub page above and click on `books`, then `projects`, etc.

ACL2 is the name of a programming language, a first order logic, a theorem prover, and a program/proof development environment. The ACL2 programming language is an extension of the applicative subset of Common Lisp [19]. The logic includes an axiomatization of that language consistent with Common Lisp. The theorem prover and environment are implemented (largely) in the ACL2 programming language.

In ACL2, the term $\texttt{R}(a, n, st)$ is written `(R a n st)`. ACL2 is case insensitive so this could also be written `(r a n st)` or `(R A N ST)`. In this paper we write variable symbols in lowercase italics. We tend to use case, both capitalization and uppercase, merely for emphasis. If our use of case and italics is confusing just ignore them!

Internal to the ACL2 theorem prover, the term `(R` a n st`)` is represented by the Lisp list that prints as `(R A N ST)`, i.e., a list of length 4 whose `car` or first element is the Lisp symbol `R`, and whose `cdr` or remaining elements are given by the list `(A N ST)`. Consing the symbol `R` onto the list `(A N ST)` produces the list `(R A N ST)`. In Lisp we could create this list by evaluating `(cons 'R '(A N ST))`, or `(cons 'R (list 'A 'N 'ST))` or `(list 'R 'A 'N 'ST)`. These three examples illustrate the most common idioms used to create terms when programming the theorem prover.

This brings us to the single quote mark and Lisp evaluation. The Lisp convention is that a single quote mark followed by a Lisp expression α is read as though the user had typed `(QUOTE` α`)`. Thus, `'(R A N ST)` is read as `(QUOTE (R A N ST))`.

`QUOTE` is a "special symbol" in the semantics of Lisp. The result of evaluating `(QUOTE` α`)` is α. This discussion of internal representation and the special meaning of `QUOTE` and the single quote mark are relevant to our discussion of metafunctions in the next section.

But to foreshadow that discussion, it happens that if α is the Lisp representation of an ACL2 term then `'`α is the Lisp representation of another ACL2 term, that second term in fact denotes a constant in the ACL2 logic, and there is an ACL2 function, say $\mathcal{E}$, called an "evaluator," that when applied to that constant and an appropriate association list ("alist") will return the same thing as the value of α. For example, since `(R` a n st`)` is an ACL2 term, then so is `'(R A N ST)`, the latter term denotes a constant in the ACL2 logic, and

`(`$\mathcal{E}$ `'(R A N ST) (list (cons 'A` a`) (cons 'N` n`) (cons 'ST` st`)))`
=
`(R` a n st`)`

is a theorem of ACL2.

In Lisp, certain constants, in particular symbols `T` and `NIL`, numbers, character objects, and strings, evaluate to themselves. Thus, when writing Lisp it is not necessary to quote these constants. But constants appearing in ACL2 terms, even `T`, `NIL`, and numbers, are always quoted. This is achieved without inconveniencing the user by *translating* user type-in into ACL2's internal form. Thus, the term we write as `(R 4520 8` *st*`)` is represented inside the theorem prover as `(R '4520 '8 ST)` which we could display as `(R (QUOTE 4520) (QUOTE 8) ST)`. The user could in fact input the term in any of these ways. All three expressions produce exactly the same internal form. And because ACL2 is Lisp, it happens that all three are not only ACL2 terms but Lisp expressions and they produce the same results when evaluated by Lisp.

Some other ACL2 function symbols used in this paper are shown in Figure 1. In Lisp, a test or predicate is said to be "false" if its value is `NIL` and is said to be "true" otherwise. The symbols `force` and `hide` of Figure 1 are trivial identity functions used to communicate pragmatic information to the ACL2 prover. See :DOC `force` and `hide`.

ACL2 term	*name*	*conventional notation*
`(if` x y z`)`	if-then-else	x `?` y `:` z
`(implies` p q`)`	logical implication	$p \to q$
`(and` p q`)`	logical conjunction	$p \wedge q$
`(or` p q`)`	logical disjunction	$p \vee q$
`(not` p`)`	logical negation	$\neg p$
`(equal` x y`)`	equality	$x = y$
`(integerp` x`)`	"is-integer"	$x \in \mathbb{Z}$
`(natp` x`)`	"is-natural"	$x \in \mathbb{N}$
`(<` x y`)`	less than	$x < y$
`(<=` x y`)`	less than or equal	$x \leq y$
`(+` x y`)`	addition	$x + y$
`(-` x y`)`	subtraction	$x - y$
`(*` x y`)`	multiplication	$x \times y$
`(ifix` x`)`	"coerce-to-integer"	if x is an integer, x; else 0
`(expt` x y`)`	exponentiation	x^y
`(mod` x y`)`	modulus	$x \bmod y$
`(ash` x y`)`	shift	$\lfloor x \times 2^y \rfloor$
`(logand` x y`)`	bitwise and	$x\&y$
`(logior` x y`)`	bitwise inclusive or	$x\|y$
`(logxor` x y`)`	bitwise exclusive or	$x\hat{}y$
`(force` x`)`		x
`(hide` x`)`		x
`(R` a n st`)`	read n bytes from addr a	
`(!R` a n v st`)`	write n bytes of v to addr a	

Fig. 1 Some ACL2 Function Symbols

In the internal representation of ACL2 terms, all function symbols take a fixed number of arguments. "Functions" that allow varying numbers of arguments are handled as Lisp macros that expand during the previously mentioned translation phase. For example, the internal form of `(+` i j k`)` is

actually `(binary-+ i (binary-+ j k))`. The symbol `+` is a macro that expands into a term that uses the function symbol `binary-+`. Of the "function symbols" shown in Figure 1 the symbols `+`, `*`, `logand`, `logior`, and `logxor` are actually macros that expand into right-associated calls of function symbols that take exactly two arguments. The "functions symbols" `and` and `or` are macros that expand into nests of `IF` expressions. But in this paper we ignore such details and will pretend that they are all function symbols, not macros; when discussing term processing functions we will act like these symbols have exactly two arguments. We mention this detail only to reassure readers familiar with ACL2 that our metafunctions do not mistake macros for function symbols.

ACL2 is untyped and all ACL2 functions are total; thus, ACL2 expressions mean *something* no matter what well-formed arguments are supplied; however we will always use them conventionally and their completions are unimportant here. For example, ACL2's universe includes the rationals but not the irrationals. Thus, `(expt 2 1/2)` is a well-formed ACL2 term, it is indeed equivalent to a certain constant, but that constant is not $\sqrt{2}$. But this does not matter here because no term involved in this work applies `expt` to a non-integer.

4 Metafunctions

ACL2 "metafunctions" are ordinary ACL2 functions that operate on the internal representation of ACL2 terms. Correctness is stated in terms of "evaluators." Once ACL2 has proved a metafunction correct, the metafunction may be used by the theorem prover directly on the internal representation of terms [4]. Metafunctions have been part of ACL2 since its beginning; indeed, they were first introduced and described in 1979 [3] as part of the prover that became Nqthm [5].

An "evaluator" is a function that interprets an object as a term, with respect to some assignment giving meaning to variable symbols. Lisp's `eval` would be a wonderful evaluator if it were admissible in ACL2's first order logic of total recursive functions, but it is not. Fortunately, it suffices for ACL2's purposes to admit evaluators for a finite number of already-introduced function symbols and the ACL2 system provides a macro, `defevaluator`, that makes this easy. See :DOC `defevaluator`.

More technically, let σ be a set of ACL2 function symbols. An ACL2 *evaluator* function over σ is a function ev of two arguments, x, treated as the internal representation of a term, and *alist*, treated as an association list mapping variable symbols to values. The value, v, of (*ev x alist*) is constrained to have certain properties including: If x is a symbol other than `NIL`, v is the value assigned x by *alist*. If x is `'`c, v is c. If x is of the form $(g\ x_1\ \ldots\ x_n)$, where $g \in \sigma$, then v is (*g* (*ev* x_1 *alist*) ... (*ev* x_n

alist)). Additional constraints include that *ev* be able to interpret LAMBDA-applications and that on x of the form $(g\ x_1\ \ldots\ x_n)$ where $g \notin \sigma$, *ev* is a function of the (*ev* x_i *alist*).

Henceforth, we will assume that $\mathcal{E}$ is an ACL2 evaluator function over all of the functions mentioned in this paper (except $\mathcal{E}$ itself!)[3].

Thus,

```
(E '(!R '4000 '8 (LOGAND X Y) ST)
   (LIST (CONS 'X x)
         (CONS 'Y y)
         (CONS 'ST st)))
=
(!R '4000 '8 (LOGAND x y) st)
=
(!R 4000 8 (LOGAND x y) st).
```

A *metafunction* is an ordinary list processing function in ACL2 with the property that it takes the internal representation of a term and returns the internal representation of an equivalent term. To be precise, a metafunction must be proved to operate correctly on "pseudo terms." Pseudo terms are term-like list structures that do not necessarily obey all the internal invariants on ACL2's term representation. Before the output of a metafunction is *used* to replace its input, the output is checked to satisfy all the internal invariants, unless the user has also proved that the function preserves them [15].

The general form of the theorem establishing that fn is a verified metafunction is:

```
(implies (and (pseudo-termp x)
              (alistp alist))
         (equiv (E x alist)
                (E (fn x mfc state) alist)))
```

where $\mathcal{E}$ is any evaluator. The variable name mfc stands for *metafunction context* and *state* is the state of the ACL2 system, which together give fn access to contextual and heuristic data.

If this theorem has been proved by ACL2, then the ACL2 rewriter is logically permitted to replace any term x by the result computed by calling fn on x provided the returned object represents a term. This argument is presented in detail in :DOC meta.

Furthermore, by convention, if the metafunction returns an answer of the form '(IF *test new x*) when applied to x, the rewriter uses *new* as the simplified version of x provided it can backchain and establish *test*. Thus, fn can check some hypotheses syntactically and leave others to be relieved

[3] The actual name of this evaluator is stateman-eval, "stateman" being the name of the "State Management" book that motivated this work. We simply find stateman-eval inconveniently long for use in a paper.

by the rewriter. This design means that the user does not have to prove that the metafunction properly interprets the data found in mfc and *state*. It also means that the ACL2 implementors do not have to formalize that data but instead merely provide functions for accessing certain parts of it. However, when those functions are used properly in a metafunction and the metafunction accurately "exports" what was learned as a conjunct included in *test*, it is generally easy for ACL2 to backchain and prove *test*: it is generally proved by the trusted internal routines of ACL2 for interpreting the data in mfc and *state*.

Since ACL2's implementation language is ACL2, programming metafunctions is just like programming theorem proving utilities, except that we generally use ACL2 to prove that our programs are correct. For example, suppose we wanted a utility for conservatively determining that an expression x always returns a natural number. Here is such a function.[4] It is not actually necessary for the user to define this particular function. ACL2 has much more sophisticated built-in ways to recognize expressions that return naturals. But this function is a good warm-up.

```
(defun syntactic-natp (x)
  (cond
   ((atom x) nil)
   ((eq (car x) 'QUOTE)
    (natp (nth 1 x)))
   ((member (car x) '(+ * LOGAND LOGIOR LOGXOR ASH MOD))
    (and (syntactic-natp (nth 1 x))
         (syntactic-natp (nth 2 x))))
   ((eq (car x) 'HIDE)
    (syntactic-natp (nth 1 x)))
   ((eq (car x) 'R) t)
   (t nil)))
```

Here we use `atom` to recognize variable symbols, `(car` x`)` to fetch the top-level function symbol (or the `QUOTE` mark) of the non-atomic term x, `(nth 1` x`)` to fetch the constant inside a `QUOTE`d expression, and `(nth` i x`)` to fetch the i^{th} argument of function application x.

ACL2 can prove that if `(syntactic-natp` *term*`)` is true, then `(natp (`$\mathcal{E}$ *term alist*`))`.

```
(implies (syntactic-natp term)        ; {syntactic-natp correct}
         (natp (E term alist)))
```

We might then use `syntactic-natp` in the definition of some metafunction. For example, suppose we wished to write a metafunction that recognized terms of the form `(natp` x`)` and replaced them by `T` when x is a `syntactic-natp` expression. Here is that metafunction:

[4] As indicated above, a correct definition will use `BINARY-+` instead of `+`, `BINARY-*` instead of `*`, etc.

```
(defun meta-natp (x)
  (cond ((and (not (atom x))
              (eq (car x) 'NATP)
              (syntactic-natp (nth 1 x)))
         '(QUOTE T))
        (t x)))
```

ACL2 can prove:

```
(implies (pseudo-termp x)                    ; {meta-natp correct}
         (equal (ℰ x alist)
                (ℰ (meta-natp x) alist)))
```

Given this theorem, ACL2 would be justified in applying `meta-natp` to every expression it ever encountered and replacing the expression by the result. That would be needlessly inefficient since `meta-natp` only changes some `NATP` expressions. The user-interface to ACL2 requires the user to provide pragmatic information identifying likely targets expressions, in this case, calls of `NATP`.

5 Bounders

The key to resolving such questions as $(a + n \leq b)$ by syntactic analysis is to be able to compute a bounded interval containing all possible values of a term. In this paper we assume all intervals are closed, bounded, and over the naturals (i.e., integer intervals with non-negative lower bound). We denote intervals over the naturals by $[lo, hi]$, where both lo and hi are natural numbers and $lo \leq hi$.

Imagine that x and y lie within certain bounded closed intervals over the naturals. Then it is easy to compute an interval containing their sum by appealing to the following theorem:

$$(x \in [lo_x, hi_x] \wedge y \in [lo_y, hi_y]) \rightarrow (x + y) \in [lo_x + lo_y, hi_x + hi_y]$$

It is easy to imagine a function that takes a term, like `(+ x y)` in ACL2, and computes an interval containing its value, provided it can recursively compute such intervals for x and y. The question is: given intervals containing the arguments of a function f, can we compute an interval containing the value of f on those arguments?

In ACL2, an n-ary function g is a *bounder* for an n-ary function f if, for closed bounded intervals int_1, int_2, ..., int_n over the natural numbers,

when $x_i \in int_i$, for all $1 \leq i \leq n$, then $g(int_1, \ldots, int_n)$ is an interval and $f(x_1, \ldots, x_n) \in g(int_1, \ldots, int_n)$.[5]

The file `books/tau/bounders/elementary-bounders.lisp`, in the ACL2 Community Books repository, developed by the author, defines and verifies bounders for `+`, `*`, `-`, `FLOOR`, `MOD`, `LOGAND`, `LOGNOT`, `LOGIOR`, `LOGXOR`, `EXPT`, `ASH` and a few other functions.

For example, here is a version of the bounder for `LOGAND` that is correct provided the two intervals int_x and int_y are closed bounded intervals over the naturals. This function is less general than that in the `elementary-bounders` Community Book, which deals with the various kinds of ACL2 intervals, including the cases where the bounds are negative integers. But the simple function below illustrates the basic ideas in all of our bounders.

```
(defun natp-tau-bounder-logand (intx inty)
  (let ((lox (tau-interval-lo intx))
        (hix (tau-interval-hi intx))
        (loy (tau-interval-lo inty))
        (hiy (tau-interval-hi inty)))
    (cond
     ((worth-computingp lox hix loy hiy)
      (make-natural-interval
       (find-minimal-logand lox hix loy hiy)
       (find-maximal-logand lox hix loy hiy)))
     (t
      (make-natural-interval 0 (min hix hiy))))))
```

Here the functions `tau-interval-lo` and `tau-interval-hi` extract the lower and upper bounds of an interval, and `make-natural-interval` constructs a closed ACL2 interval over the naturals when given appropriate lower and upper bounds. We discuss the "Tau System" of ACL2 in the next section.

The naive analytic bound on `(logand x y)` is $[0, \min(hi_x, hi_y)]$: the minimum possible value is `0` because x and y may not have any bits in common. The maximum possible value is the smaller of the upper limits of x and y, since `logand` just turns some bits off. For example, if $x \in [1032, 1039]$ and $y \in [520, 527]$, then this naive approach tells us that `(logand x y)` $\in [0, 527]$.

But this naive approach can grossly overestimate the bounding interval. In fact, `(logand x y)` $\in [8, 15]$, for any x and y bounded as assumed above, as can be confirmed by simply trying every combination of x and y in the two intervals and `logand`ing them together. If the two input intervals are sufficiently small this empirical approach is practical and often produces much tighter results. The functions `worth-computingp`, `find-minimal-logand`, and `find-maximal-logand` implement this empirical approach to interval

[5] ACL2 is actually a little more relaxed: it does not require that every argument of f be confined to an interval. ACL2 furthermore allows both open and closed intervals, possibly unbounded at either end, over not just the integers but also the rationals.

analysis. `Worth-computingp` deems it worth trying if the number of combinations is less than 2^{20}. ACL2 can do that many `logand` operations in about 0.004371 seconds on a MacBook Pro laptop with a 2.6 GHz Intel Core i7 processor.

6 Ainni: Abstract Interpreter over Natural Number Intervals

The "easy to imagine" function mentioned above, that takes a term and tries to compute an interval containing its value, is formalized in our function `Ainni`. `Ainni` is an abstract interpreter over natural number intervals. It uses the bounders in the elementary bounders book, and a few more, compute intervals.

To suggest how `Ainni` is defined we exhibit a simpler function `aii` below. For the full definition of `Ainni` see the ACL2 Community Book `books/-projects/stateman/stateman22.lisp`.

Suppose we have k function symbols, $op_1, \ldots, op_k$, of arities $n_1, \ldots, n_k$, and suppose we have a bounder function for each, `bounder-`$op_1, \ldots,$ `bounder-`op_k, respectively. Suppose x is a term over the op_i. Then here is a sketch of `aii`, an abstract interpreter that attempts to compute an interval containing the value of x. If it fails to find an interval it returns `nil`. We show the definition below and then paraphrase each case shown.

```
(defun aii (x)
  (cond
   ((atom x) nil)
   ((eq (car x) 'QUOTE)
    (cond ((natp (nth 1 x))
           (make-nat-interval (nth 1 x) (nth 1 x)))
          (t nil)))
   ...
   ((eq (car x) 'op_i)
    (let ((int1 (aii (nth 1 x)))
          ...
          (intn_i (aii (nth n_i x))))
      (cond
       ((and int1 ... intn_i)
        (bounder-op_1 int1 ... intn_i)
        (t nil)))))
   ...
   ((eq (car x) 'R)
    (cond ((and (not (atom (nth 2 x)))
                (eq (car (nth 2 x)) 'QUOTE)
```

```
            (natp (nth 1 (nth 2 x))))
       (make-nat-interval
        0
        (- (expt 2 (* 8 (nth 1 (nth 2 x)))) 1)))
      (t nil)))
(t nil)))
```

If x is a variable symbol, `aii` fails and returns `nil`. If x is a natural number constant, 'k, it returns the interval $[k, k]$. If x is an application of one of the known op_i, `aii` recursively computes an interval for the n_i arguments and, provided it succeeds on each, it calls the bounder for op_i to compute the interval for the call. If x is an application of `R`, `aii` asks whether the extent is a natural number constant, 'k, and if so returns $[0,2^{8k}-1]$. Otherwise, `aii` fails and returns `nil`.

Of course, the definition could be made more efficient by "failing early," e.g., not trying to compute an interval for the second argument if it failed to find one for the first. Furthermore, some terms can be bounded even if some of their arguments cannot be, e.g., `(logand` x `31)` $\in [0, 31]$ regardless of x's value. But `aii` is offered only as a suggestive model of our more sophisticated `Ainni`.

A more basic question arises when looking at the definition of `aii`: What about intervals for variables? The function above just fails if it encounters a variable. `Ainni` on the other hand takes another argument, called *ctx*, which provides contextual information, gleaned from the hypotheses governing the occurrence of the term x. For our purposes here, think of *ctx* as a map from Boolean terms to truth values. For example, the assumption that (`(R` a `8` st`)` < 16) would be coded in *ctx* as a pair associating the term `(<` `(R` a `8` st`)` `16)` with true.[6] `Ainni` uses its *ctx* argument to determine the arithmetic bounds on variable values. In our application, the only "variables" encountered are actually reads from memory, i.e., expressions of the form `(R` a n st`)`. If the extent of the read is a natural number constant then `(R` a 'k st`)` $\in [0, 2^{8k}-1]$. However, `Ainni` uses the *ctx* argument to try to narrow that interval by looking for assumptions on the bounds of `(R` a 'k st`)`.

`Ainni` takes three inputs: the term x to bound, a list of hypotheses, *hyps*, assumed so far, and *ctx*. It returns three values. These values are formally written as shown below and have the following interpretations:

- `(mv-nth 0 (Ainni` x *hyps ctx*`))`: the 0^{th} returned value of `(Ainni` x *hyps ctx*`)`. Informally this result is called the "output flag." When the output flag is non-`nil` ("true") it means `Ainni` successfully computed an interval for x; when the output flag is `nil`, `Ainni` could not find a suitable interval, e.g., perhaps the input term x is not in the set of terms recognized by `Ainni`. When the output flag is `nil`, the other two results are `nil` (and irrelevant).

[6] What we are calling *ctx* here is actually ACL2's "type-alist," and it pairs arbitrary terms with "types" gleaned from the governing hypotheses.

- `(mv-nth 1 (Ainni` *x hyps ctx*`))`: the 1^{st} returned value of `(Ainni` *x hyps ctx*`)`. Informally this result is called the "list of output hypotheses" and each element is called an "output hypothesis." When the output flag is non-`nil`, the list of output hypotheses is a list of terms that `Ainni` is relying on for the correctness of its answer. The output hypotheses include all of the elements of the input hypotheses *hyps* plus any hypotheses that `Ainni` extracted from *ctx* that contributed to its answer.
- `(mv-nth 2 (Ainni` *x hyps ctx*`))`: the 2^{nd} returned value of `(Ainni` *x hyps ctx*`)`. Informally this result is called the "output interval." When the output flag is non-`nil`, the output interval is a bounded natural number interval and the value of *x* (under the evaluator $\mathcal{E}$ with any variable assignment *alist*) lies within the output interval, provided the value of each output hypothesis is true (under the same evaluator $\mathcal{E}$ with the same variable assignment *alist*).

Four important theorems about `Ainni` have been proved with ACL2. The first says that when given a pseudo term *x* and a list of pseudo terms *hyps* the output hypotheses are all pseudo terms.

```
(implies                                              ; {Ainni 1}
 (and (pseudo-termp x)
      (pseudo-term-listp hyps))
 (pseudo-term-listp
  (mv-nth 1 (Ainni x hyps ctx))))
```

The second theorem establishes that when `Ainni`'s output flag is non-`nil` its output interval is indeed a bounded interval over the naturals.

```
(implies                                              ; {Ainni 2}
  (and (pseudo-termp x)
       (mv-nth 0 (Ainni x hyps ctx)))
  (and (tau-intervalp
        (mv-nth 2 (Ainni x hyps ctx)))
       (equal (tau-interval-dom
               (mv-nth 2 (Ainni x hyps ctx)))
              'INTEGERP)
       (tau-interval-lo
        (mv-nth 2 (Ainni x hyps ctx)))
       (tau-interval-hi
        (mv-nth 2 (Ainni x hyps ctx)))
       (<= 0 (tau-interval-lo
              (mv-nth 2 (Ainni x hyps ctx))))))
```

The first conjunct in the conclusion states that the output interval is an interval; the next conjunct states that the domain of the interval is `INTEGERP`. The next two conjuncts state that the lower and upper bounds of the output interval are non-`nil`, which (because of the first two conjuncts) means they

are both integers, the lower bound is weakly below the upper one, and the interval is closed.[7]

The third theorem establishes that for pseudo term x such that `Ainni`'s output flag is non-`nil` and all of the output hypotheses are true (i.e., the evaluator $\mathcal{E}$ evaluates the conjunction of those terms to non-`nil`), then the value (under $\mathcal{E}$) of x is contained in the output interval.

```
(implies                                                ; {Ainni 3}
 (and (pseudo-termp x)
      (mv-nth 0 (Ainni x hyps ctx))
      (E (conjoin (mv-nth 1 (Ainni x hyps ctx)))
         alist))
 (in-tau-intervalp (E x alist)
                   (mv-nth 2 (Ainni x hyps ctx))))
```

The fourth theorem establishes that `Ainni` actually preserves the internal invariants on ACL2 terms, i.e., that if the input term and the elements in the input hypotheses each satisfy ACL2's internal invariant then the output hypotheses satisfy that invariant. The constant `*stateman-arities*` is an alist pairing each of the function symbols known to $\mathcal{E}$ with its arity.

```
(implies                                                ; {Ainni 4}
 (and (termp x w)
      (term-listp hyps w)
      (arities-okp *stateman-arities* w))
 (term-listp
  (mv-nth 1 (Ainni x hyps ctx))
  w))
```

This last theorem allows ACL2 to avoid checking that the output hypotheses satisfy the internal invariants on terms. Instead, ACL2 just has to check that each of the function symbols listed in `*stateman-alist*` has the given arity in ACL2's then-current logical theory.

`Ainni` is closely related to the Tau System in ACL2. See :DOC `tau-system`. Tau is a user extensible abstract interpreter over sets of monadic predicates describing the types of values returned by an expression. It includes containment in constant intervals as a "type." ACL2 users think of the Tau System as a quick, incomplete "type checker" for the untyped language of ACL2. By design, the Tau System answers yes/no questions: is this formula trivial by type-like reasoning?

`Ainni` is designed to answer quantitative questions: What are the minimal and maximal values of this expression? `Ainni` exploits some of the same theorems (in the elementary bounders book) used to extend Tau. But by

[7] By definition of `tau-intervalp`, any interval with `INTEGERP` domain has integers for its bounds unless there is no bound (i.e., a "bound" of `nil`) in some direction. Furthermore, all bounded integer intervals are, by convention, closed. That is, if the domain is `INTEGERP` then instead of, say, [0,8) we use [0,7].

defining `Ainni` in the logic and verifying it, we make it possible to use interval reasoning during rewriting.

7 Some Examples

Consider this expression:

`(+ 2000 (* 8 (LOGAND 31 (R 1000 8 `*st*`)))).`

This is the formal expression of a fairly typical machine address encountered in symbolic code evaluation. It corresponds to the compiled version of an array element reference, where the base address of the array is `2000`, the array consists of quadword (8-byte) elements, and the index is formed by taking the bottom 5 bits of the quadword at address `1000`. The prevalence of constants in the expression is also quite common when exploring code recovered from an actual machine image: the locations of data are fixed or at computed offsets from fixed addresses like the initial stack pointer.

What can `Ainni` tell us about this expression? We answer that by evaluating

`(Ainni '(+ 2000 (* 8 (LOGAND 31 (R 1000 8 `*st*`)))) nil nil)`

Ainni will return three values. Its output flag will be `T`, the list of output hypotheses will be `nil`, and the output interval will be the ACL2 data structure that represents the integer interval [2000, 2248].[8]

The derivation of the output interval is as follows: `(R 1000 8 `*st*`)` is known to be in $[0,2^{64}-1]$, but the `LOGAND` is in [0,31]. Thus, the product with `8` is in the interval [0,248], so the sum with 2000 is in [2000, 2248].

Now imagine *ctx* contains the assumption that `(R 1000 8 `*st*`)` is below 16 and reconsider

`(Ainni '(+ 2000 (* 8 (LOGAND 31 (R 1000 8 `*st*`)))) nil `*ctx*`).`

This time the output flag will be `T`, there will be one output hypothesis, namely `(<= (R 1000 8 `*st*`) 15)`, and the output interval will be [2000, 2120]. The third correctness theorem for `Ainni` assures us that `(+ 2000 (* 8 (LOGAND 31 (R 1000 8 `*st*`))))` lies in [2000, 2120] provided `(<= (R 1000 8 `*st*`) 15)` is true.

Finally, to demonstrate `Ainni`'s speed compared to ACL2's more powerful arithmetic, consider the expression

```
(LOGIOR (ASH (MOD (R 1000 4 ST) 2) 0)
        (ASH (MOD (R 1004 4 ST) 2) 1)
```

[8] As noted earlier, the actual input to `Ainni` should be in ACL2's internal form, so, for example, the "`+`" should be `binary-+` and the numbers should be quoted. The data structure representing the output interval is `(INTEGERP (NIL . 2000) . (NIL . 2248))`, indicating an integer domain, bounded above and below by `2000` and `2248` respectively. The `NIL`s indicate that $\leq$ rather than $<$ is used to check whether a number is in bounds.

```
(ASH (MOD (R 1008 4 ST) 2) 2)
...
(ASH (MOD (R 1052 4 ST) 2) 13)
(ASH (MOD (R 1056 4 ST) 2) 14)
(ASH (MOD (R 1060 4 ST) 2) 15)).
```

The value of this expression lies in the interval $[0, 2^{16} - 1]$ regardless of the values of the `R`-expressions. Any programmer would realize the expression is bounded above by 2^{16}: each `MOD` is just a single bit, and the expression shifts those bits into positions 0–15. Using similar "forward" reasoning from the expression, `Ainni` computes the interval $[0, 2^{16} - 1]$ in 0.012 seconds on a MacBook Pro laptop with a 2.6 GHz Intel Core i7 processor running ACL2 in CCL.

On the other hand, *proving* the expression is so bounded can feel harder! Indeed, it takes the same laptop about 1306 seconds to use the standard ACL2 arithmetic library from the Community Books (`books/arithmetic-5/top`) to prove that the expression above is less than 2^{16}. The library splits the goal into 2^{16} cases.

Of course, ACL2's arithmetic library is much more powerful than `Ainni`. The library is essentially a collection of theorems about arithmetic/logical functions which informs the ACL2 rewriter and its integrated linear arithmetic decision procedure. Those systems can be made to prove anything that is provable about ACL2 arithmetic, whereas `Ainni` is much more limited. But we embarked on the development of `Ainni` because we saw the importance of a verified tool to look at typical machine arithmetic expressions and do what every programmer can do: bound it by interval reasoning. In addition, `Ainni` is fast.

The expression above is small compared to expressions encountered when doing code analysis, especially of long sequences of machine instructions. The expression above has 63 function calls in it (when the `LOGIOR` macro is expanded into a right-associated nest of calls of `BINARY-LOGIOR`) of which 16 are calls to `R` and the rest are calls to logical functions. By contrast, the largest arithmetic/logical expression encountered in the disassembly of a machine code implementation of the DES algorithm is a term involving 147,233 function applications, 31,361 of which are calls of `R` and the rest are calls of arithmetic/logical functions. `Ainni` can bound that very large expression in about 0.01 seconds. It is completely impractical to use the standard arithmetic library to confirm the correctness of that answer (other than by relying on the verified correctness of `Ainni`).

But another major advantage of `Ainni`, aside from being very fast and quite capable on huge expressions, is that it *discovers* bounds whereas the rest of ACL2 (e.g., the Tau System) is oriented toward *proving* things. That is, ACL2 is generally used to answer specific Boolean questions, e.g., "Does this value fit in 16-bits" whereas `Ainni` gives it the capability of answering quantitative questions such as "How big is this?" These advantages mean `Ainni` can effectively be used in simplification.

8 Using Ainni in a Metafunction

Because `Ainni` has been proved correct by ACL2, it can be used in metafunctions which are in turn used by the rewriter. Thus, one need not choose between `Ainni` and a conventional rewrite-driven arithmetic book; one can have both.

Here is a very simple metafunction that shows how we use `Ainni`. The following function simplifies `(MOD` x `'`k`)` expressions, where k is some natural constant, using the fact that `(MOD` x `'`k`)`$= x$, if x is an integer less than k.

```
(defun mod-constant-simplifier (term mfc state)
  (declare (ignore state))
  (cond
   ((and (not (atom term))
         (eq (car term) 'MOD)
         (not (atom (nth 2 term)))
         (eq (car (nth 2 term)) 'QUOTE))
    (let ((x (nth 1 term))
          (k (nth 1 (nth 2 term)))
          (ctx (mfc-type-alist mfc)))
      (cond
       ((and (natp k)
             (syntactic-natp x))
        (mv-let
         (flg hyps int)
         (Ainni x nil ctx)
         (cond
          ((and flg
                (< (tau-interval-hi int) k))
           (list 'IF (conjoin hyps) x term))
          (t term))))
       (t term))))
   (t term)))
```

This function checks that *term* is a call of `MOD` and that the second argument is a quoted constant. If so, it binds x to the first argument of the `MOD` and k to the constant. It also extracts the type-alist from the metafunction context mfc and binds the variable *ctx* to that. Then it checks that k is a natural number and x is a syntactic natural. If so, it calls `Ainni` and if `Ainni` reports success and thm upper bound of the resulting interval is below k, it creates and returns an `IF`. The test of the `IF` is the conjunction of the output hypotheses, the true branch is x, and the false branch is *term*.

The correctness of this metafunction follows from the correctness of `syntactic-natp` and `Ainni` and the previously mentioned fact about `(MOD` x `'`k`)`. Once verified and installed as a metafunction for `MOD`, `mod-constant-simplifier` is run on every `MOD` expression and, when it returns something

different from its input, the theorem prover backchains to establish the truth of the tested output hypotheses and if so replaces the target term with x.

For example, once `mod-constant-simplifier` is verified as a metafunction for `MOD`, the expression:

```
(MOD (LOGIOR (ASH (MOD (R 1000 4 ST) 2) 0)
             (ASH (MOD (R 1004 4 ST) 2) 1)
             (ASH (MOD (R 1008 4 ST) 2) 2)
             ...
             (ASH (MOD (R 1052 4 ST) 2) 13)
             (ASH (MOD (R 1056 4 ST) 2) 14)
             (ASH (MOD (R 1060 4 ST) 2) 15))
     (EXPT 2 24))
```

immediately simplifies to the `LOGIOR` expression.

9 Other Uses of Ainni

While `Ainni` was developed for answering questions about machine addresses it is generally useful for answering quantitative questions about formal machine arithmetic as illustrated in the previous section.

Another very helpful use of `Ainni` is in a metafunction to simplify `(<` x y`)`. Triggered by the less than operator, `<`, the metafunction uses `Ainni` on x and y and if `Ainni` succeeds the metafunction can use quick checks on the endpoints to often reduce the `(<` x y`)` to `T` or to `NIL`. The comparable reduction by the native rewriter and its linear arithmetic procedure involves duplication of effort, essentially trying to rewrite both the inequality and its negation since only Boolean questions can be asked of them.

The motivating applications for `Ainni` were metafunctions to handle read-over-write and write-over-write expressions. Consider a read-over-write. Typically, the write expression is a deep nest of `!R` expressions. The metafunction uses `Ainni` on the read address and extent to compute the interval containing the region to be read. Then with that interval in hand it searches down the nest of writes comparing the read interval to the write intervals (using `Ainni` on each write address and extent). Quick checks on the resulting intervals can determine when the regions are disjoint – without having to reanalyze the addresses to determine whether the read is "above" or "below" the write.

Thus, `Ainni` allows the read-over-write metafunction to be much more efficient than rewrite rules because the read address and each write address is analyzed just once. This illustrates a major advantage of being able to answer a quantitative question rather than just a Boolean one.

ACL2 permits memoization and that has proven helpful in avoiding repeated calls to `Ainni` on the write addresses. However, we found that it was

best to memoize the read-over-write metafunction itself rather than the individual calls of `Ainni` inside it.[9]

The details of the metafunctions using `Ainni` may be found by looking at the heavily commented proof script in the ACL2 Community Book `books/projects/stateman/stateman22.lisp`.

10 Related Work

Simplification and abstract interpretation are so ubiquitous it is beyond the scope of this paper to offer much background on them. Basically every mechanized prover has libraries or tactics or built-in routines to simplify formulas using various standard heuristics to control inference; see "auto" in Coq and HOL and the built-in notion of "simplification" in PVS. The name "abstract interpretation" was introduced by the Cousots in 1977 [8] and is basically the generalization of an operational semantics or interpreter to deal with conservative approximations of the actual data (e.g., intervals instead of numbers). Type checking is an example of abstract interpretation.

The work most closely resembling that reported here is probably the Astrée static analyzer [9]. Astrée aims at proving the absence of run time errors in programs written in C. It is based on abstract interpretation and uses interval analysis to approximate numeric data values.

However, Astrée is a standalone static analyzer whose input is a C program, whereas `Ainni` is a user-defined and mechanically verified extension of the ACL2 simplifier. While both are relying on abstract interpretation, Astrée interprets C programs (including its arithmetic/logical expression language) while `Ainni` only interprets arithmetic/logical expressions in the ACL2 logic. The program control and data manipulation done by Astrée is, in our case, done by the ACL2 system, specifically its simplifier applied to the formal operational semantics and the object code. So there are really two different abstract interpreters involved in our code proofs, one over the program text (done by the simplifier) and one over the semantics of arithmetic/logical expressions (done by `Ainni`), and in Astrée they are combined. One presumes that Astrée contains an abstract interpreter for arithmetic/logical expressions that produces interval bounds on those expressions.

Finally, `Ainni` is available as a mechanically verified extension of the ACL2 simplifier and is hence of use in any theorem proving setting requiring reason-

[9] One could memoize the ACL2 rewriter itself and hope to speed up the rewrite-rule approach. However this has been unsuccessful because the ACL2 rewriter takes so many arguments to record the context, the objective of the rewrite, equivalence relations to be maintained, histories used to avoid infinite backchaining and looping, stacks to track the lemmas used for reporting purposes, counters to measure or limit the work done, etc. All these extra arguments mean that identical calls to rewrite virtually never occur and so memoization costs more time than it saves. `Ainni` and its callers use far fewer arguments and memoization is effective on them.

ing about the bounds of arithmetic/logical expressions. Furthermore, `Ainni` has been mechanically verified to be correct by ACL2.

11 Conclusion

We have described an ACL2 function, `Ainni`, for answering the quantitative question "what are the minimal and maximal magnitude of the value of this expression?" The function is an abstract interpreter for machine arithmetic expressions composed of arithmetic/logical operators and interprets them over bounded closed natural number intervals. `Ainni` can be thought of as a type inference procedure where the types are intervals.

`Ainni` has been verified with ACL2 to be correct and can therefore participate in formal proofs. The vehicles for that participation are metafunctions designed to simplify machine arithmetic expressions.

`Ainni` has allowed ACL2 to do symbolic exploration of sequences of realistic machine code containing thousands of instructions, whose end states contain millions of function applications. This was not possible using other techniques we have tried with ACL2.

The success of `Ainni` has raised important new questions: how can the rest of ACL2 be made to cope with the expressions now being produced? This is a welcome — and very typical — step along the evolutionary path ACL2 has followed. A solution to one scaling problem introduces new scaling challenges.

12 Acknowledgments

I would especially like to thank Warren Hunt for his invaluable help during the development of `Ainni`. Warren developed the definitions and proved many of the basic rewrite rules for the byte addressed read and write functions, `R`, and `!R`. He also provided an ACL2 formalization of a realistic ISA and implemented the DES algorithm in ACL2. We then compiled the DES algorithm into the instructions of the ISA thus obtaining an interesting symbolic evaluation challenge for ACL2. I would also like to thank Matt Kaufmann, who gave me some strategic advice on lemma development to prove the correctness of one of the metafunctions here as well as his usual extraordinary efforts to maintain ACL2 while I pursue topics such as this one. This work was partially supported by ForrestHunt, Inc.

References

1. W.R. Bevier, W. A. Hunt, Jr., J S. Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.
2. R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
3. R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. Technical Report CSL-108, SRI International, December 1979.
4. R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. Academic Press, London, 1981.
5. R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.
6. R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.
7. B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293, Heidelberg, November 1996. Springer-Verlag, LNCS 1166. `http://www.cs.utexas.edu/users/moore/publications/bkm96.ps.Z`.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. The astre analyser. In M. Sagiv, editor, *European Symposium on Programming (ESOP 2005)*, LNCS 3444, pages 21–30. Springer, April 2005.
10. S. Goel, W.A. Hunt, and M. Kaufmann. Simulation and formal verification of x86 machine-code programs that make system calls. In K. Claessen and V. Kuncak, editors, *FMCAD'14: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 91–98. EPFL, Switzerland, 2014.
11. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.
12. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
13. M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
14. M. Kaufmann and J S. Moore. The ACL2 home page. In *`http://www.cs.utexas.edu/users/moore/acl2/`*. Dept. of Computer Sciences, University of Texas at Austin, 2014.
15. M. Kaufmann and J S. Moore. Well-formedness guarantees for ACL2 metafunctions and clause processors. In *Design and Implementation of Formal Tools and Systems (DIFTS) 2015*, 2015.
16. H. Liu and J S. Moore. Java program verification via a JVM deep embedding in ACL2. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *17th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 184–200. Springer, 2004.
17. J S. Moore and M. Martinez. A mechanically checked proof of the correctness of the boyer-moore fast string searching algorithm. In *Engineering Methods and Tools for Software Safety and Security (Proceedings of the Martoberdorf Summer School, 2008)*, pages 267–284. IOS Press, 2009.

18. Anna Slobodova, Jared Davis, Sol Swords, and Jr. Warren Hunt. A flexible formal verification framework for industrial scale validation. In Satnam Singh, editor, *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 89–97. IEEE, 2011.
19. G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.
20. E. Toibazarov. An ACL2 proof of the correctness of the preprocessing for a variant of the boyer-moore fast string searching algorithm. Honors thesis, Computer Science Dept., University of Texas at Austin, 2013. See `www.cs.utexas.edu/users/moore/publications/toibazarov-thesis.pdf`.
21. M. Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Heidelberg, 1993. Springer-Verlag.